Formalising the Timebands Model in Timed *Circus*

Kun Wei
Jim Woodcock
Alan Burns

Department of Computer Science
University of York, UK.

Technical Report

August 2010

Abstract

Complex real-time systems exhibit dynamic behaviours on many different time levels. To cope with the wide range of time scales and produce more dependable computer-based systems, we develop a timebands model that can explicitly recognise a finite set of distinct time bands in which temporal properties and associated behaviours are described. In general, the common way to deal with multiple-level behaviour of a system is to embed time granularity into a logical specification. Therefore, we propose a timed version of *Circus* to embody time granularity. Timed *Circus* is a compact extension to *Circus*; that is, it inherits only the CSP part of *Circus* while introducing time. However, this model is very different from timed CSP, since it uses a complete lattice instead of the expected complete partial order. The complete lattice gives rise to a number of strange processes which can violate many fundamental assumptions of standard CSP, when the miracle, the top element in the complete lattice, is combined with various operators. The application of the miracle is the kernel of the new model that constructs all brand-new features of the timebands model such as simultaneous events and punctual clocks.

We formalise the timebands model by using the semantics of the timed *Circus* to guarantee soundness of each operator and maintain consistency and coordination between different time bands. By means of two rather complicated examples, the mine pump and the banking system, we demonstrate how significantly the timebands model contributes to describing and specifying complex real-time systems with multiple time scales. We believe that the modelling with a time-based hierarchy is able to help develop a comprehensive foundation to dependable systems.

Contents

Abstract **i**

1 Introduction **1**

2 Timed *Circus* **2**
2.1 Background . . . 2
2.2 Reactive designs . . . 3
2.3 Semantics . . . 7
2.4 Applications . . . 14
2.4.1 Urgent and uninterrupted events . . . 15
2.4.2 Instant events and punctual clocks . . . 15
2.4.3 Shared variables . . . 17

3 Semantics of the Timebands Model **18**
3.1 Time bands . . . 18
3.2 Punctual clocks . . . 19
3.3 Events and precision . . . 20
3.4 Timeless traces . . . 22
3.5 Explicit clock-tick events . . . 23
3.6 Activities . . . 25
3.7 Mappings and abstractions between bands . . . 26
3.8 Integrating time bands . . . 28
3.9 Discussion . . . 29

4 Case Study **30**
4.1 Mine pump . . . 31
4.2 Banking system . . . 35

5 Conclusion **38**

1 Introduction

Complex real-time systems exhibit dynamic behaviours on many different time levels. For example, the circuits have nanosecond speeds for computation in a component, whereas the slower functional units may take seconds to achieve their goals; furthermore, the involvement of human activities related to calendar units such as days, weeks, months and even years may cost more time between interactions of operators and a system. To cope with the wide range of time scales, many approaches have introduced time granularity, so that system specifications and requirements could be naturally described within best suitable time scales. However, they usually transform or project all descriptions into a finest time granularity in the end. Such an abstraction fails to support the structure properties of systems, sacrifices the separation of concerns in the analysis of complex real-time systems and lacks efficiency in implementation.

To overcome the above weakness when traditional approaches model dynamic behaviours of a system in multiple time scales, Burns and Hayes [3] 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 certain entities are considered to be instantaneous in a band, and they are then mapped to actions that have durations in a finer band. It allows dynamic properties to be partitioned but not isolated from each other.

The concept of time granularity has been well defined in the literature [7, 18] and many researches have been conducted on time granularity with different areas of computer science, such as temporal databases, date mining, formal specification and so on. General speaking, the basic idea of time granularity is to partition a universal time domain into differently-grained granules, and that, a granularity is a set of indexed granules, any one of which is a set of time instants. The choice for time domain is typically between continuous (dense) and discrete.

We focus on developing a natural specification language which is able to specify and verify the behaviour of a real-time system whose components engage in different time scales. In other words, we attempt to embed time granularity in a logical specification language. However, augmenting time granularity to a formalism may give rise to semantic issues like problems of assigning a proper meaning to statements with different time domains and of switching one domain to a coarser/finer one. So far, most of work has been focused on embedding time granularity in temporal logic languages. For instance, early exploration [9, 26] consists of translation mechanisms that map a formula associated with different time constraints to the finest granularity. They [6] later revise the simple approach by extending the basic logic language with contextual and projection operators, so that the enhanced semantics can express more general and complete properties. More other work [8, 11] for reasoning about time granularity is proposed as well.

There is little work on embedding time granularity in process algebra languages. In the timebands model, the behaviour of a system is mainly described by events and activities. Therefore, *Circus* [34, 42] naturally becomes a potential specification language to embody different time scales, because it unifies CSP [16, 32, 35] and Z [38, 43] in order to define both data and behavioural aspects of a system. To formalise the timebands model, we propose a timed

model of *Circus* whose semantics is based on *Unifying Theories of Programming*(UTP). Apart from introducing time, this model uses a complete lattice with respect to the implication ordering, which is rather different from previous models such as the complete partial order of CSP. In our timed *Circus*, each process is described as a reactive design, and the reactive design miracle, the top element, is exploited to define new operators such as *deadline* and thereby provides us with more power and flexibility in system specifications.

The semantics of the timebands model is built upon the timed *Circus* model, fully applying the miracle to express those brand-new features such as simultaneous events and mappings. The informal description of the timebands framework has been given in [3] by a formal model expressed in the Z notation, and the formal semantics of the framework is developed in the report.

2 Timed *Circus*

2.1 Background

Real-time systems are rather complicated as their components may execute in parallel and may also interact with their physical environment, as well as satisfy certain critical timing requirements. Over the past decades, formal methods have been remarkably successful in their application to the analysis of real-time systems. Broadly speaking, the approaches to the analysis of real-time systems have fallen into two camps. Process algebra approaches, such as timed CSP [31, 35], describe the behaviour of a system in terms of events, mostly addressing the interaction of its components. Temporal logic approaches, such as DC [5] and RTL [21], are convenient in describing the change of states but lack support for concurrency.

Recently the combination of different approaches by means of unifying their semantics has been developed in order to tackle a wider variety of systems. *Circus* is one of the successful combinations, which unifies CSP [16, 32, 35] and Z [38, 43], so that it can define both data and behavioural aspects of a system; that is, it can describe the change of states and define data operations while dealing with concurrency. A well-defined syntax and a sound semantics [34, 42] have been given in *Circus*, based on the semantics of UTP [17]. Sherif and He [36] subsequently develop a timed model of *Circus* that takes a subset of *Circus* and creates an abstraction function to map the timed model to the original untimed model.

In real-time programming, the deadline command [10] is a simple and flexible language primitive to directly express the desired timing behaviour. A similar operator is also provided in process algebra approaches such as timed CSP by means of the timeout operator. However, the occurrence of events in timed CSP depends on their environment's interaction; in other words, we cannot specify that 'something must occur'. When formalising observation of complex real-time systems, there are a variety of factors, such as physical laws and physical variables, which are part of systems but beyond the control of any other participants. For instance, in modelling a scheduling system, a constant sampling frequency is achieved by executing the sampling tasks at precise points of time. Therefore, we possibly need to define a punctual clock whose clock-tick events *must* arrive on time. It is also very convenient to define uninterrupted events

in specification, which usually mean that the execution of a sequence of events cannot be interfered. Our new timed *Circus* is able to elegantly describe such behaviours that something must take place.

Woodcock et al [30] recently have proposed new semantics, also based on UTP, for *Circus* in which each process is described as a reactive design. The so-called reactive designs come from the fact that the new semantics applies the well-defined healthiness conditions of reactive processes to embed the theory of designs. The first denotational semantics, also based on UTP, was published in [42]. However it actually describes *Circus* programs as Z specifications in order to use tools like Z/EVES [33] to reason about properties. Unfortunately, this semantics is insufficient to prove refinement laws. The new semantics completely overcomes the weakness, and even the sophisticated refinement laws of CSP can be adopted in the refinement of *Circus* specifications. Our timed model, built on such new semantics, is a compact extension of *Circus*. It does not inherit Z specifications and is in fact closer to timed CSP in syntax, but partially preserves the ability of *Circus* to handle data by allowing processes to hold a set of local variables.

There is a lot of related work that combines two or more well-researched languages for specifications of behaviour, data and time. For example, He [15] proposes a hybrid system to integrate CSP and DC for specifying the behaviour of continuous devices in a digital world, in which CSP describes the interaction of its components, whereas DC is for specification of continuous changes of states. Moreover, Sherif and He [36] develop a timed model of *Circus* that takes a subset of *Circus* and creates an abstraction function to map the timed model to the original untimed model. The semantics of the two models is also based on the theory of alphabetised relations in UTP. By comparison, our timed model uses reactive designs (the sub-theory of relations built upon applying corresponding healthiness conditions to relations) to describe processes, and exploits the top element of the complete lattice to model some brand-new properties of a system.

Similar to *Circus*, many combinations focus on integrating CSP and Z or their extensions such as Timed CSP and Object-Z [37]. For instance, Hoenicke and Olderog propose a combination CSP-OZ-DC in which CSP specifies behaviour of processes and their communications, Object-Z describes data and state information, and DC captures real-time constraints. Different from the semantics of our timed model of *Circus*, their model translates all specifications of OZ and DC into a form of traces and acceptance sets and then uses a combined application of the model checkers FDR [1] and UPPAAL [22] to verify properties.

2.2 Reactive designs

In UTP, Hoare and He use the alphabetised relational calculus to give a denotational semantics that can explain a wide variety of programming paradigms. A relation P is a predicate with an alphabet 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 $in\alpha P$.

Following the theory of alphabetised relations, many interesting sub-theories are constructed by applying healthiness conditions to characterise different as-

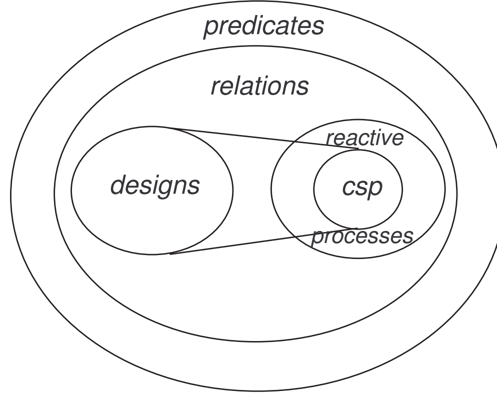

Figure 1: The relation of theories in UTP

pects of the sub-theories. Hoare and He first build the theory of designs within the relational calculus. They subsequently build the theory of reactive processes that is disjoint from the theory of designs. The theory of CSP processes results from using reactive healthiness conditions to embed designs. Our timed model can also be considered an extension of the theory of CSP[1], in which each process is described as a reactive design. The relation of these theories in UTP can be further illustrated in Figure 1 which is cited from [29].

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.

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

$$H1(P) = (ok \Rightarrow P)$$

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

The second healthiness condition is

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

[1]The theory of CSP in UTP is different from those in Hoare's book [16] and Rocsco's book [32].

where square brackets denote universal quantification over all the variables in the alphabet. $H2$ can also be rewritten by removing the universal quantifier,

$$P = P\,;\,J$$

where $J \mathrel{\widehat=} (ok \Rightarrow ok') \wedge v' = v$ and the semicolon denotes the sequential composition. Note that v and v' in the alphabet of J denotes all the variables except for ok and ok'. $H2$ actually states a design cannot require nontermination, since if P is satisfied when ok' is false, it must also be satisfied when ok' is true. Thus, a design is a relation that is $H1$-$H2$-healthy. For a tutorial introduction to designs, the reader is referred to [44].

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, 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 trace is a sequence of timed events ordered by time.
- 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 local variables, and v' records the final values.

As a result, we are able to represent any case of states of a process by combining the values of ok and $wait$. For instance, 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 deadlocked; otherwise it successfully terminates if $wait'$ is false. Similarly, the values of undashed variables represent the states of a process's predecessor. For a more detailed introduction to the theory of reactive designs, the reader is referred to the tutorial [4].

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

$$R1: \quad P = P \wedge tr \le tr'$$

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 an empty sequence.

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

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

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

$$R3: \quad P = II_{rea} \lhd wait \rhd P$$

where the reactive identity, II_{rea}, is defined as follows:

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

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 can elapse. Further, if P and Q are predicates, $P \lhd b \rhd Q$ describes a program which behaves like P if the condition b is true, or like Q if b is false.

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

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

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

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

Hoare and He have given a new semantic to CSP in their UTP book [17] where the theory of CSP is a complete lattice, rather than the complete partial orders of the standard models of CSP [16, 32]. In UTP, the theory of CSP is

built by applying few healthiness conditions to reactive processes. For example, a CSP process is an reactive process that satisfies $CSP1$ and $CSP2$:

$$CSP1(P) = P \vee (lnotok \wedge tr \leq tr' \wedge t \leq t')$$
$$CSP2(P) = P \,;\, J$$

where the first healthiness condition requires that, in case of divergence, the extension of traces and times should be the only guaranteed property. The second one means that P cannot require nontermination, so that it is always likely to terminate.

However it can also be achieved by using the healthiness conditions R to embed designs within the theory of reactive processes. As an extension of the theory of CSP, the theory of our timed model is built by the same approach, in which processes are represented in the form of R-healthy designs. For example, the reactive design miracle $\top_R$ is defined by means of the design miracle made R-healthy:

$$\begin{aligned}
\top_R &\mathrel{\widehat{=}} R(true \vdash false) && \text{[design]}\\
&= R(ok \wedge true \Rightarrow ok' \wedge false) && \text{[true unit for conjuction]}\\
&= R(ok \Rightarrow false) && \text{[contradiction]}\\
&= R(\neg\, ok)
\end{aligned}$$

The top element of the theory of CSP in UTP, the reactive design miracle, had been completely unexplored until Woodcock [41] gave an insight into the nature of the reactive design miracle and provides a few interesting applications of miracles. Our new timed model further applies the reactive design miracle to show how useful it might be in developing timed reactive systems.

2.3 Semantics

As an extension of *Circus*, our timed model is very close to timed CSP in syntax but capable of describing data and time. Most of the operators come from timed CSP, and the operators of assignment and guarded processes are obtained from *Circus*. A new operator, deadline, is defined by means of the reactive design miracle. The reactive design semantics based on UTP has been firstly proposed for CSP processes in [4], and the similar semantics for *Circus* in [42, 30]. Here we rewrite the semantics on account of the involvement of time.

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

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

Reactive design miracle

The reactive miracle can never be executed and it is defined in terms of the design miracle made R-healthy. This semantics can be further simplified somewhat [2]:

[2] 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.

$$
\begin{aligned}
&\top_R && \text{[definition]}\\
=&R1 \circ R3j \circ R4(true \vdash false) && [R3j^3]\\
=&R1 \circ R4((true \vdash I\!I) \lhd wait \rhd (true \vdash false)) && \text{[design-conditional]}\\
=&R1 \circ R4((true \lhd wait \rhd true) \vdash (I\!I \lhd wait \rhd false)) && \text{[p. c.]}\\
=&R1 \circ R4(true \wedge ok \Rightarrow ok' \wedge I\!I \wedge wait) && \text{[p. c.]}\\
=&R1 \circ R4(\neg\, ok \vee (wait \wedge ok' \wedge I\!I)) && [R1\text{-}R4]\\
=&R1 \circ R4(\neg\, ok) \vee (wait \wedge ok' \wedge I\!I) && \square
\end{aligned}
$$

The explanation of the simplified miracle in our timed model can help us understand what on earth the miracle is. 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.

Chaos

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

$$\bot_R \mathrel{\widehat=} R\,(false \vdash true) = R(true)$$

Stop

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

$$STOP \mathrel{\widehat=} R\,(true \vdash tr' = tr \wedge wait' \wedge v' = v)$$

Skip

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

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

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

Sequential composition

The definition of sequential composition is the same as the one in alphabetised relational calculus. If two processes P_1 and P_2 are in sequence, P_1 is executed

[3] Woodcock [41] 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 $R1$.

firstly and then P_2 once P_1 terminates, and meanwhile the final state of P_1 is passed on as the initial state of P_2.

$$P_1 \,;\, P_2 \mathrel{\widehat=} \exists\, x_0 \bullet P_1[x_0/x'] \wedge P_2[x_0/x]$$

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

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

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

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

Law 1 states that the reactive design miracle is a left-zero for relational composition and Law 2 says that the miracle is a right-zero for the reactive design identity. Their proofs are non-trivial and given in Appendix.

Prefix

The process $a \rightarrow P$ in CSP denotes the process that behaves like P after performing the event a. The prefix process in UTP can be represented by a composition of a simple prefix and P itself, written as $(a \rightarrow SKIP)\,;\, P$. As the reactive design definition of simple prefix in [30], its timed version can be expressed as follows:

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

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

Assignment

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

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

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

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

where P^a_b represents $P[a/ok'][b/wait]$, and the abbreviation follows the definitions throughout this paper. For instance, P^f_f denotes $P[false, false/ok', wait]$, saying that P diverges while not waiting for its predecessor to terminate.

External choice

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

$$\begin{aligned} P_1 \Box P_2 \mathrel{\widehat=} R\,((\neg\, {P_1}^f_f \wedge \neg\, {P_2}^f_f) \\ \vdash ({P_1}^t_f \wedge {P_2}^t_f \lhd tr' = tr \wedge wait' \rhd {P_1}^t_f \vee {P_2}^t_f)) \end{aligned}$$

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

There are two interesting laws with respect to the external choice of two primitive processes and the miracle. They are obvious but useful to understand the role of the miracle. The proof of Law 3 is given in Appendix and the one of Law 4 is left as an exercise for the interested reader.

Law 3. $STOP \Box \top_R = \top_R$

Law 4. $SKIP \Box \top_R = SKIP$

Input and output

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

Internal choice

The internal choice $P_1 \sqcap P_2$ can behave either like P_1 or like P_2, but it is out of control of its environment. It can be simply defined as $P_1 \sqcap P_2 \mathrel{\widehat=} P_1 \vee P_2$. Since the reactive design miracle is the top element of the complete lattice or it should be included in any reactive design process, the following law is easily proved.

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

Parallel composition

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

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

$$\exists\, 1.tr', 2.tr' \bullet (P_1{}^f_f; (1.tr' = tr)) \wedge (P_{2f}; (2.tr' = tr)) \\ \wedge (1.tr' - tr) \upharpoonright A = (2.tr' - tr) \upharpoonright A$$

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

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

$$P_1 \underset{A}{\parallel} P_2 \mathrel{\widehat=} R \left(\begin{array}{c} \neg\ \exists\, 1.tr', 2.tr' \bullet (P_1{}^f_f; (1.tr' = tr)) \wedge (P_{2f}; (2.tr' = tr)) \\ \wedge\ 1.tr' - tr \upharpoonright A = 2.tr' - tr \upharpoonright A \\ \wedge \neg\ \exists\, 1.tr', 2.tr' \bullet (P_{1f}; (1.tr' = tr)) \wedge (P_2{}^f_f; (2.tr' = tr)) \\ \wedge\ 1.tr' - tr \upharpoonright A = 2.tr' - tr \upharpoonright A \\ \vdash \\ ((P_1{}^t_f;\ U1(out\alpha P_1)) \wedge (P_2{}^t_f;\ U2(out\alpha P_2)))_{+\{v,tr\}}\ ;\ M_{\parallel}(A) \end{array} \right)$$

In the postcondition $P_1{}^t_f$ and $P_2{}^t_f$ denote that they are not divergent. The labelling process $Ul(m)$ simply passes dashed variables of its predecessor to labelled variables, which is defined as $Ul(m) \mathrel{\widehat=} \mathbf{var}\ l.m := m;\ \mathbf{end}\ m$ where $\alpha\, Ul(m) \mathrel{\widehat=} \{m, l.m'\}$. Notice that $Ul(m)$ only obtains the values of dashed variables of its predecessor. However under some circumstances we do need the initial values of its predecessor's variables. For this reason, we expand the alphabet after the labelling process. For example, $P_{+\{n\}}$ denotes $P \wedge n' = n$. Here we are only interested in v and tr that are used in $M_{\parallel}$.

The process $M_{\parallel}$ merges the traces and refusal sets from its predecessor in regard to the interface A, and also combines all other variables, defined as follows:

$$M_{\parallel}(A) \mathrel{\widehat=}\ tr' - tr \in (1.tr - tr \underset{A}{\parallel} 2.tr - tr) \wedge t' = max(1.t, 2.t) \\ \wedge (1.tr - tr) \upharpoonright A = (2.tr - tr) \upharpoonright A \\ \wedge \left(\begin{array}{c} (1.wait \vee 2.wait) \wedge ref' \subseteq MRef \\ \lhd\ wait' \rhd \neg\ 1.wait \wedge \neg\ 2.wait \wedge MSt \end{array} \right)$$

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

An interleaving of two processes P_1 and P_2 executes each part independently and is equivalent to $P_1 \underset{\emptyset}{\parallel} P_2$. 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. The proof of Law 6 is given in Appendix.

Law 6. $P \underset{A}{\parallel} \top_R = \top_R$

Hiding

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

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

The hiding may introduce divergence; therefore, the process *SKIP* is used to capture the observation of possible divergences. With the maximal progress requirement that an event occurs when all participants are ready, internal events (hidden events) must occur as soon as they are enabled. The above definition can guarantee that hidden events become urgent.

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:

$$\begin{aligned} \mathit{WAIT}\ d \mathrel{\widehat{=}} R\,(true \vdash tr' = tr \wedge v' = v \wedge ((wait' \wedge t' - t < d) \\ \vee (\neg\, wait' \wedge t' - t = d))) \end{aligned}$$

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

Timeout

The time-sensitive choice $P_1 \rhd \{d\} P_2$ resolves the choice in favour of P_1 if P_1 is able to execute observable events by the time d, or resolves the choice in favour of P_2. Here, we use a similar definition given in [35]:

$$P_1 \rhd \{d\} P_2 \mathrel{\widehat{=}} (P_1 \;\Box\; \mathit{WAIT}\ d\,;\, e \rightarrow P_2) \setminus \{e\} \qquad e \notin \alpha P_1$$

which uses the event e to resolve the external choice if no external event from P_1 occurs within d.

Deadline

Compared with the above timeout operator, the deadline operator claims that P *must* perform observable events by the time d units, defined in terms of the

reactive design miracle:

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

which uses event 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 event e_2 ($e_2 \notin \alpha P$) to resolve the first external choice if P does nothing when d is due. This is really a very strong requirement in which there is no alternative but to meet the deadline; that is, within d time units, either P performs observable events or P terminates without doing anything, otherwise the process will never start.

Interrupt

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

$$\begin{aligned} (b \rightarrow c \rightarrow SKIP) \triangle \{a\} P_2 = (b \rightarrow (&\quad c \rightarrow SKIP \\ &\ \Box\ a \rightarrow P_2) \\ \Box\ a \rightarrow & P_2) \end{aligned}$$

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

$$I3(P) = P \lhd wait \rhd I\!I_R$$

which, opposite to $R3$, states that P behaves like itself if its predecessor has not finished yet. This is a very strange healthiness condition, but it works surprisingly effective to bring forward $a \rightarrow P_2$. In their definition of the interrupt operator, they firstly redefine $a \rightarrow P_2$ by using $I3$ to force a to be available in any state of its predecessor, expand the alphabet of P_1, and finally restrict $I3$ to be valid within P_1 and P_2. Our definition, entirely based on the above procedure, is given as follows:

$$\begin{aligned} try(a, P) &\mathrel{\widehat=} (a \notin ref' \wedge I\!I \lhd wait' \rhd tr' = tr \frown \langle (t', a) \rangle)\, ; P \\ force(a, P) &\mathrel{\widehat=} I3 \circ try(a, P) \end{aligned}$$

$$\begin{aligned} a \triangle P &\mathrel{\widehat=} CSP1(ok' \wedge force(a, P)) \\ P_1 \triangle \{a\} P_2 &\mathrel{\widehat=} R(P_1^{+a}\, ; (a \triangle P_2)) \qquad a \notin \alpha P_1 \end{aligned} \tag{2.1}$$

where $I\!I$ is the relational identity that just keeps all variables unchanged. In (2.1), P_1^{+a} expands the alphabet of P_1 by adding a, and the combination of healthiness conditions R restricts the boundary of $I3$.

Recursion

The set of processes in this model is a complete lattice with respect to the implication ordering where the top element is the reactive design 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 $X = F(X)$.

2.4 Applications

The reactive design miracle is rarely explored in the theory of reactive processes and the theory of CSP. The involvement of miracles with the combinators of CSP gives rise to some very strange processes, each of which violates axioms of the standard CSP failures-divergences model. Woodcock has discussed and proved a few such strange processes in [41]. For example, we combine the miracle with a simple prefix, and then get the following miraculous process[4]:

$$a \to \top_R = R(true \vdash tr' = tr \wedge a \notin ref' \wedge wait' \wedge v' = v) \tag{2.2}$$
$$\mathcal{T}(a \to \top_R) = \{\langle\rangle\}$$

which states that it is waiting for interaction with its environment, but never actually performing it even if the event a has been offered. In addition, $\mathcal{T}$ represents its traces. This process violates an axiom of the CSP failures-divergences model [32],

$$F3.\ (s, X) \in F \wedge \exists\, a \in Y \bullet s \frown \langle a\rangle \notin traces_{\perp}(P) \Rightarrow (s, X \cup Y) \in F$$

saying if at a state an event is not in the refusal set then the process is willing to execute the event.

Another strange process [5] is that the external choice of the miracle with a simple prefix:

$$\begin{aligned} &(a \to SKIP) \,\Box\, \top_R \\ = \ & R(true \vdash \neg\, wait' \wedge tr' = tr \frown \langle(t', a)\rangle \wedge v' = v) \end{aligned} \tag{2.3}$$

$$\mathcal{T}(a \to SKIP \,\Box\, \top_R) = \{\langle(t', a)\rangle \mid t' \in \mathbb{R}^+\} \tag{2.4}$$

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; in other words, no empty trace exists for such a process. Obviously, it violates another important axiom of the standard failures-divergences model of CSP where traces are prefix closed. For example, there is no empty trace in (2.4). In our timed model, this process reveals more interesting features. Because of no constraint on timing in (2.3), the event a will occur when the environment is willing to interact with it. However, there is still no state between the start of the process and the occurrence of a, or the time before the occurrence of a has become invisible. As traces in (2.4), besides the absence of the empty trace, the value of t' completely depends on the environment.

[4]Woodcock has proved this strange process in an untimed model [41]. The proof of the timed process is quite similar and has been done as well.

[5]The proof of this process is given in Appendix.

2.4.1 Urgent and uninterrupted events

As shown in 2.3, the miracle in an external choice can impose the other participant to become *urgent*. For example, a is an urgent event that either occurs instantly or evolves invisibly. One possible application of this strange process is that we can construct *uninterrupted* events or an *uninterrupted* trace in which either all of the elements can happen or none of them can happen individually, but the time when they occur is still controlled by their environment. For example, the traces of the following process are produced by applying (2.3) twice:

$$\begin{aligned} &\mathcal{T}((a \to SKIP \,\Box\, \top_R)\,;\,(b \to SKIP \,\Box\, \top_R)) \\ = &\{\langle (i, a), (i + j, b)\rangle \mid i, j \in \mathbb{R}^+\} \end{aligned} \tag{2.5}$$

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

Interestingly, this application has been partially achieved in some specification languages. For example, RAISE Specification Language (RSL) [13, 45] has an interlock operator which can prevent the interlocked processes from communicating with other processes until one of them terminates. Of course, the communication can take place between the locked processes if they are able to. Promela/SPIN [19, 20] can define atomic sequences which encapsulate a fragment of code to be executed uninterruptedly and individually. In the interleaving of process executions, no other process can execute statements from the moment that the first statement of an atomic sequence is executed until the last one has completed. Unfortunately, to our best knowledge, neither of the two operators has denotational semantics probably because of the insufficient capability of current languages to express the property that something *must* occur. Therefore, our time model is very likely to give denotational semantics to the two specification languages, so that the soundness of the languages can be proved and specification can be verified in theorem provers.

2.4.2 Instant events and punctual clocks

The deadline operator in our timed model is different from the deadline operator used in most other models. The '*hard*' deadline in real-time programming [10] plays a role of a compiler directive, which acts like an assertion to statically check whether a program will meet its timing requirements, and returns errors if these requirements are not satisfied. In timed CSP, the deadline operator is usually constructed by the timeout operator and the process $STOP$, i.e., the process will be deadlocked if the deadline is breached. By comparison, our 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 go through only feasible paths to meet the deadline. If the deadline cannot be satisfied anyway, the whole process will not start at all.

In combination with the deadline operator, we are able to specify some interesting processes. For instance, the following process represents that if the

event a occurs, the event b must occur by d time units but after a:

$$\mathcal{T}(a \to ((b \to \mathit{SKIP}) \blacktriangleright d)) = \{\langle (i, a), (i + j, b) \rangle \mid i, j \in \mathbb{R}^+ \land i \leq d\}$$

The traces of this process are quite similar to the ones in (2.5) and the difference is that the occurrence of b is restricted by d. We use the following process to illustrate that a will not happen if b is blocked:

$$(a \to ((b \to \mathit{SKIP}) \blacktriangleright 0)) \underset{\{b\}}{\parallel} \mathit{STOP} = a \to \top_R$$

The result is the same as (2.2), which means that a will never happen even if it is not in the refusal set.

Setting the value of the deadline as zero can make a process or an event become *instant*. The substance of the instantaneity is to use the inexecutable reactive design miracle to force a process to instantly occur. For the sake of convenience, we use the following abbreviations as a shorthand to represent instant events or processes:

$$\begin{aligned} &\ddagger P = P \blacktriangleright 0 \\ &P_1 \ddagger P_2 \mathrel{\widehat=} P_1 \,; (P_2 \blacktriangleright 0) \\ &a \ddagger b \mathrel{\widehat=} (a \to \mathit{SKIP}) \ddagger (b \to \mathit{SKIP}) \end{aligned}$$

Here the instantaneity operator squeezes the 'distance' of events and processes to zero. In fact, instant events are an extreme instance of urgent events.

The instantaneity operator is unique from other specification languages. In timed CSP two or more events may occur without any delay, but it completely depends on its environment. For instance, a process, $a \to b \to \mathit{SKIP}$, may have a trace like $\langle \{i, a\}, \{i, b\} \rangle$ if its environment is friendly. However, if its environment is unpredictable, there is no guarantee that the above process behaves in that way. Lawrence proposes CSPP [24] which is an extension of CSP and HCSP [23] to capture the semantics of hardware compilation. One of the interesting features of CSPP is that it allows true concurrency in which multiple events can occur instantly but without any order. Similarly, this approach cannot guarantee the instantaneity of events because the rest of the events can still happen if some of them have been blocked. The true meaning of instantaneity of events in our timed model is that not only are we unable to identify events by timing, but also they are so tightly attached that none of them can happen individually; in other words, if any of these events is blocked, none of them will happen.

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

$$\begin{aligned} C &= \mathit{WAIT}\ 1 \,; ((a \to \mathit{SKIP}) \triangleright \{0\} \mathit{STOP}) \\ &= \mathit{WAIT}\ 1 \,; ((a \to \mathit{SKIP}) \,\Box\, (\mathit{WAIT}\ 0; e \to \mathit{STOP})) \setminus \{e\} \\ &= \mathit{WAIT}\ 1 \,; (a \to \mathit{SKIP} \,\Box\, \tau \to \mathit{STOP}) \end{aligned}$$

where, obviously, a may not occur because of the nondeterminism. However, the situation can entirely change if we use the deadline operator:

$$\begin{aligned} C' &= WAIT\ 1\ ; ((a \to SKIP) \blacktriangleright 0) \\ &= WAIT\ 1\ ; (((a \to SKIP) \Box (WAIT\ 0; e \to STOP)) \setminus \{e\}) \\ &\qquad\qquad\qquad\qquad \Box\ WAIT\ 0\ ; \top_R) \\ &= WAIT\ 1\ ; (((a \to SKIP \Box e \to STOP) \setminus \{e\}) \Box \top_R) \end{aligned}$$

where a must occur instantly after one time unit otherwise the process will behave like the miracle. If a cannot occur right now, C'' behaves like as follows:

$$\begin{aligned} C' &= WAIT\ 1\,; (STOP \Box \top_R) \\ &= WAIT\ 1\,; \top_R \end{aligned}$$

The punctual clocks are the key factor to the unification of the continuous change of states and the discrete behaviour of events, which especially contributes to the semantics of hybrid systems.

2.4.3 Shared variables

The combination of instantaneity and parallel composition gives many strange but extremely useful processes. For example, we may use instant events to simulate a locking protocol for accessing a shared variable. Usually, a user can lock the shared variable until it has finished related operations. However, in a concurrent system, many users access the shared variable only for reading its value, rather than executing updating operations. The goal of designing such a protocol is to allow those users who want to update the value to lock the variable, and meanwhile those users who go only for reading the value to freely access the variable. To describe the protocol in timed CSP is awkward and error-prone. By contrast, using instant events may give a easier solution to this tricky problem. For instance, we may define the following process which holds a shared variable and updates it by communication:

$$SV(v) = read?id!v \to SV(v) \Box read?id!v \ddagger update!id?x \to SV(x)$$

which uses the channel *read* to give a user the value of v and the channel *update* to change the shared variable. In practice, users only for reading the value will not choose the right path of the external choice, since they do not have updating actions. Although the initial events of both sides of the external choice are same, they will not result in nondeterminism.

To prove the correctness and consistency of the model, we have done a shallow embedding [39] of the semantics of our timed *Circus* in the theorem prover PVS. One of our aims to develop this model is trying to find an approach to formalise the timebands model [3] in which a system is decomposed to reveal different behaviours in different time bands (granularities). For example, an event in a higher (coarser) band can be mapped into an activity in a lower (finer) bands. To achieve the goal, it seems useful to use urgent events to maintain the consistency and coordination of the two bands. In addition, this model also provides an excellent platform for exploring a hybrid system that models continuous changes of the physical environment and communications of discrete events.

3 Semantics of the Timebands Model

In consideration of the nature of the timebands model, we intend to use the timed version of *Circus* to express its semantics. That is, the timebands model is a subset of *Circus* and it is one of the applications of *Circus* in the analysis of various systems. In addition, the newly explored process, *the miracle*, plays a crucial role in the construction of the timebands model to link all time bands as a whole. The entire model is developed in a number of stages in this chapter including time bands, granularity and precision, simultaneous events and durative activities, mappings and abstractions between bands.

3.1 Time bands

A system in the timebands model recognises a finite set of distinct time bands, and it always has the highest and the lowest bands that give a temporal system boundary. Each band is defined by a granularity, representing the basic unit of time in that band. This is different from temporal logic approaches which can represent a possibly infinite set of time bands. For instance, time granularity is usually expressed by means of a layered structure, which contains two kinds of layered structures, *downward unbounded layered structures*, that is, it consists of a coarsest band together with an infinite number of finer bands, and *upward unbounded layered structures*, that is, it consists of a finest band together with an infinite number of coarser bands.

The timebands model adopts continuous time, usually represented by real numbers, to support the change of states, such as physical environmental variables. Therefore, a granule is simply a set of time instants and a granularity is a mapping G from the integers to a granule, so that it satisfies the following healthiness condition:

$$\boldsymbol{G1} : \forall\, i, j : \mathbb{N} \mid i < j \land G(i) \neq \emptyset \land G(j) \neq \emptyset \bullet (\forall\, t : G(i), u : G(j) \bullet t < u)$$

which states that any two granules of a granularity have no overlap and the elements of granules are ordered the same as their index order.

A granule $G(i)$ can be comprised of a single instant, a set of contiguous instants, or even a set of non-contiguous instants. For example, the bank holidays for 2009 in England, defined as a collection of several days from different months, can be used as a granule. There are many relationships defined between granularities and we introduce only two most used ones as follows:

- groups into: $G1 \trianglelefteq G2 \Leftrightarrow \forall\, j : \mathbb{N} \bullet \exists\, S : \mathbb{P}\,\mathbb{N} \bullet G2(j) = \bigcup_{i \in S} G1(i)$
- finer than: $G1 \preceq G2 \Leftrightarrow \forall\, i : \mathbb{N} \bullet \exists\, j : \mathbb{N} \bullet G1(i) \subseteq G2(j)$

The ***groups into*** relationship states that every granule in $G2$ is the union of some sets of granules in $G1$. For example, *days* groups into *business days* that usually denotes Monday to Friday. The ***finer than*** relationship states that every granule in $G1$ is a subset of some granules of $G2$. For example, *business days* is finer than *days*, or *days* is finer than *months*. Of course, any two of granularities are not always comparable, e.g., *weeks* and *months* are incomparable. The switch between different time bands must occur between comparable bands.

3.2 Punctual clocks

In modelling of real-time systems, we have used to employ 'clocks' to aid scheduling and coordination. Based on time granularity, we can define not only some universal measures such as second, minutes, hour, day and so on, but also some specific time scales such as *business day, generation* and *era*. We represent a default clock in a band by defining each granule as a 'clock-tick' event, which is modelled just like any other event. When necessary, more abstract clocks can be defined by the basic unit of time in the band. For instance, the clock called *business days* is placed in the day band, however it is different from the default day clock.

In the implementation of a process algebra language like timed CSP, discrete time units are simulated by clock-tick events so as to translate timed specifications to untimed specifications which can be executed by the model checker FDR. Nevertheless, there is a subtle drawback for the clocks defined in the context of timed CSP. For example, a simple clock in timed CSP may be defined as follows:

$$C = (tick \rightarrow SKIP);\ \ WAIT\ 1\ ;\ C$$

Notice that the event *tick* does not always precisely occur every one unit, because it can be easily interfered or blocked by other participants in synchronisation. Admittedly, it is impossible to specify a process in which an event must occur at certain time points using timed CSP, e.g., clock-tick events must occur on time. We perhaps compromise the requirement by stating that the event should occur at specified time points, or else the process stops. Unfortunately, even such a process cannot be properly defined in time CSP. For example, we may use the timeout operator to define the following process:

$$C1 = (WAIT\ 1\ ;\ (tick \rightarrow SKIP)) \triangleright\{1\}\ STOP$$

where $\triangleright\{1\}$ is the timeout for one time unit. However, the timeout operator cannot guarantee that *tick* occurs exactly at 1:00 (if $C1$ starts from 0:00), since the process $C1$ has a nondeterministic behaviour when the deadline is due; that is, *tick* may occur or $C1$ just behaves like $STOP$ at 1:00.

Apart from precise clock-tick events, a natural clock in a real system should not be interfered whilst other components coordinate with the system by the clock-tick events. Such a clock can be modelled in terms of the strong deadline operator defined in Section 2.3.

$$C2 = ((tick \rightarrow SKIP) \blacktriangleright 0);\ \ WAIT\ 1\ ;\ C2$$

which states that the event *tick* must precisely occur at an interval of one time unit, otherwise it behaves like the miracle. This clock is an ideal clock which can satisfy all above requirements, but its definition is too strong to be useful. For example, the process $C2$ has one and only one trace like $\langle tick, tick, ... \rangle$, and any process can not synchronise on *tick* with $C2$ unless it has enough clock-tick events. Finally, we define a feasibly punctual clock with a bounded number of

clock-tick events as follows:

$$C3(n) = \Box_{i \in \{0..n\}} C3(i)'$$

$$C3'(i) = \textbf{if } i == 0 \textbf{ then } \mathit{SKIP}$$

$$\textbf{else } (\mathit{tick} \rightarrow \mathit{SKIP}) \blacktriangleright 0 \,;\, \mathit{WAIT}\ 1 \,;\, C3'(i-1)$$

whose traces are given by

$$\mathcal{T}(C3(n)) = \{\langle \mathit{tick} \rangle^i \mid i \in \{0..n\}\}$$

where $\langle \mathit{tick} \rangle^i$ is the concatenation of finite traces, for example, $\langle \mathit{tick} \rangle^0 = \langle\rangle$ and $\langle \mathit{tick} \rangle^2 = \langle \mathit{tick}, \mathit{tick} \rangle$.

Because of at least one default clock within a band, a comprehensive clock consisting of all sub-clocks from each band can be built as well. For instance, in a system with two time bands, a day band and an hour band, the punctual clock may be defined as follows:

$$C4(m, n) = CD(m) \,|||\, CH(n)$$

where $CD(m)$ is the clock of the day band and $CH(n)$ is the one of the hour band, and both of them are defined in the same way as $C3$ but they have different clock-tick events to denote different time units. Here $C4$ is a miraculous process in which clock-tick events from different bands are automatically matched to interact with other components. Note that this clock is simply a demonstration of how a punctual clock looks like. In practice, we have a number of mechanisms to generate punctual clocks with regard to the time bands and their relationships.

3.3 Events and precision

Events are instantaneous, but they must be defined within a certain band. For example, compared to the basic time unit, an event defined in the day band does not take any time to execute, however it might take several hours in a finer time band. Indeed, there are a few relationships between events within a band. Firstly, urgent events have been defined in Section 2.4 such as $a \ddagger b$. Urgency is the strongest constraint which squeezes any interval between two events into zero. This is a very useful relation and is used especially to link different time bands via events and activities.

In specification of a system, an event may cause a immediate response. For example, we may consider such a requirement like 'when the fridge door opens the light must come on immediately', which means that the response is within the precision of the band. Precision, representing the measure of the accuracy of events within that band, can only be expressed using the granularity of finer bands. Accordingly, two *simultaneous* events must, when viewed from a finer band, be within the precision of the current band. In respect to the finite number of time bands in the model, the finest (lowest) band has no precision, and the behaviours should be described precisely in the band. Due to precision, two simultaneous events cannot be exactly distinguished because the 'gap' between them is too small to be considered. Here the small 'gap' also results in the

tolerance of the bahaviours when mapping the two events to corresponding activities of a finer band.

Similar to the definition of urgent events, the simultaneous operator is defined as follows:

$$P_1 \overrightarrow{\#} P_2 \mathrel{\widehat=} P_1 \,;\, (P_2 \blacktriangleright \rho)$$

where ρ is the precision of that band. Two simultaneous events, e.g. events a and b, are expressed as either a is before b or b is before a, but they must occur within the precision. We also use the following abbreviations to represent simultaneous events:

$$a \overrightarrow{\#} b = (a \rightarrow \mathit{SKIP}) \overrightarrow{\#} (b \rightarrow \mathit{SKIP})$$
$$a \# b = a \overrightarrow{\#} b \;\square\; b \overrightarrow{\#} a$$

where $\#$ denotes that a and b are simultaneous, and $\overrightarrow{\#}$ that they are simultaneous but with the order. Note that $\#$ and $\overrightarrow{\#}$ are used to connect processes, e.g., a appeared on the left side is actually the abbreviation of $a \rightarrow \mathit{SKIP}$. And also this abbreviation is applied to all simultaneous events.

Simultaneity is also a very strong property which is similar to urgent events, that is, either two events occur together or neither of them occurs individually. The difference is that two simultaneous events allow one of them to occur within the precision after the other has occurred, even though such a short delay is too small to be considered. There is a subtle differences between $\overrightarrow{\#}$ and the sequential composition operator. For example, the process, $(a \rightarrow \mathit{SKIP}) \,;\, (b \rightarrow \mathit{SKIP})$ may behave as if b occurs without any delay after a provided the environment is friendly, because of one of assumptions of CSP and *Circus*, maximal progress, which means that an event must occur at the instant that all participants are ready. Of course, b may also happen far late after a occurs. In other words, the occurrence of b is completely determined by its environment.

The difference of two simultaneous events and two free sequential events is illustrated as well by their traces:

$$\mathcal{T}((a \rightarrow \mathit{SKIP}) \,;\, (b \rightarrow \mathit{SKIP})) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$$
$$\mathcal{T}(a \overrightarrow{\#} b) = \{\langle\rangle, \langle a, b\rangle\}$$

where $a \overrightarrow{\#} b$ does not have the trace $\langle a\rangle$ as the first process does. So far, we may have at least three approaches to express the sequential relations of events, however all of them have different underlying meanings. For example, expressing a commitment that the event a occurs at 1:00, $\mathit{tick} \ddagger a$ denotes that a *must* precisely happen at 1:00 even if we observe it in a finer band; $\mathit{tick} \# a$ means that a *must* occur at 1:00 too, but a is allowed to happen early or late within the bound of the precision; $\mathit{tick} \rightarrow a \rightarrow \mathit{SKIP}$ or $(\mathit{tick} \rightarrow \mathit{SKIP}) \,;\, (a \rightarrow \mathit{SKIP})$ states that a occurs only if its environment provides the offer, and a occurs exactly at 1:00 only if it environment is friendly.

We cannot visually distinguish two simultaneous events; however, the interval between simultaneous events will be revealed in the form of precision when mapping these events to corresponding activities in a finer band. As a result, the precision basically plays two roles in a band: one is to measure the accuracy

of events such as simultaneous events; the other is to restrict the duration of activities. In addition, simultaneous events, consecutively simultaneous events and ordinary sequential events can be clearly differentiated by their traces:

$$\mathcal{T}((a \to b \to c \to SKIP) \blacktriangleright \rho) = \{\langle (i,a), (i+j,b), (i+j+k,c) \rangle \mid j+k \leq \rho\}$$
$$\mathcal{T}(a \overrightarrow{\#} b \overrightarrow{\#} c) = \{\langle\rangle\} \cup \{\langle (i,a), (i+j,b), (i+j+k,c) \mid j,k \leq \rho \rangle\}$$
$$\mathcal{T}(a \to b \to c \to SKIP) = \{\langle\rangle, \langle (i,a)\rangle, \langle (i,a), (i+j,b)\rangle, \\ \langle (i,a), (i+j,b), (i+j+k,c)\rangle\}$$

Unfortunately, simultaneity is not transitive, i.e., the fact that a and b are simultaneous, so are b and c, does not imply that a and c are simultaneous. This also elegantly explains that a sequence of consecutively simultaneous pairs, or repeatedly fast-moving events can be observed durative behaviours. We might not recognise any pair of them because the interval between them is less than the precision. Nevertheless, the whole duration may take a long time. For instance, the following process demonstrates the failure of transitivity:

$$a \overrightarrow{\#} b \overrightarrow{\#} c \neq (a \overrightarrow{\#} b \overrightarrow{\#} c \underset{\{a,c\}}{\parallel} a \overrightarrow{\#} c)$$

where the left side of the expression denotes that c is simultaneous with b which is in turn simultaneous with a; the right side further imposes a constraint that a and c are simultaneous too. To better differentiate the two processes, these events have to be mapped into activities of a finer band to exhibit more detailed information. In addition, clock-tick events cannot be simultaneous on account of the punctual clocks.

3.4 Timeless traces

In timed CSP and *Circus*, a timed trace is used to denote a record of performed events, any of which is a pair, consisting of a time and an event. However, the refinement of a timed model is much harder than the one of an untimed model. Therefore, *timewise refinements* are considered by relating untimed process descriptions to timed ones. The basic idea is that some untimed properties such as safety and liveness of a timed system may be verifiable in the untimed models, and the more complex timed model needs to be used for analysis only where timed behaviour is critical.

In the timebands model, we explicitly use clock-tick events to represent discrete time and directly express continuous time by the operator $WAIT$. Within a band, clock-tick events are guaranteed to arrive on time, and hence it is possible to know when events occur by finding the nearest clock-tick events, instead of recording the time of events individually. This point provides a strong support to the timewise refinement of the timebands model, or the '*timeless*' timebands model.

Also, we are not interested in any specific values of intervals less than a basic time unit between events in that time band. On the contrary, for events whose gaps are in such a scale, we prefer to know their precedence relations and rough time constraints. For example, an explicit deadline between two events, a and b, say 9.3 minutes, is too concrete in the hour band. A more reasonable description is that a is before b (the precedence relation) but they occur at the

same hour. The detailed deadline will become useful when the two events are mapped into activities in a finer band. Note that there is a difference between two events being simultaneous and being 'at the same time'. The former is a much stronger statement which must be viewed from a finer band by means of the precision of the band, but 'at the same time' only requires the two events to occur within the granularity of the band.

Thus, when necessary, an event can be located in absolute time by stating a precedence relationship between the event and one or more clock ticks. For instance, an event a must occur between 1.00 and 2.00, and then a process expressing this can be defined as follows:

$$\begin{aligned} P_1 &= tick \rightarrow a \rightarrow tick \rightarrow SKIP \\ \mathcal{T}(P_1) &= \{\langle\rangle, \langle tick\rangle, \langle tick, a\rangle, \langle tick, a, tick\rangle\} \end{aligned}$$

where we assume that the first *tick* denotes 1.00. This is different from saying the event must occur 'at 1:00', which may be expressed as follows:

$$\begin{aligned} P_2 &= tick \# a \\ \mathcal{T}(P_2) &= \{\langle\rangle, \langle tick, a\rangle, \langle a, tick\rangle\} \end{aligned}$$

If the precision is 5 minutes in this band, in fact it means that a is able to occur 5 minutes early or late.

Similarly, we may read more information from traces of a process. For example, we have the following traces:

$$\mathcal{T} = \{\langle\rangle, \langle tick\rangle, \langle tick, a, b\rangle, \langle tick, a, b, c\rangle\}$$

which states that a and b are bundled to occur together. Here we cannot distinguish that they are simultaneous, urgent or just uninterrupted events because the interval between them is not recorded. However, the interval will be revealed if a and b are mapped into activities in a finer band. The event c happens after b but before the next clock-tick event. Thus, the above traces may correspond to the following process:

$$P_3 = (tick \rightarrow SKIP)\,;\, a \overrightarrow{\#} b\,;\, (c \rightarrow SKIP)$$

To adopt untimed events in the timebands model, we modify the definition of the simple prefix listed in Section 2.3 by deleting the time of events in the procedure of concatenation.

$$a \rightarrow SKIP \mathrel{\widehat{=}} R \left(true \vdash \begin{array}{c} tr' = tr \wedge a \notin ref' \\ \lhd\ wait' \rhd \\ tr' = tr ^\frown \langle a\rangle \end{array} \wedge v' = v \right)$$

3.5 Explicit clock-tick events

Since the introduction of clock-tick events, we have to change the way in which we have used to define a process. For example, we can read different stories from a very simple process, $Q = tick \rightarrow a \rightarrow SKIP$, if it is placed in different occasions. First, without the synchronisation with punctual clocks, Q states that a occurs whenever its environment is ready. In accordance with the definition of

the prefix, events always wait for the interaction with its environment while the time is elapsing. The processes defined in the timebands model must synchronise with punctual clocks of the system, and hence Q actually means that a should occur before the next coming *tick*, otherwise it becomes deadlocked. We may modify Q, described as $Q1$, to make a arbitrarily occur to interact with other components.

$$Q1 = tick \rightarrow Q1'$$
$$Q1' = a \rightarrow SKIP \;\Box\; tick \rightarrow Q1'$$

Second, the event *tick* does not always occur immediately when the process starts. For instance, in a process, $tick \rightarrow b \rightarrow Q$, the event *tick* should occur at once, whereas the *tick* in Q still keeps idle when Q has started and is waiting for the arrival of the second *tick* of a punctual clock.

We also become aware that the operator *WAIT* defined in CSP and *Circus* is inadequate to this timebands model, because it just allows time to elapse and cannot interact with punctual clocks. For example, the process, $WAIT\ 2\ ; (a \rightarrow SKIP)$, cannot be synchronised on clock-tick events with punctual clocks. Therefore, we define a new operator *WAITC* which simply does the same as *WAIT* except it explicitly provides corresponding clock-tick events:

$$WAITC\ d\ = ((WAIT\ d) ; (e \rightarrow SKIP) \underset{\{e\}}{\parallel} C \triangle \{e\} SKIP) \setminus \{e\}$$

where C is a default clock, defined in Section 3.2, to provide clock-tick events of a local band, and the event e is used to terminate C when the deadline has passed. Here it is unnecessary to make those clock-tick events punctual in C because they are bound to be synchronised with the punctual system clock. This operator automatically generates the right number of clock-tick events in considering the context. For example, $WAITC\ 0.5$ provides one clock-tick event if its predecessor finishes in excess of a half time unit, or none if otherwise. Note that the newly defined *WAITC* cannot fully replace the original *WAIT* in many circumstances. Similarly, the operator *STOP* is redefined as $STOPC \mathrel{\widehat=} STOP \mathbin{|||} C$.

The parallel composition is a bit complex in that we have to provide enough clock-tick events to those participants which have terminated earlier. Hence, we define a process to be appended to all participants in parallel.

$$C_0 = e \rightarrow SKIP \;\Box\; tick \rightarrow C_0$$
$$P \underset{A}{\parallel}{}_c\, Q = (P ; C_0 \underset{\{e, tick\} \cup A}{\parallel} Q ; C_0) \setminus \{e\}$$

where *tick* represents the local clock-tick event and e is not included in the alphabets of P and Q. The urgency of the hidden event makes the parallel composition terminate once both P and Q have terminated.

In fact, the explicit clock-tick events can compromise the timeout operator. For example, in a process, $P \rhd Q$, we may define P as follows:

$$P = a \rightarrow SKIP \;\Box\; tick \rightarrow P$$

which means that a occurs whenever its environment is willing to interact with it, and meanwhile the clock-tick events can be observed. Unfortunately, in our

timed model those clock-tick events can trigger the timeout operator to pass the control of the program over to Q in advance.

In order to make up the deficiency of the standard *timeout* operator, we define an explicit *timeout* to clarify that the process will behave like Q if a certain event in P can not happen within d.

$$P \rhd \{a, d\} Q = (T \underset{\{a, e_1\}}{\parallel} (T_0 \triangle \{e_1\} SKIP)) \setminus \{e_1, e_2\} \tag{3.6}$$

$$T = (P' \triangle \{e_2\} SKIP) \underset{\{a, e_1, e_2\}}{\parallel} (WAIT\ d;\ e_2 \rightarrow Q' \Box (T_0 \triangle \{e_1\} SKIP))$$

$$P' = P \,;\, e_1 \rightarrow SKIP$$

$$Q' = Q \,;\, e_1 \rightarrow SKIP$$

$$T_0 = a \rightarrow T_0$$

This is a pretty delicate definition. T_0 provides enough offers of a in case the process repeatedly executes a no matter the timeout operator is resolved by P or Q. The event e_1, denoting termination of P or Q, interrupts T_0 once either P or Q terminates and then makes the whole process terminate. The event e_2 is used to interrupt P if a does not occur within d. As a result, in a band these modified operators are used to synchronise the clock-tick events and thereby avoid deadlock.

3.6 Activities

An activity consists of a nonempty set of events, all of which are in the same band. Activities are detailed explanations of events of higher bands, and then they are given only in suitable time bands in which clock-events are employed to denote the beginning and the end. Hence, every activity starts and also finishes with clock-tick events. For instance, an activity may be defined as follows:

$$A = tick \# a_1 \,;\, tick \# a_2 \,;\, tick \# a_3$$

which means that the events such as a_1, a_2 and a_3 are simultaneous with clock-tick events; in other words, each event occurs 'at basic time units' and the duration of the activity is two time units. Note that a_1 may actually occur before the event *tick* in $tick \# a_1$, but we consider the activity still starts with the clock-tick event roughly since *tick* and a_1 cannot be visually distinguished in this time band.

It is not necessary that each event must be bundled with clock-tick events. For instance, the activity, $tick \rightarrow a \rightarrow tick \rightarrow SKIP$, represents that a must occur anywhere within a basic time unit. In timed CSP and *Circus*, time is represented by implication. In the timebands model, we explicitly express clock-tick events as the reference to synchronisation. Therefore, the clock-tick events do contribute to observation of the activity; that is, the duration is one time unit rather than the time of executing the event a. The duration of an activity is determined by how many clock-tick events it involves.

Activities might be without duration. For example, the activity, $tick \# a$, has no duration where the event *tick* plays the dual roles of the starting event and the ending event. Here the duration of the simple activity is too small to be considered in this band. Obviously, the simplest activity is a single clock-tick event.

Each activity must have at least one signature event, which is not only the major observation of the activity, but also the linking to the corresponding event in a higher band. Activities may have more than one signature events. For example, making a drink by a vending machine may have two choices, tea or coffee, which may be described as follows:

$$\begin{aligned} \mathit{Drink} = \quad & (\mathit{tick}\#\mathit{hotwater}\,;\,\mathit{tick}\#\mathit{milk}\,;\,\mathit{tick} \rightarrow \mathit{tick}\#\overline{\mathit{tea}}) \\ & \Box\,(\mathit{tick}\#\mathit{hotwater}\,;\,\mathit{tick}\#\mathit{milk}\,;\,\mathit{tick}\#\overline{\mathit{coffee}}) \end{aligned}$$

where $\overline{\mathit{tea}}$ and $\overline{\mathit{coffee}}$ are the signature events to coordinate the event in a higher band when mapping, and we use overhead lines to make them different from other ordinary events. However, these signature events must be linked with a same event of a coarser band.

So far, all above activities that we have defined have the fixed-length duration. Additionally, we may define an activity with a flexible duration as follows:

$$\begin{aligned} A1 &= \mathit{tick}\#a_1\,;\,A1' \\ A1' &= \mathit{tick}\#\overline{a_2}\,\Box\,\mathit{tick} \rightarrow A1' \end{aligned}$$

where a_2 waits at least one time unit and the activity allows more time to pass over. Immediately, we notice that the duration of the activity should be less than or equal to the precision, otherwise it cannot be considered an event of a higher band. This imperative requirement will be fulfilled when mapping or abstracting, since the precision for an activity is not yet decided until the link with an event in a higher band has been established. For example, there are two activities, A and B, in the minute band, but A and B are linked to two events in the day band and in the hour band respectively; consequently, their precisions might be different.

In summary, activities are special processes which use clock-tick events as starting and finishing events, contain at least one signature event and impose a time constraint on their duration.

3.7 Mappings and abstractions between bands

In the components of the model so far considered, all behaviours have been confined to a single band. The essence of the timebands model is to describe the behaviour of each component of a system in a best suitable time band, and compose the multiple-band behaviours regarding the properties to be verified. To achieve this goal, events in one band need to be mapped into activities in finer bands, or activities are abstracted to events in coarser bands.

Activities become useful only when they are linked with events in higher time bands. As illustrated in Figure 2, processes are defined in their own time bands and the ones defined in different time bands have no intersection except for the linking of events and activities. Furthermore, these links are the one and only channel to integrate all behaviours of the timebands system. The establishment of the links is achieved by means of imposing the events and the signature events of the activities to be urgent events, so that they are constrained to occur together at all time.

Mappings and abstractions are different procedures, which are similar to the two different approaches, *top-down* and *bottom-up*, to construct a complex

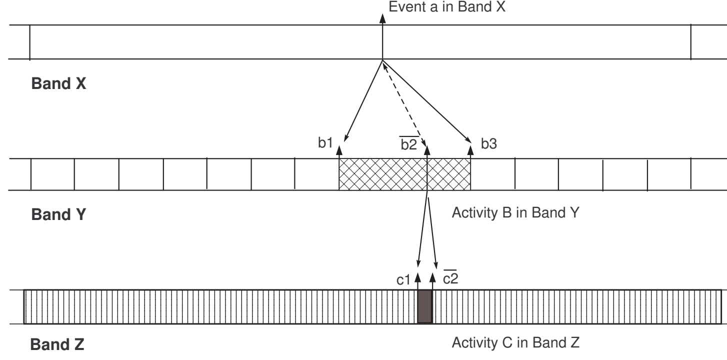

Figure 2: Mappings and Abstractions Example

system. When mapping an event into an activity in a finer band, the activity should always terminate within the precision of the band where the event is placed, and moreover, the activity must happen as a whole. That is, the activity either timely terminates or never starts. The constraints on activities are able to preserve consistency of time bands. If an event may freely occur in a band, the behaviour of its corresponding activity, viewed from the higher band, should be fully revealed in a lower band. For example, in a lecture example, a lecture is allocated in a day band, which basically depends on the time slots of lecturers and students; however, in an hour band, the successful allocation also relies on the availability of resources such as projectors, labs and so on. For this reason, mappings are aimed to avoid the inconsistent behaviour that the lecture occurs in the day band, but it does not finish because of the shortage of the resources. The constraint, which is imposed to any activity in the mappings, is a function defined as follows:

$$TER(A, \rho) = ((A\,;(e \to SKIP)) \triangleright \{e, \rho + 2 * \rho'\}\top_R) \setminus \{e\} \qquad e \notin \alpha A$$

where ρ is the precision of the band where the corresponding event is placed, and ρ' is the precision of the current band. We restrict the duration of an activity is no longer than ρ, however, in fact, we do allow non-clock-tick events to occur early or late within the bound of ρ'. Since an activity always starts and finishes with clock-tick events, considered the precision of the band where the activity is defined, the maximal duration of the activity should be $\rho + 2 * \rho'$. Of course, in most cases the value of ρ' is zero, unless one or more events involved in this activity are mapped into activities in lower bands again.

In the definition of *TER*, we use the newly-defined timeout operator to detect the added event e. If A terminates before the deadline expires, the sequent occurrence of e discharges the operator to throw away $\top_R$, otherwise A will not start in order to prevent the process from behaving like the miracle. Note that e is different from the event 'e' used in the explicit timeout operator, because 'e' is not included in the alphabet of $A\,;(e \to SKIP)$ in accordance with the definition of this operator.

Different from mappings, finding right positions in lower bands for those events in higher bands, abstractions are searching matched activities for higher

bands' events. The operation of an abstraction simply trims the activity so as to make its duration no longer than the precision for the purpose of consistency. If the signatures events of the pruned activity are able to happen, the linked event in a higher band can occur as well. If none of the signature events occurs during execution of the activity, we cannot observe the corresponding event. This procedure does allow the behaviour of an activity to partially happen.

For example, we are going to record the attendance of a lecture. If the roll call is implemented in the beginning of the lecture, a student still has a successful attendance even if he/she leaves the classroom later. If no roll call holds during the lecture, no successful attendance can be observed, e.g., in the day band. We consequently conclude that it is inappropriate for linking the event, e.g. successful attendance, in a higher band with the activity, e.g. the whole lecture, in a lower band. Although we do not request activities in abstractions must terminate, we restrict the duration of the activities should be within the precision. Similarly, the constraint on activities during abstractions is defined as the following function:

$$ABS(A, \rho) = ((A \,;\, (e \rightarrow SKIP)) \triangleright \{e, \rho + 2 * \rho'\} SKIP) \setminus \{e\} \qquad e \notin \alpha A$$

which states if e cannot occur by the deadline, A will be compulsively interrupted by the timeout operator and then terminate.

3.8 Integrating time bands

The basic framework of the timebands model is to express dynamic behaviours of a system in best suitable time bands, and then, in the light of the requirements of verifying properties, map these behaviours into lower bands or abstract them upto higher bands. To maintain consistency and coordination between different time bands, the events of higher bands are linked with the activities of lower bands by forcing these events and signature events of the activities become urgent events. The linking is the sole channel through which different time bands can communicate, and establishing the links is the only way to integrate all the behaviours of the system into a whole. The procedure of linking is also defined as a function:

$$L(a, *A) = \Box_{s \in *A}\, a \ddagger s$$

where a is the event in a higher band and $*A$ is the collection of all signature events of activity A. The event is bundled with the signature events so tight that either both of them instantly occur together or neither of them happens. This function is applied to both mappings and abstractions.

Finally, we use a simple example, illustrated in Figure 2, to demonstrate the integration of time bands. Suppose that three time bands, X, Y and Z, are given in an increasingly finer order, and then granularity, precision and the clock-tick event of each band are defined as follows:

$$\begin{array}{ll} Granularity(X, Y) = \{15\} & Granularity(Y, Z) = \{10\} \\ Precision(X, Y) = 2 & Precision(Y, Z) = 2 \\ Event\ xtick : X, ytick : Y, ztick : Z & \end{array}$$

We consider that activity B in Band Y is the major observation, and it will be abstracted upto the Band X and mapped into Band Z for verifying properties which cannot be properly
described in Band Y.

$$Process\ P : X = xtick \rightarrow a \rightarrow xtick \rightarrow SKIP$$
$$Activity\ B : Y = ytick\#b_1\ ;\ (ytick \rightarrow ytick\#\overline{b_2})\ ;\ ytick\#b_3$$
$$Activity\ C : Z = ztick\#c_1\ ;\ ztick\#\overline{c_2}$$

Before events and activities are linked together, processes defined in different time bands have no any interaction, which may be expressed as follows:

$$S1 = P \mid\mid\mid B \mid\mid\mid C$$

Furthermore, activity B is abstracted into event a in Band X and event b_2 of B is mapped into activity C in Band Z. The system with the linking is given as follows:

$$S2 = ((P \mid\mid\mid ABS(B,2) \mid\mid\mid TER(C,2)) \underset{\{a\}\cup *B}{\parallel} L(a,*B)) \underset{\{b_2\}\cup *C}{\parallel} L(b_2,*C))$$

At last, the integrated system is achieved by synchronising with the punctual clock:

$$S = S2 \underset{\{xtick,ytick,ztick\}}{\parallel} PC$$

where PC denotes the punctual clock of the system, which includes three sub-clocks with regard to the granularity. Note that in the abstraction of a and B, event b_3 cannot occur in fact due to the precision. It states that we might not have properly defined B, so we should go back to redefine it or consider to change the related precision.

From the definition of P, a may occur any time between two clock-tick events in Band X. After the abstraction between a and B, event a looks like simultaneous with the first tick event, because B starts from the beginning in Band Y. Also, B does not execute in full and a_3 is just abandoned on account of the precision. In addition, activity C does not occur in the beginning of Band Z and it is bounded to occur around the time point when $\overline{b_2}$ happens; in other words, after the mapping, $\overline{b_2}$ determines when activity C can occur in Band Z.

3.9 Discussion

The construction of complex real-time systems, such as large socio-technical systems, imposes a number of significant challenges, one of which is the natural expression and efficient verification of dynamic behaviours on a wide range of time scales. The traditional approaches usually apply the finest time scale to describe all behaviours of the systems even if some of them only require very 'loose' time constraints. For example, milliseconds are used to represent the change of a reservoir which may take months to be filled in. Also, the fact that the reservoir is filling in a month does not mean it is filling in every

millisecond. However, the timebands model can overcome the weakness of the traditional approaches by supporting the structure properties of systems. That is, it expresses the multiple-scale behaviours of the systems in most suitable time bands and proves the properties only in involved bands.

In addition, we may use a simple example to show how natural and convenient the timebands model is to construct the system which might be awkwardly built by other specification languages. Suppose that we have a meeting at the beginning of each month and the meeting takes two hour each time. To model the statement in one flat time scale, e.g., the basic time unit is one hour, we may have the following CSP specification:

$$M = (a_1 \rightarrow SKIP\,;\; WAIT\ 2\,;\, a_2 \rightarrow SKIP)\,;\; WAIT\ 30 * 24\,;\, M$$

where we assume each month has 30 days on average, and a_1 and a_2 denote the start and the finish of a meeting respectively. In fact, the process M states every 720 hours we have a two-hour meeting. To express extra information, e.g., the meeting can take place anytime during the daytime and start earlier or later than the expected time, the traditional approaches become cumbersome or even incompetent.

The timebands model is able to flexibly and sufficiently express the full requirements of the example. First, we recognise three time bands: a month band, a day band and an hour band. Thus, in the month band we may just have the following process to express the fact that a meeting takes place every month:

$$M_M = mtick\#meeting\,;\, M_M$$

where the simultaneity of *meeting* and *mtick* denotes that the meeting can actually happen at the end of previous month or at the beginning of the current month in consideration of the precision. In the day band, we define an activity as follow:

$$A_D = dtick \rightarrow \overline{a} \rightarrow dtick \rightarrow SKIP$$

which is used to represent that the meeting can take place anytime in a day (i.e., we may define a working day which only includes eight hours.). In the hour band, we project the event a into another activity:

$$A'_H = htick\#a_1\,;\; WAITC\ 2\,;\, htick\#\overline{a_2}$$

where a_1 and a_2 denote the same things as the ones in M. Of course, we may further explore whether the meeting starts and finishes early or late if recognising a finer time band.

4 Case Study

In this section we use two rather complex examples to demonstrate the power of the timebands model, and reveal how natural and efficient to describe multi-scales behaviours of a system.

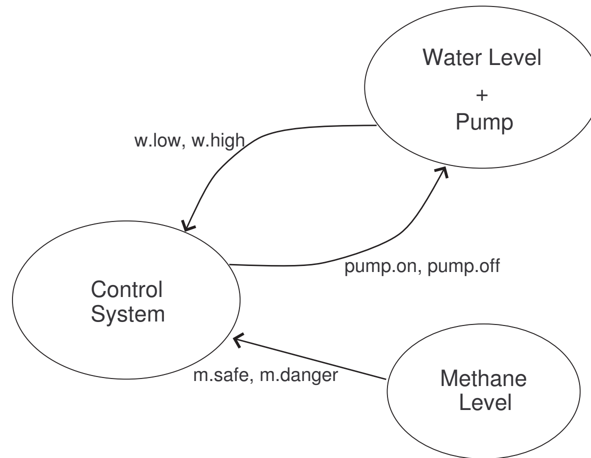

Figure 3: Mine pump example

4.1 Mine pump

The mine pump system is used to control a pump to pump out the water which is collected in a sump. The system has two sensors to indicate the water level. However, for simplicity, we only keep one sensor to denote the high level of the water, because the low level of water does not affect the safety requirement of the system. When the water level is high, the pump is switched on. If the pump is on, the level of the water should drop below the high level during a certain time interval, and then the pump is switched off. The system also has another sensor to monitor the level of methane. Due to the risk of explosion, the pump must not operate when the methane level is above a critical level.

As illustrated in Figure 3, the system is basically decomposed into three major components: a control system for operating the pump according to the readings of those sensors, a subsystem for switching the pump on or off and thereby changing the water level, and a component for passing the level of methane back to the control system. We here remove any time constraint from the control system in order to make it become a purely logic judgement for proper operations, e.g., it can switch the pump on only if the methane level is safe or the system will not switch the pump off if it has been off. The sampling process of the sensors is not modelled in the example because the absence of real data of the mine, and so the worst-case execution time is applied throughout the modelling.

The system is modelled in two time bands, the minute band (M) and the second band (S), in regard to different properties. In the minute band, we recognise the following events to denote the operations of the pump and the states of water and methane:

$$
\begin{array}{lll}
\mathit{Event} & w.low, w.high, & \text{\% the water level} \\
 & m.safe, m.danger, & \text{\% the methane level} \\
 & pump.on, pump.off : M & \text{\% the pump operations}
\end{array}
$$

In addition, we define the event *failure* to express the water level exceeds the critical level, and *mtick* to represent the clock-tick event. Thus, the behaviour of the control system may be described as follows:

$$Bool\ \ ps \qquad \text{\%true if the pump is on}$$

$$\begin{aligned} CS = &\ \ w.low \rightarrow pump.off \rightarrow (ps := false\,;\, CS) \\ &\ \Box\ w.high \rightarrow m.safe \rightarrow pump.on \rightarrow (ps := true\,;\, CS) \\ Control = &\ CS \triangle \{m.danger\}(ps \& pump.off \rightarrow CS \ \Box\ CS) \end{aligned}$$

where *ps* is a local variable to denote the state of the pump, in order to avoid redundant operations to the pump. In the process *CS*, the program is waiting for the interaction of *m.safe* when the water level is high to safely switch the pump on. Finally, the event *m.danger* has the highest priority to interrupt the program to switch the pump off.

We suppose that the change of the water level is slow, [6] and the methane level is stable in most of the time but may incidently change very fast, e.g., it may sharply reach the danger level in a few second. Obviously, such a dramatic change of the methane level cannot be precisely modelled in the minute band. Also, the delay of the operations of the pump is going to be modelled in the second band.

For modelling the change of the water level and methane level, we use worst-case execution time to describe the worst situations. For example, the methane level stays safe at least for a period of time and the pump takes at most certain time units to bring the water level back. To express the change of the methane level in the minute band, it is convenient to define a specification macro as a shorthand to represent a lifetime of a process which must terminate after some time units.

$$\begin{aligned} MT(P, d) = ((P\,;(e_1 \rightarrow SKIP))\triangle\{e_2\}SKIP & \qquad \text{providing } e_1, e_2 \notin \alpha P \\ \underset{\{e_1,e_2\}}{\|}\ {}^{c} & \\ (WAITC\ \ d\ ;(e_2 \rightarrow SKIP))\ \Box\ e_1 \rightarrow SKIP) \setminus \{e_1, e_2\} & \end{aligned}$$

where P is terminated by the interruption if it has not finished yet by d time units, and e_1 is used to make the right part of the parallel composition to terminate if P terminates earlier than the expected d.

The methane component notifies the control system once the state of the methane level has changed, and consequently it may be defined as follows:

$$\begin{aligned} M &= m.safe \rightarrow M \ \Box\ mtick \rightarrow M \\ ML &= MT(M, ml_1)\,;\,(m.danger \rightarrow WAITC\ \ ml_2\ ;\, ML) \end{aligned}$$

where, as illustrated in Figure 4, ml_1 denotes the shortest period when the methane level stays safe, and ml_2 represents the longest period when the methane level is continuously danger. The behaviour between the safe and danger levels is not considered in the minute band because of the usually sharp change,

[6] In fact, it is reasonable to have this assumption because the sharp change of the water level such as flooding will inevitably result in an accident.

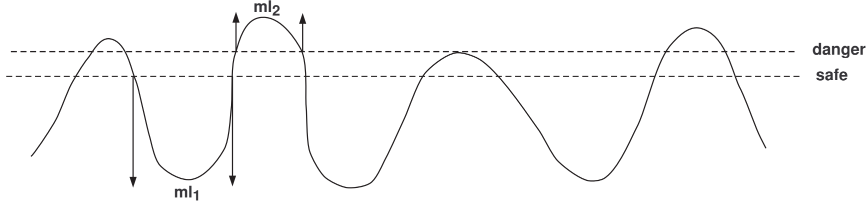

Figure 4: The change of the methane level

whereas it is detailed modelled in the second band. If the methane level is safe, the process M can provide the safe state whenever the control system requests it.

The behaviour of the water component in the minute band is to receive commands from the control system so as to control the pump to change the water level.

$$WL = w.low \rightarrow pump.off \rightarrow w.high \rightarrow (WL_1 \triangleright \{wl_1\} failure \rightarrow STOP)$$
$$WL_1 = pump.on \rightarrow WAITC\ wl_2\ ;\ WL$$

where it starts from the low level and then waits for the command to switch the pump off. We are actually not interested in the behaviour between the low level and the high level, since the worst case is that the water level is high while the methane level is danger. When the water level is high, the *pump.on* in WL_1 should occur within wl_1 time units, otherwise the system fails. If the pump is on, we assume it takes at most wl_2 time units to bring the water level to the low level.

The whole system in the minute band can be comprised of the parallel composition of the three components:

$$SYS_M = Control \underset{A}{\parallel} (ML \mid\mid\mid WL)$$
$$A = \{pump.on, pump.off, w.low, w.high, m.safe, m.danger\}$$

The safety of the whole system completely depends on the control system which sends right commands by means of reading the states of sensors. In the minute band, we concentrate on the correctness of the logic judgement of the control system. For example, we may verify at least two safety requirements:

1. The water level can not exceed the critical level, otherwise the mine must be evacuated.

2. The pump must not operate when the methane level is danger.

The relationship of those local variables is one of the major assumptions to prove the safety properties of the system. For example, wl_1 should be longer than ml_2 ($wl_1 > ml_2$), otherwise the water level will exceed the critical level due to the danger methane level; wl_2 should be shorter than ml_1 ($wl_2 < ml_1$), which enables the water level to return the safe level, or it will breach the relation

$wl_1 > ml_2$. We deal with these relations as invariant, and mingle it with the system specifications in practice.

The timebands model takes a CSP-like refinement theory, which is represented by a notion that which one process is more deterministic than the other. For example, for verifying the first requirement, we simply need to define a process which can execute any event except for the event *failure*. If such a process is refined by SYS_M, it implies that *failure* does not appear in any trace of the system. To prove the pump must not operate during the danger methane level, we need to show that the boolean variable *ps* is false after *methane_danger* occurs. However, we have not stated when *pump_off* would happen after the methane level becomes danger, since any delay of the operations is not considered in the minute band. Therefore, in the second band we may request more restricted time constraints such as *m.danger* and *pump.off* must be simultaneous if the pump is on.

With more details disclosed in the second band, we can tackle different properties, e.g., the pump should be switched off within a certain time interval once the methane level is danger. This property perfectly interprets the significant feature of real-time systems, that is, correctness of the system depends not only on the logical result of the computation but also on the time at which results are produced.

For the reason of simplicity, we still ignore the delay of communication, but consider the delay caused by the pump's operations and the change between the safe and danger levels of the methane. The detailed behaviour of switching the pump on or off can be described as the following activities:

$$\begin{aligned}
&Event\ on, off : S\\
&Activity\ PON : S = WAITC\ \delta\,;\ stick\#\overline{on}\\
&Activity\ POFF : S = WAITC\ \delta\,;\ stick\#\overline{off}
\end{aligned}$$

where δ is the cost of time for the pump's operations. In practice, we might have to modify the specific values of the water level and the methane level to balance the difference.

In addition, we assume the methane level takes at least ml_3 time units to reach the danger level from the safe level, which is expressed as an activity and is linked with the event *m.safe* in the minute band:

$$\begin{aligned}
&Event\ msafe : S\\
&Activity\ MS : S = stick\#\overline{msafe}\,;\ WAITC\ wl_3
\end{aligned}$$

Obviously, mappings are involved only in the integration of two time bands, and then it may be described as follow:

$$\begin{aligned}
SYS_{MS} =&(SYS_M \ ||| \ TER(PON,5) \ ||| \ TER(POFF,5) \ ||| \ TER(MS,5))\\
&\underset{\{pump.on,on\}}{\|} \ L(pump.on, \{on\})\\
&\underset{\{pump.off,off\}}{\|} \ L(pump.off, \{off\}\\
&\underset{\{m.safe,msafe\}}{\|} \ L(m.safe, \{msafe\})
\end{aligned}$$

As a result, both safety requirements in the minute band are unlikely to hold any more since the introduction of the delay in the second band. To maintain the first requirement, we may adjust a margin to allow the system to tolerate the response time. For the second requirement, we may further impose a tighter time constraint in the minute band, e.g., $m.danger$ and $pump.off$ must be simultaneous if the pump is on. Just similar to the proof in CSP and *Circus*, proving the correctness of a system by hand is very hard and error-prone in the timebands model. We strongly require a tool to assist us in mechanising the construction of the proof, which will be detailed discussed in the future work.

4.2 Banking system

The banking system example models a small part of normal banking business, which simply consists of two modules: a personal banking and a support for point of sale (POS). This example is a single-user system and it only includes two basic operations: paying money into an account and withdrawing money from the account. The POS module is also a process of withdrawing money but in a different way from withdrawing cash. To cooperate with other business in the future (e.g., transferring money overseas and clearing a cheque may take days), we model the major behaviour of the two modules in a hour band (H) and exhibit some technical issues in a second band (S). The biggest difficulty in modelling this system is the expression of a shared variable, which is used to denote the balance of the account. Since different modules of the banking system may randomly access the balance and all operations are based on its value, we must guarantee each module is able to exclusively manipulate the value of the shared variable. Here, we adopt one of applications of the miracle, discussed in Section 2.4, to simulate the lock algorithm to prevent other module from changing the balance when it has been occupied. Therefore, the access to the balance is modelled as follows:

$$\begin{aligned} SUM(s)_H = \quad & read!s \rightarrow SUM(s)_H \\ & \Box\ update?x \rightarrow SUM(x)_H \\ & \Box\ (s < 0) \& failure \rightarrow STOP \end{aligned}$$

where s denotes the balance of the account and the process reports a failure once the balance is below zero.

We define another specification macro as a convenient abbreviation to represent the if-then construction:

$$P \lhd b \rhd Q \mathrel{\widehat=} b \& P \Box \neg\, b \& Q$$

where b is a testable condition.

The banking system is modelled from the viewpoint of banks and then it is designed to meet some security properties, e.g., the balance should never be negative. The security and privacy of users are not discussed in this example. Therefore, in the hour band we model the behaviour of the personal banking and the POS module respectively, and subsequently put them in interleaving to check whether the system can maintain the essential security properties even if customers arbitrarily use these services. Suppose that the personal banking module just allows users to pay in and withdraw money from their accounts,

defined as follows:

$$money = \{10 * N \mid N \leftarrow \{0..10\}\}$$
$$Event\ payin, withdraw : H : money$$
$$Event\ receipt, cash : H$$

$$\begin{aligned} ACC = {} & payin?x \rightarrow read?y \ddagger update!(x+y)\,;\, receipt \rightarrow ACC \\ & \Box\ withdraw?x \rightarrow read?y \rightarrow \left(\begin{array}{l} decline \rightarrow ACC \\ \lhd\ x < y\ \rhd \\ \ddagger update!(y-x)\,;\, cash \rightarrow ACC \end{array} \right) \end{aligned}$$

where *money* is used to give the bound of the bank balance in order to control the state space in a reasonable size when model checking it. We can also make the bound of the account balance generic such as natural numbers if proving it in theorem-proving approaches. In any case the definition of *money* is not the major concern of the example. The module ACC accepts the request of paying-in and gives a receipt to denote the success of the transaction. Alternatively, it acknowledges the request of withdrawing cash by checking the balance of the account to decide whether this transaction can be continued.

In the hour band, the POS module is rather similar to the withdrawing process of ACC, and it may be defined as follow:

$$POS = request?x \rightarrow read?y \rightarrow \left(\begin{array}{l} decline \rightarrow POS \\ \lhd\ x < y\ \rhd \\ \ddagger update!(y-x)\,;\, auth \rightarrow POS \end{array} \right)$$

Finally, the system is constructed by putting ACC and POS in interleaving to provide individually different services, and communicating each other through the shared variable to maintain the integrity and security of the system.

$$SYS_H = (ACC \mid\mid\mid POS) \underset{\{read, update\}}{\parallel} SUM(0)$$

The most important security property for banks is that the account balance of customers cannot become negative. To verify this property, we simply check whether the event *failure* appears in the traces of the system. Note that we have not even mentioned time in modelling the system so far, and consequently the verification can be executed in the model checker FDR. We may temporarily use a trick to simulate the urgency property just for this case in the untimed environment of FDR. As a result, the correctness of the above security property of the system in the hour band has been proved in FDR. Notice that, for the simplicity, we do not set the top limit on the account balance and customers may cause the system to deadlock if paying in excessive money. However, this behaviour does not breach the security properties of banks.

Suppose that designers of the system mainly consider the structure (by both hierarchy and timing) and functionality in the hour band, and the proof of the security properties in this time scale can greatly reduce the risk that faults arise in the beginning of the design. Afterwards, in the second band, the designers can check detailed properties inside each module or component with more information revealed step by step. Therefore, in the second band we explore the elaborate procedures of *update*, *cash* and *auth*.

In accordance with the precision (120 seconds) between the hour and the second band, we first define the following activities in the second band, corresponding to the events *cash* and *auth* in the hour band:

$$\textit{Activity CASH} : S = stick\#\overline{taking}\,;\, stick \rightarrow stick \rightarrow stick\#leaving$$
$$\textit{Activity AUTH} : S = stick\#\overline{success}\,;\, stick \rightarrow stick\#removing$$

where we consider taking cash and successful authorisation are the signature events of two activities respectively.

In general, banks give a customer money only when his/her account balance has been updated. Nevertheless, this procedure is a bit different from the one of the module in supporting POS. In practice, because of various restrictions such as technical issues, the banks usually authorise transactions only if the balance is enough to afford to the requested amount of money, and thereafter (e.g., within 2 minutes) deduct the outstanding balance from the account. We may model the two different procedures of updating in the following activity:

$$\begin{aligned} UPDATE = \quad & stick \rightarrow stick \rightarrow stick\#\overline{mod} \\ \Box\ & stick\#\overline{account}\,;\, WAITC\ 100\,;\, stick\#change \end{aligned}$$

where the first line represents the normal procedure of modifying the account balance, e.g., the cashier may just take few seconds to finish the modification of the balance; another choice in *UPDATE* denotes that in the case of dealing with POS, the system marks each transaction but allows the authorisation to happen first, and then updates the balance after 100 seconds. This is also the example where an activity has two different signature events.

The imperfect approach of updating may cause various security problems, but it is still extensively applied in real systems on account of a number of reasons. For instance, because of the limited bandwidth and huge communication traffic between banks and shops, individual transaction cannot be updated timely. If we change the signature event to the event *change*, it means that each customer has to wait more than one minute for the completion of the transaction. This is beyond the tolerance of ordinary customers. Another reason that the defective approach is employed by high-security banking systems might be because this procedure is not properly modelled and verified in an real-world environment.

Our analysis of the example is able to find out a hidden hole of the system which has been utilized by criminals. The whole fraud is cooperated by two persons: one deposits a large amount of money from a counter, while the other is paying by a debit card in a shop. The transaction is authorised by the bank because of just deposited money. However, the one in the bank may immediately withdraw the same amount of cash since the balance has not been changed by the POS transaction. The bank may find this false transaction very soon, and yet both persons have left the venues. Admittedly, there are a lot of approaches used to patch the system hole. For example, we may allow the POS module to lock the amount of money which has not been taken from the account balance in a transaction. However, this approach might achieve the system security at the cost of sacrificing system efficiency and greatly increasing the difficulty of system verification.

We perhaps just slightly change the accessing mechanism of the shared variable to enable the system to avert the above hole. For instance, we impose a

short time interval before each *read* operation, so that it cannot happen simultaneously twice or more.

$$\begin{aligned} SUM(s)_H = \quad & WAIT\ \rho\,;(read!s \rightarrow SUM(s)_H) \\ & \Box\ update?x \rightarrow SUM(x)_H \\ & \Box\ (s < 0)\&failure \rightarrow STOP \end{aligned}$$

In fact, this modified module simply prevents *read* from occurring close enough. Obviously, the reading operation plays a crucial role in determining whether the transactions are able to carry on. In other words, if one is accessing the account balance, another has to wait for a while (e.g., the precision of the hour band and the second band) even if he/she merely wants to see the balance. This is a reasonable time constraint for a personal account because bankers do not expect that a customer simultaneously has two operations within the bank account.

5 Conclusion

In this report we have shown how significantly the timebands model contributes to describing and specifying dynamic behaviours of complex real-time systems at many different time scales. Viewing a system as a collection of behaviours within a finite set of bands and integrating these behaviours through linking events and corresponding activities are a natural and effective approach to separate concerns and identify inconsistencies between different time bands of the system. In general, the common way to deal with multiple-level behaviour of a system is to embed time granularity into a logical specification language. So far, most work has focused on embedding time granularity in temporal logic languages, all of which use a quite similar approach to achieve that.

For example, Interval temporal logic was originally designed for reasoning about hardware circuits [14, 27]. When modelling hardware, it is natural to look at a circuits behaviour at different granularities of time. For instance, the units of time might correspond to regularly spaced clock ticks or to nanoseconds. ITL has a temporal projection operator to denote the process of mapping from one level of time to another: $w_1\, proj\, w_2$, where w_1 and w_2 both denote ITL formulae. This operator has been implemented in the Tempura programming language [28]. Consider the formula

$$len(4) \wedge (I = 0) \wedge (I\ gets\ I + 1)$$

This is true on any interval σ whose length is 4 and in which I's value starts at 0 and increases by 1 from σ_0 to σ_1, from σ_1 to σ_2, and so on. The formula

$$len(2)\ proj\ [len(4) \wedge (I = 0) \wedge (I\ gets\ I + 1)]$$

is true on any interval σ whose length is 8 and in which I's value starts at 0 and increases by 1 from σ_0 to σ_2, from σ_2 to σ_4, and so on. The value of I in an even-numbered state is one greater than its value in the preceding even-numbered interval, but the value of I in an odd-numbered state is simply not specified. This example suggests that projection is a kind of iteration operator, and it is related to the ITL *chop* operator in the same way that the Kleene star operator is usually related to sequential composition.

Ciapessoni et al [6] have defined a logic language, based on a revised version of the specification language TRIO [12], for specifying the temporal constraints of real-time systems with different time levels. Time granularity is expressed by a finite set of disjoint temporal domains, each of which is discrete except that the finest domain may be dense. This language has two basic but key operators: the *contextual* operator $\nabla^A \mathcal{F}$ and the *projection* operator $\Box \mathcal{F}$ where $\mathcal{F}$ is a formula and A is a context term. The contextual operator ∇^A confines the evaluation of $\mathcal{F}$ to be valid only within the time domain A, and the projection operator $\Box \mathcal{F}$ evaluates to true if $\mathcal{F}$ is true at all time instants related to the current one by the projection relation. Thus, a number of useful operators derive from the two operators and their combinations can flexibly express formulae whose components are involved in different time levels. For example, the sentence, quoted from [6],

"There exists some days during which the plant works every hour"

is specified by the formula

$$\exists\, \alpha \triangle_{\alpha}^{day} \Box \nabla^{hour} work(plant)$$

where $\triangle_{\alpha}^{day} \mathcal{F}$, a derivation from the contextual operator, means $\mathcal{F}$ is true at some days at distance α from the current day. Obviously, $\Box$ makes the formula in the hour domain to hold in the day domain.

The timebands model adopts the semantics of timed *Circus*, which is a compact extension of *Circus* [34, 42]. The timed *Circus* developed in this report has dropped out Z specifications of the original *Circus* Compared to temporal logic approaches of embedding time granularity, the timebands model supports for imperative programming with concurrency operators similar to those found in CSP, and has many distinct features such as simultaneous events. Within a system from a viewpoint of the timebands model, there will always be a relation between bands but the bands need not be tightly synchronised. Some level of imprecision between different time bands makes the modelling of a system closer to the reality.

However, there is little related work on embedding time granularity in process algebra languages or hybrid specification languages. Broy [2] takes a highly abstract view of real-time interactive systems, where the system is described by a set of timed events that represent possible observations. This set is represented by a function $time : E \rightarrow TIME$, which maps each event to the time of its occurrence. Broy considers time transformers to change the timing of systems. Suppose that $trans : Time \rightarrow TIME$, then it can be used to transform the system using function composition: $time' = trans \circ time$. As a result of a time transformation, the new timing may be coarser. Two events e_1 and e_2, with the timing property $time(e1) < time(e2)$ may become simultaneous events under $time'$: we may get $time'(e1) = time'(e2)$. Broy goes on to introduce a pair of complementary functions $COA(n)$ and $FINE(n)$, which make a system's timing coarser or finer by a factor of n. They satisfy the properties

$$COA(n) \circ FINE(n).x = x$$
$$x \in FINE(n) \circ COA(n).x$$

where the dot denotes functional image. These functions permit the scaling of a timed system by any rational amount.

Proving the correctness of a system specified in the timebands model by hand is very hard and error-prone. One of our ongoing work involves mechanising the construction of proof. The employment of explicit clock-tick events and punctual clocks provides a platform for translating the timed model to the untimed model which is substantially supported by a number of successful tools such as the FDR refinement model-checker and the ProBe animator. To equalise the timed and untimed descriptions, we have to insert a lot of timers into the untimed ones. The miracle is used to force 'something must happen' in the timebands model, and therefore it must also be carefully simulated. We are going to implement the untimed descriptions in Java using JCSP [40], which is a pure Java class library providing a base range of CSP primitives plus a rich set of extensions. It also includes a package providing CSP process wrappers giving a channel interface to all Java AWT widgets and graphics operations.

In the future work we will apply the timebands framework to the analysis of more complex system such as socio-technical systems. We believe that the modelling with a time-based hierarchy is able to help develop a comprehensive foundation to dependable systems.

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.

Appendix

Lemma 1. *For a reactive process P* [7]*,*

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

Lemma 2. *For a reactive process P ,*

$$(wait \wedge ok' \wedge \mathit{II}) \,;\, P = wait \wedge ok \wedge \mathit{II}$$

Proof.

$$\begin{aligned}
&(wait \wedge ok' \wedge \mathit{II}) \,;\, P && \text{[relational calculus]}\\
=&wait \wedge ok \wedge (\mathit{II} \,;\, P) && [\mathit{II}\text{-unit}]\\
=&wait \wedge ok \wedge P && [R3]\\
=&wait \wedge ok \wedge (\mathit{II}_{rea} \lhd wait \rhd P) && \text{[propositional calculus]}\\
=&wait \wedge ok \wedge \mathit{II}_{rea} && \text{[p. c.]}\\
=&wait \wedge ok' \wedge \mathit{II}
\end{aligned}$$

□

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

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

Proof.

$$
\begin{aligned}
& \top_R ; P && [\text{simplified } \top_R] \\
=& (R1 \circ R4(\neg\, ok) \vee (wait \wedge ok' \wedge \mathit{I\!I})) ; P && [\vee\text{-; distr}] \\
=& (R1 \circ R4(\neg\, ok) ; P) \vee ((wait \wedge ok' \wedge \mathit{I\!I}) ; P) && [R1\text{-}R4] \\
=& (\neg\, ok \wedge tr \leq tr' \wedge t \leq t' ; P) \vee ((wait \wedge ok' \wedge \mathit{I\!I}) ; P) && [\text{p. c.}] \\
=& (\neg\, ok \wedge (tr \leq tr' \wedge t \leq t' ; P)) \vee ((wait \wedge ok' \wedge \mathit{I\!I}) ; P) && [\text{L1}] \\
=& (\neg\, ok \wedge tr \leq tr' \wedge t \leq t') \vee ((wait \wedge ok' \wedge \mathit{I\!I}) ; P) && [\text{L2}] \\
=& R1 \circ R4(\neg\, ok) \vee (wait \wedge ok' \wedge \mathit{I\!I}) = && \top_R \quad \square
\end{aligned}
$$

□

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

Proof.

$$
\begin{aligned}
& SKIP ; \top_R && [\text{definitions}] \\
=& R\,(true \vdash tr' = tr \wedge v' = v \wedge \neg\, wait' \wedge t' = t) ; R(\neg\, ok) \\
& && [\text{design and } R\text{-}\vee] \\
=& (R(\neg\, ok) \vee R(ok' \wedge tr' = tr \wedge v' = v \wedge \neg\, wait' \wedge t' = t)) ; R(\neg\, ok) \\
& && [\text{dist}] \\
=& R(\neg\, ok) ; R(\neg\, ok) \vee R(ok' \wedge tr' = tr \wedge v' = v \wedge && [\text{p.c.}] \\
& \qquad \neg\, wait' \wedge t' = t) ; R(\neg\, ok) \\
=& R(\neg\, ok) \vee ((ok' \wedge tr' = tr \wedge v' = v \wedge \neg\, wait' \wedge t' = t) ; && [\text{p.c.}] \\
& \qquad R4 \circ R1(\neg\, ok) \vee (wait \wedge \mathit{I\!I} \wedge ok')) \\
=& R(\neg\, ok) \vee false \vee false && [\text{p.c.}] \\
=& \top_R
\end{aligned}
$$

□

Lemma 3. *for a reactive design process P,*

$$STOP_f^f = \top_{R_f^f} = \top_{R_f^t} = R1 \circ R4(\neg\, ok)$$

Proof.

$$
\begin{aligned}
STOP_f^f &= R(true \vdash wait' \wedge tr = tr' \wedge v' = v)_f^f && [\text{def-design}] \\
&= R(\neg\, ok \vee ok' \wedge wait' \wedge tr = tr' \wedge v' = v)_f^f && [ok' \text{ is false}] \\
&= R(\neg\, ok)_f && [\text{R3}] \\
&= R1 \circ R4(\mathit{I\!I}_{rea} \lhd wait \rhd \neg\, ok)_f && [wait \text{ is false}] \\
&= R1 \circ R4(\neg\, ok)
\end{aligned}
$$

$$
\begin{aligned}
\top_{R_f^f} &= (R4 \circ R1(\neg\, ok) \vee (wait \wedge \mathit{I\!I} \wedge ok'))_f^f && [wait \text{ is false}] \\
&= R1 \circ R4(\neg\, ok) = \top_{R_f^t}
\end{aligned}
$$

□

Lemma 4. $STOP_f^t = R1 \circ R4(\neg\, ok \vee (ok' \wedge tr' = tr \wedge wait' \wedge v' = v))$

Proof.

$$
\begin{aligned}
STOP_f^t &= R(true \vdash tr' = tr \wedge wait')_f^t && \text{[design]}\\
&= R(\neg\, ok \vee (ok' \wedge tr' = tr \wedge wait'))_f^t && [R3 \text{ and } wait \text{ is false}]\\
&= R1 \circ R4(\neg\, ok \vee (ok' \wedge tr' = tr \wedge wait'))
\end{aligned}
$$

□

Law 3. $STOP \,\Box\, \top_R = \top_R$

Proof.

$$
\begin{aligned}
& STOP \,\Box\, \top_R && \text{[definition]}\\
=\;& R((\neg\, STOP_f^f \wedge \neg\, \top_{R_f^f}) \vdash && \text{[L3]}\\
& \quad (STOP_f^t \wedge \top_{R_f^t} \lhd tr' = tr \wedge wait' \rhd STOP_f^t \vee \top_{R_f^t}))\\
=\;& R(\neg\, R1 \circ R4(\neg\, ok) \vdash STOP_f^t \wedge R1 \circ R4(\neg\, ok) && \text{[p.c.]}\\
& \quad \lhd tr' = tr \wedge wait' \rhd STOP_f^t \vee R1 \circ R4(\neg\, ok))\\
=\;& R(\neg\, ok \vee R1 \circ R4(\neg\, ok) && \text{[absorption-}\vee\text{]}\\
& \quad \vee (ok' \wedge STOP_f^t \wedge R1 \circ R4(\neg\, ok) \wedge tr' = tr \wedge wait')\\
& \quad \vee (ok' \wedge (STOP_f^t \vee R1 \circ R4(\neg\, ok)) \wedge \neg\, (tr' = tr \wedge wait')))\\
=\;& R(\neg\, ok \vee (ok' \wedge (STOP_f^t \vee R1 \circ R4(\neg\, ok)) \wedge \neg\, (tr' = tr \wedge wait')))\\
& && \text{[absorption-}\vee\text{]}\\
=\;& R(\neg\, ok \vee (ok' \wedge STOP_f^t \wedge \neg\, (tr' = tr \wedge wait'))) && \text{[L4]}\\
=\;& R(\neg\, ok \vee (ok' \wedge R1 \circ R4(\neg\, ok \vee (ok' \wedge tr' = tr \wedge wait'))\\
& \quad \wedge \neg\, (tr' = tr \wedge wait'))) && \text{[absorption-}\vee\text{]}\\
=\;& R(\neg\, ok \vee (ok' \wedge R1 \circ R4(ok' \wedge tr' = tr \wedge wait') \wedge \neg\, (tr' = tr \wedge wait')))\\
=\;& R(\neg\, ok \vee false) && \text{[p.c.]}\\
=\;& R(\neg\, ok) = \top_R
\end{aligned}
$$

□

Lemma 5. *For a reactive design process P,*

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

Proof.

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

□

Lemma 6.

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

Proof.

$$\begin{aligned}
&\left(\begin{array}{c}\exists\, 1.tr', 2.tr' \bullet (P^f_f;\, (1.tr' = tr)) \wedge (\top_{Rf};\, (2.tr' = tr)) \\ \wedge\, 1.tr' - tr \upharpoonright A = 2.tr' - tr \upharpoonright A)\end{array}\right) && [L3] \\
=&\exists\, 1.tr', 2.tr' \bullet (P^f_f;\, (1.tr' = tr)) \wedge && [R1 \text{ and p.c.}] \\
&\quad (R1 \circ R4(\neg\, ok);\, (2.tr' = tr)) \wedge 1.tr' - tr \upharpoonright A = 2.tr' - tr \upharpoonright A) \\
=&(tr \leq tr' \wedge R4(P)^f_f;\, (tr' = tr)) \wedge && [\text{L1, L5 and p.c.}] \\
&\quad (tr \leq tr' \wedge R4(\neg\, ok);\, (tr' = tr)) \wedge tr' - tr \upharpoonright A = tr' - tr \upharpoonright A) \\
=&R1 \circ R4(\neg\, ok)
\end{aligned}$$

□

Lemma 7.

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

Proof.

$$\begin{aligned}
&\left(\begin{array}{c}\exists\, 1.tr', 2.tr' \bullet (P_f;\, (1.tr' = tr)) \wedge (\top^f_{Rf};\, (2.tr' = tr)) \\ \wedge\, 1.tr' - tr \upharpoonright A = 2.tr' - tr \upharpoonright A)\end{array}\right) && [L3] \\
=&\exists\, 1.tr', 2.tr' \bullet (P_f; (1.tr' = tr)) \wedge && [R1] \\
&\quad (R1 \circ R4(\neg\, ok); (2.tr' = tr)) \wedge 1.tr' - tr \upharpoonright A = 2.tr' - tr \upharpoonright A) \\
=&(tr \leq tr' \wedge P_f; (tr' = tr)) \wedge && [\text{L1,L5}] \\
&\quad (tr \leq tr' \wedge R4(\neg\, ok); (tr' = tr)) \wedge tr' - tr \upharpoonright A = tr' - tr \upharpoonright A) \\
=&R1 \circ R4(\neg\, ok)
\end{aligned}$$

□

Lemma 8.

$$((P^t_f;\, U1(out\alpha P)) \wedge (\top^t_{Rf};\, U2(out\alpha \top_R)))_{+\{v,tr\}}\, ;\, M_{\|}(A) = R1 \circ R4(\neg\, ok)$$

Proof.

$$\begin{aligned}
&((P^t_f;\, U1(out\alpha P)) \wedge (\top^t_{Rf};\, U2(out\alpha \top_R)))_{+\{v,tr\}}\, ;\, M_{\|}(A) && [L3] \\
=&((P^t_f;\, U1(out\alpha P)) \wedge (R1 \circ R4(\neg\, ok);\, U2(out\alpha \top_R)))_{+\{v,tr\}}\, ;\, M_{\|}(A) \\
& && [L1] \\
=&((P^t_f;\, U1(out\alpha P)) \wedge R1 \circ R4(\neg\, ok))_{+\{v,tr\}}\, ;\, M_{\|}(A) && [L5] \\
=&(R1 \circ R4(\neg\, ok))_{+\{v,tr\}}\, ;\, M_{\|}(A) && [\text{L1 and p.c.}] \\
=&R1 \circ R4(\neg\, ok)
\end{aligned}$$

□

Law 6. $P \underset{A}{\parallel} \top_R = \top_R$

Proof.

$$
\begin{aligned}
& P \underset{A}{\parallel} \top_R && \text{[L6, L7, L8]} \\
=& R((\neg R1 \circ R4(\neg ok) \wedge \neg R1 \circ R4(\neg ok)) \vdash R1 \circ R4(\neg ok)) && \text{[def-}\vdash\text{]} \\
=& R(R1 \circ R4(\neg ok) \vee \neg ok \vee (ok' \wedge R1 \circ R4(\neg ok))) && \text{[absorption-}\vee\text{]} \\
=& R(\neg ok)
\end{aligned}
$$

□

Lemma 9.

$$(a \to SKIP)^f_f = R1 \circ R4(\neg ok)$$

Lemma 10.

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

Lemma 11.

$$\top_{R}{}^t_f \wedge (a \to SKIP)^t_f = R1 \circ R4(\neg ok)$$

The external choice with the miracle

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

Proof.

$$(a \rightarrow SKIP) \,\Box\, \top_R \qquad [\text{def-}\Box]$$

$$= R(\neg \top_{R_f}^{f} \wedge \neg (a \rightarrow SKIP)_f^f \vdash \begin{pmatrix} \top_{R_f}^{t} \wedge (a \rightarrow SKIP)_f^t \\ \lhd tr' = tr \wedge wait' \rhd \\ \top_{R_f}^{t} \vee (a \rightarrow SKIP)_f^t \end{pmatrix}) \qquad [\text{L9,L11}]$$

$$= R(\neg R1 \circ R4(\neg ok) \vdash \begin{pmatrix} R1 \circ R4(\neg ok) \\ \lhd tr' = tr \wedge wait' \rhd \\ R1 \circ R4(\neg ok) \vee (a \rightarrow SKIP)_f^t \end{pmatrix})$$

[def-design]

$$= R(\neg R1 \circ R4(\neg ok) \wedge ok \Rightarrow ok' \wedge \begin{pmatrix} R1 \circ R4(\neg ok) \\ \lhd tr' = tr \wedge wait' \rhd \\ R1 \circ R4(\neg ok) \vee (a \rightarrow SKIP)_f^t \end{pmatrix})$$

[p. c.]

$$= R(R1 \circ R4(\neg ok) \vee \neg ok \vee (ok' \wedge R1 \circ R4(\neg ok) \wedge tr' = tr \wedge wait')$$
$$\vee (ok' \wedge \neg (tr' = tr \wedge wait') \wedge (R1 \circ R4(\neg ok) \vee (a \rightarrow SKIP)_f^t)))$$

[absorption-$\vee$ and p.c.]

$$= R(R1 \circ R4(\neg ok) \vee \neg ok \qquad [\text{absorption-}\vee \text{ and p.c.}]$$
$$\vee (ok' \wedge \neg (tr' = tr \wedge wait') \wedge R1 \circ R4(\neg ok))$$
$$\vee (ok' \wedge \neg (tr' = tr \wedge wait') \wedge (a \rightarrow SKIP)_f^t))$$

$$= R(\neg ok \vee (ok' \wedge \neg (tr' = tr \wedge wait') \wedge (a \rightarrow SKIP)_f^t)) \qquad [\text{L10}]$$

$$= R(\neg ok \vee (ok' \wedge \neg (tr' = tr \wedge wait') \wedge \begin{pmatrix} & tr' = tr \wedge a \notin ref' & \\ true \vdash & \lhd wait' \rhd & \wedge v' = v \\ & tr' = tr \frown \langle (t', a) \rangle & \end{pmatrix})$$

[absorption-$\vee$ and p.c.]

$$= R(\neg ok \vee (ok' \wedge \neg (tr' = tr \wedge wait') \qquad [\text{p.c.}]$$
$$\wedge (\neg wait' \wedge tr' = tr \frown \langle (t', a) \rangle \wedge v' = v)))$$

$$= R(\neg ok \vee (ok' \wedge (\neg wait' \wedge tr' = tr \frown \langle (t', a) \rangle \wedge v' = v))) \qquad [\text{def-}\vdash]$$

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

□

References

[1] A. W. Roscoe. Model-checking CSP. In *A Classical Mind: essays in Honour of C.A.R. Hoare*, chapter 21. Prentice-Hall.

[2] M. Broy. Time, abstraction, causality and modularity in interactive systems: Extended abstract. *Electronic Notes in Theoretical Computer Science*, 108:3–9, 2004.

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

[4] 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.

[5] Z. Chaochen, C. A. R. Hoare, and A. P. Ravn. A Calculus of Duration. pages 40:269–276, 1991.

[6] E. Ciapessoni, E. Corsetti, A. Montanari, and P. San Pietro. Embedding time granularity in a logical specification language for synchronous real-time systems. In *6IWSSD: Selected Papers of the Sixth International Workshop on Software Specification and Design*, pages 141–171, Amsterdam, The Netherlands, The Netherlands, 1993. Elsevier Science Publishers B. V.

[7] J. Clifford, J. Clifford, A. Rao, and A. Rao. A simple, general structure for temporal domains. In *Temporal Aspects in information Systems*, pages 23–30. AFCET, 1987.

[8] C. Combi, M. Franceschet, and A. Peron. Representing and reasoning about temporal granularities. *J. Log. and Comput.*, 14(1):51–77, 2004.

[9] E. Corsetti, A. Montanari, and E. Ratto. Time granularity in logical specifications. In *proceedings of the 6th Italian Conference on Logic Programmming, Pisa, Italy*, 1991.

[10] C. Fidge, I. Hayes, and G. Watson. The Deadline Command. *IEE Proceedings—Software*, 146:104–111, 1998.

[11] M. Franceschet and A. Montanari. Temporalized logics and automata for time granularity. *Theory Pract. Log. Program.*, 4(5-6):621–658, 2004.

[12] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: A logic language for executable specifications of real-time systems. *J. Syst. Softw.*, 12(2):107–123, 1990.

[13] T. R. L. Group. *The RAISE specification language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[14] J. Y. Halpern, Z. Manna, and B. C. Moszkowski. A hardware semantics based on temporal intervals. In J. Díaz, editor, *ICALP*, volume 154 of *Lecture Notes in Computer Science*, pages 278–291. Springer, 1983.

[15] J. He. Integrating CSP and DC. In *ICECCS '02: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, page 47, Washington, DC, USA, 2002. IEEE Computer Society.

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

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

[18] J. Hobbs. Granularity. In *proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, California*, pages 432–435, 1985.

[19] G. Holzmann. *Spin model checker, the: primer and reference manual.* Addison-Wesley Professional, 2003.

[20] G. J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[21] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Softw. Eng.*, 12(9):890–904, 1986.

[22] K. G. Larsen, J. Bengtsson, J. Bengtsson, K. Larsen, F. Larsson, F. Larsson, P. Pettersson, P. Pettersson, W. Yi, and W. Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *Hybrid Systems III, LNCS 1066*, pages 232–243. Springer-Verlag, 1995.

[23] A. Lawrence. HCSP: Extending CSP for Codesign and Shared Memory. In P. H. Welch and A. W. P. Bakkers, editors, *Proceedings of WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 133–156, Mar 1998.

[24] A. E. Lawrence. CSP extended: imperative state and true concurrency. *IEE Proceedings - Software*, 150(2):61–69, 2003.

[25] A. McEwan and J. Woodcock. Unifying theories of interrupts. In *proceedings of the second UTP Symposium, Trinity College Dublin*, 2008.

[26] A. Montanari, E. Ratto, E. Corsetti, and A. Morzenti. Embedding time granularity in logical specifications of real-time systems. In *proceedings of the third Euromicro Workshop on Real-Time Systems, Paris, France*, 1991.

[27] B. Moszkowski. *Reasoning about Digital Circuits.* PhD thesis, Department of Computer Science, Stanford University. (Available as technical report STANCCSC83C970.), 1983.

[28] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[29] M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in ProofPower-Z. In S. Dunne and B. Stoddart, editors, *UTP*, volume 4010 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2006.

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

[31] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *International Colloquium on Automata, Languages and Programming on Automata, languages and programming*, pages 314–323, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

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

[33] 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.

[34] A. Sampaio, J. Woodcock, and A. Cavalcanti. Refinement in Circus. In *FME '02*, pages 451–470, London, UK, 2002. Springer-Verlag.

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

[36] 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.

[37] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

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

[39] K. Wei, J. Woodcock, and A. Burns. Embedding the timed *Circus* in PVS. Technical Report .It is available at http://www-users.cs.york.ac.uk/~kun/, University of York, 2009.

[40] P. H. Welch. Process oriented design for java: Concurrency for all. In H. R. Arabnia, editor, *PDPTA*. CSREA Press, 2000.

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

[42] 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.

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

[44] J. C. P. Woodcock and A. L. C. Cavalcanti. A Tutorial Introduction to Designs in Unifying Theories of Programming. In *IFM 2004*, volume 2999 of *LNCS*, pages 40 – 66.

[45] X. Yong and C. George. An operational semantics for timed raise. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, pages 1008–1027, London, UK, 1999. Springer-Verlag.