Modelling and Implementing Complex Systems with Timebands

(Invited Paper)

Jim Woodcock∗, Marcel Oliveira†, Alan Burns∗, and Kun Wei∗

∗Department of Computer Science
University of York
York, UK
Email: {jim, burns, kun}@cs.york.ac.uk

†Computer Science Department
Federal University of Rio Grande do Norte
Natal, Brazil
Email: marcel@dimap.ufrn.br

Abstract—We describe how to use a timeband architecture to model real-time requirements. The architecture separates requirements that use different time units, producing a family of models. Each model is characterised by its granularity and precision. These models are then linked using superposition, a kind of event refinement, and a loose synchronisation of their time units, with respect to their precision. Our models are written using CSP and checked using the FDR model checker. More complicated models use Circus, the state-rich process algebra. We show how to implement such a timeband architecture using the JCSP Java class library.

Keywords—Architecture, Circus, CSP, FDR, Java, JCSP, real-time systems, requirements modelling, event refinement, time granularity, timebands.

I. INTRODUCTION

This paper concerns the modelling of complex systems using an architecture based on the structure of the system’s timing requirements. Good requirements engineering involves a lot of modelling [11], and so finding appropriate models is of the utmost importance in getting the right system. A good model can help in eliciting requirements; it can be tested for certain kinds of completeness and inconsistencies; and it can be used to test our understanding by using it to predict and to explain behaviours.

Complex real-time systems often have their requirements expressed in different time units, and this is especially the case for socio-technical systems. Modelling such requirements can lead to a structure clash [10]. Consider as an example an online airline flight reservation system. The system’s transaction processing software runs on hardware that executes its instructions in nanoseconds. The transactions themselves usually take a few seconds to complete, and when they start to take several minutes, their performance becomes unacceptable to human users. Business users are often quite specific about timing when booking their flights (“Get me to Brussells before 9am”), whereas budget travellers are more flexible (“If no flights are available on the 25th, then I could travel a day earlier or a day later”). Some airlines run weekly services to certain destinations, except when there is a Bank Holiday at either the source or the destination, and of course different countries have different sets of Bank Holidays.

There are two kinds of boundary clashes here [10]. The first comes from the fact that some of these time units do not nest exactly one within the other. For example, there are eight Bank Holidays each year in England and Wales, but they are not regularly spaced in terms of days or weeks. But there is also a boundary clash that involves two time units where one appears to be expressible exactly in terms of the other. There are exactly $864 \times 10^{11}$ nanoseconds in a day, but this is not usually a useful fact. For example, I want to arrive in Dublin on Tuesday for a meeting on Wednesday. My budget airline offers me a ticket for just £1, but the drawback is that I actually land at one minute past midnight. Technically, it is Wednesday morning, but I may be willing to regard it as fulfilling my requirement to arrive on Tuesday, as it is so cheap. Time is often expressed like this as a time unit and a precision: days are 24-hours long, plus or minus half-an-hour.

So how should we model this airline reservation system? Structure clashes are usually resolved by separating concerns. Our boundary clashes arise from trying to arrange the requirements hierarchically. Instead, we should model requirements with different time units separately, and then provide links between the members of the resulting family of models, explaining both the fuzziness of boundaries and the refinement of instantaneous events at higher bands into activities with duration at lower levels. This is essentially Burns’ timeband model [4], in which a system is decomposed to reveal different behaviours in different timebands.

Our contribution in this paper is to show how systems can be specified using a timeband architecture, and how this architecture can be realised in a programming language such as Java. For specification, we use the CSP process algebra [8], [17], [18] in our examples in this paper, and check them using the FDR model checker [6]; elsewhere, we use the state-rich process algebra, Circus [20], [16]. For implementation, we use the JCSP Java class library [19]. JCSP is a binding of the occam and CSP parallel computing model for Java. Basic packages provide processes, channels, parallel and choice (ALT) constructors. A channel interface to the Java AWT components is also included, bringing
the use of GUI elements within the CSP paradigm. Our framework is based on two main ideas: superposition and timebands. We discuss the former in Section II, and the latter in Section III, and we assume the reader is familiar with our notations.

II. SUPERPOSITION

We start by considering, apparently paradoxically, timeless timebands. Suppose we have two levels of description for modelling a university lecture. At the higher level, in the hour timeband, we describe lectures as atomic events, with no more than one event per hour for each lecturer. At the lower level, in the minute timeband, we describe a lecture as a slideshow, a sequence of events interacting with the presentation software’s API, with a few events occurring each minute. The atomic service—the lecture—is related to a lower-level protocol—the slideshow, which must be completed within the hour scheduled for the lecture (plus or minus a few minutes). In this way, operations, when viewed in a particular timeband, are instantaneous events, but when viewed in a lower timeband they become activities with an overall duration. If we abstract from time for a moment, then we see this phenomenon as an example of event refinement.

As another example, consider modelling a distributed filestore. We could describe the operation of storing a file as being atomic when we view it at a high level. Of course, if we zoom in, then we see it is actually composed of many events, and different operations actually interleave. When we store a file, we first open an account for the file; subsequent low-level operations store the file byte by byte; a finalisation operation closes the account and makes the file available for other users. The atomic service (store my file) has been refined into a protocol for achieving that effect, and concurrent runs of the protocol can easily be arranged to avoid interference because a single low-level event (the tipping point) achieves our desired high-level atomicity. In this particular example, the world turns about the low-level finalisation event. It corresponds to the higher-level event: before the finalisation occurs, the file is inaccessible; afterwards it is available. Of course, the finalisation can be achieved atomically by flipping a bit. In general, the tipping point need not be the finalisation, and could be any point within the protocol. Burns describes this as the relationship between a high-level event and a low-level signature event. We describe it as the superposition of a lower-level event onto a higher-level event.

Superposition refinement [1] enhances an algorithm by superposing one computation mechanism onto another mechanism, in such a way that preserves the behaviour of the original mechanism. This allows a simple, abstract description that is then extended with mechanisms that distribute control and state information to many lower-level processes, thus permitting efficient parallel execution of the abstract algorithm.

When an event \( c \) from a lower timeband is superposed onto an event \( a \) in a higher timeband, the two events occur simultaneously, but in a particular order (let us suppose in this paper that \( c \) occurs before \( a \)). In this sense, the event \( a \) becomes urgent as soon as the event \( c \) occurs. There is no intermediate state between the occurrence of \( c \) and the occurrence of \( a \); they are inseparable. We introduce an operator that makes an event urgent: ++. The CSP process \( c -> ++a -> \text{SKIP} \) has exactly two traces: \(<c,a>\) and \(<c,a>\). So, either nothing has happened, or exactly two things have happened (\( c \) followed by \( a \)), and the traces of the process are clearly not prefix-closed.

Our urgency operator is not expressible in the standard accounts of CSP [8], [17], [18] that use complete partial orders for the language’s denotational semantics. However, in Hoare and He’s Unifying Theories of Programming [9], [21], [5], a complete lattice is adopted for the semantic model, and the top element of the lattice corresponds to the familiar miracle from the refinement calculus: \( w : [\text{true, false}] \) [12]. This design is always guaranteed to terminate if it is started, and when it does terminate, it achieves the impossible. As well as representing intermediate results in refinement, it gives a semantics to naked guarded commands, allowing them to wait [13]. Of course, such miracles do not really exist as executable programs: they are convenient high-level fictions; so as programs, they could never be started. If we write the CSP miracle as \( \text{top} \), then the process

\[
(a -> \text{Skip}) \ [\text{top}]
\]

makes \( a \) an urgent event. When we activate this process it offers a choice between engaging in the \( a \)-event and behaving like miracle. But this choice is illusory: the miracle could never be executed, and so the choice simply does not exist. But that means that the only behaviour is to have already engaged in the \( a \)-event. There is no state in which the \( a \) has not already happened; it has become urgent. For a more detailed account of the semantics behind this argument, and for more examples of the usefulness of miracles, see [22].

Now consider a kind of vending machine modelled with two timebands. In the upper band \( \text{ATM} \), there is a rather abstract account of the behaviour of the machine. The machine is equipped with five abstract events: paying for a drink (\( \text{pay} \)), selecting tea (\( \text{select.TEA} \)), selecting a different drink (\( \text{select.OTHER} \)), obtaining a refund (\( \text{refund} \)), and dispensing tea (\( \text{dispense.TEA} \)); it is not equipped to dispense anything other than tea. We specify the behaviour in FRD’s machine-readable version of CSP, namely \( \text{CSP}_M \).

\[
\text{ATM}_I = \text{ATM}(\text{false, false, false})
\]

\[
\text{ATM(paid,tea,other)} =
\begin{cases}
\text{if (not(tea and other) and (not(tea or other) or paid))}
\end{cases}
\]

\[
\text{then (}
\end{cases}
\]

\[
\text{not paid) & pay -> ATM(true,tea,other)}
\]

\[
\text{[1] (paid and not (tea or other)) &}
\]
The three boolean parameters to ATM specify the state: whether the customer has paid, whether tea and other have been selected. Initially, all parameters are false. The recursive process ATM has an invariant which must hold, otherwise it diverges: two different selections cannot be made; and if the customer has paid, then they must also have made a selection. If the invariant holds, then the machine behaves as follows. If no payment has been made, it offers only pay; otherwise: (1) if no choice has been made, it offers the two select events; (2) it offers a refund; and (3) if tea has been chosen, it offers dispense.TEA.

The behaviour of the tea machine at the lower level is described by CTM. It differs from the ATM in that: (1) the choice for drinks is made using numeric keys, and (2) the payment is made using coins of varying values. Its invariant requires that at most three keys are pressed during any transaction and that a payment of 50p must be made before the third key is pressed.

CTM_I = CTM(0,<>)

CTM(sum,keys) =
  if #keys<3 and
    (not #keys==3 or sum>=50)
    then (sum<50) & coin?c -> CTM(sum+c, keys)

  [] (sum>=50 and #keys!=3) &
      key?k -> CTM(sum,keysˆ<k>)

  [] sum>=50 & refund -> CTM(0,<>)

  [] (sum>=50 and keys==<1,2,3>) &
      dispense.TEA -> CTM(0,<>)
  )
else
  DIV

If the sum of 50p has not yet been reached, all the user can do is to input more coins. However, if the payment has already reached 50p then the user may do the following: (1) if fewer than three keys have been pressed, press one more key; (2) get a refund; or (3) if the code <1,2,3> has been entered, have the tea dispensed.

There is an obvious connection between the two levels, as is shown by the following superposition (for simplicity, we have omitted the invariant checking, which is similar to that in the previous processes):

Super(sum,keys,paid,tea,other) =
  sum<50 & coin?c ->
    if sum+c>=50
      then ++pay ->
        Super(sum+c,keys,true,tea,other)
    else
      Super(sum+c,keys,paid,tea,other)
  [] sum>=50 and #keys!=3 &
    key?k ->
    if keysˆ<k>=<1,2,3>
      then
        ++select.tea ->
        Super(sum,keysˆ<k>,paid,tea,other)
    else
      if #keys==3
        then
          ++select.other ->
          Super(sum,keysˆ<k>,paid,tea,other)
        else
          Super(sum,keysˆ<k>,paid,tea,other)

In this superposition, the low-level event coin is superposed with the high-level event pay if the sum reaches 50p, and the low-level event key is superposed with the abstract event select.tea if the sequence <1,2,3> has been pressed, or otherwise with select.other if three other keys have been pressed. Notice how the superposition is achieving a conditional synchronisation between the two levels.

Used in this way, superposition is a device for linking two system descriptions to show how an abstract description can simulate a concrete one. In general, the superposition may be tiresome to construct, and what is needed is a protocol that achieves the same effect. Since the superposition is constructed with access to both the low- and the high-level states, it would be good if the protocol avoids this and preserves the encapsulation of data used in the original processes.

A. The Protocol

The approach to implement the superposition is to use a protocol that enforces the desired behaviour. The low-level process needs to be changed slightly to make use of an event master that interacts with a newly defined controller for the abstract process. After every event in the low-level process that is superposed with some event in the abstract process, we invoke the corresponding master as presented below.

CTM_I’ = CTM’(0,<>)

CTM’(sum,keys) =
  if (#keys<3 and ((not #keys==3) or sum>=50))
    then (sum<50) & coin?c ->
      (MASTER_COIN(sum+c); CTM’(sum+c,keys))

    [] (sum>=50 and #keys==3) &
        key?k ->
        (MASTER_KEY(keysˆ<k>); CTM’(sum,keysˆ<k>))

    [](sum>=50 & refund -> CTM’(0,<>))

    [] (sum>=50 and keys==<1,2,3>) &
        dispense.TEA -> CTM’(0,<>)
  )
else
  DIV

After each low-level coin and key event, we invoke a master process to decide whether the high-level event should be enabled or not. The master takes the decision and then instructs a controller in the high-level process accordingly.
It then waits for a confirmation from the controller before finishing.

\[\text{MASTER\_COIN}(\text{sum}) = \]
\[\begin{cases} 
\text{sum}\geq50 & \text{do.C\_PAY} \rightarrow \text{done.C\_COIN} \rightarrow \text{SKIP} \\
\text{sum}<50 & \text{donot.C\_PAY} \rightarrow \text{done.C\_COIN} \rightarrow \text{SKIP} 
\end{cases}\]

\[\text{MASTER\_KEY}\] performs a similar service enabling selection based on key pressing.

\[\text{MASTER\_KEY}(\text{keys}) = \]
\[\begin{cases} 
\text{keys}==1,2,3 & \text{do.C\_Select.TEA} \rightarrow \\
\text{done.C\_KEY} \rightarrow \text{SKIP} \\
\#\text{keys}==3 \text{ and not keys}==1,2,3 & \text{do.C\_Select.OTHER} \rightarrow \text{done.C\_KEY} \rightarrow \text{SKIP} \\
\#\text{keys}<3 & \text{donot.C\_Select.ANY} \rightarrow \\
\text{done.C\_KEY} \rightarrow \text{SKIP} 
\end{cases}\]

The process in the higher is enslaved to the master.

\[\text{ATM\_CTRL} = \]
\[\begin{cases} 
\text{do.C\_PAY} \rightarrow \text{pay} \rightarrow \text{done.C\_COIN} \rightarrow \text{ATM\_CTRL} \\
\text{donot.C\_PAY} \rightarrow \text{done.C\_COIN} \rightarrow \text{ATM\_CTRL} \\
\text{do.C\_Select.TEA} \rightarrow \text{select.TEA} \rightarrow \\
\text{done.C\_KEY} \rightarrow \text{ATM\_CTRL} \\
\text{do.C\_Select.OTHER} \rightarrow \text{select.OTHER} \rightarrow \\
\text{done.C\_KEY} \rightarrow \text{ATM\_CTRL} \\
\text{donot.C\_Select.ANY} \rightarrow \\
\text{done.C\_KEY} \rightarrow \text{ATM\_CTRL} \\
\text{kill} \rightarrow \text{SKIP} 
\end{cases}\]

Left to its own devices, \text{ATM\_CTRL} would run indefinitely; however, if the original processes terminate, the controller should also terminate. For this reason, we add the termination event \text{kill}. The protocol uses a controlled version of the high-level process \text{ATM\_I, CONTROLLED\_ATM}, presented below. Both the controller and the original process, synchronise on the abstract superposed events only.

\[\text{CONTROLLED\_ATM} = \]
\[\text{ATM\_I} \left[\begin{array}{c}
\{|{\text{pay, select}}\}\right] \text{ATM\_CTRL}
\]

The low-level process \text{CTM\_I'} is replaced by its derivation \text{CTM\_I''} presented above. However, after its termination, we need to tell the controller to terminate, and this is a standard behaviour in our approach. We define a process \text{KILLER}(\text{P}, \text{nmasters}) that behaves like \text{P} but synchronises on \text{kill} after termination of \text{P}.

\[\text{KILLER}(\text{P}, \text{nmasters}) = \]
\[\text{P;}
\left[\begin{array}{c}
\{|{\text{kill}}\}|}
\text{m}:{1..\text{nmasters}} \rightarrow \text{kill} \rightarrow \text{SKIP}
\right]
\]

For each master, there is a different synchronising branch on \text{kill}, because in our implementation the masters are the objects who have knowledge of the \text{kill} channel.

Finally, we have the final superposed version of the tea machine, which is a parallel composition of the controlled ATM with the terminating version of the concrete CTM. The internal communication between the masters and the controller are hidden from the environment.

\[\text{MASTERS\_A} = \{|{\text{do, donot, done, kill}}\|}
\]

\[\text{TM} = \]
\[\left[\begin{array}{c}
\{|{\text{CONTROLLED\_ATM}}\}
\left[\begin{array}{c}
\{|{\text{union(MASTERS\_A, \{|\text{refund, dispense}\}})}\|}
\text{KILLER(\text{CTM\_I''},\text{2})}
\right]
\right]
\]

Using FDR, we have proved that the process \text{TM} does indeed have the behaviour expected: it is a failures-divergences refinement of the superposition. In what follows, we present the JCSP classes we implemented to provide users with an easier manner to implement superposition using JCSP.

\section{JCSP Classes}

In Figure 1, we present the UML Class Diagram for the tea machine example. We have highlighted the basic classes that can be used in the implementation of other superposition examples. The overall system is implemented as the parallel composition of the \text{TM} with its GUI, \text{TMGui}. In Figure 2, we graphically present the overall tea machine implementation. In this Figure, we indicate the super-classes of each component of the system using different boxes whose meaning are in the left-hand side of the figure. Standard one-to-one channels like \text{coin} and \text{key} are represented as simple arrows, whereas multi-synchronised channels (\text{AltingBarriers}), like \text{pay} and \text{refund}, are represented as circles. Finally, multi-synchronisation on channels that communicate values like \text{dispense} are implemented using one further dimension in the array of \text{AltingBarriers}; this is represented using bold lines coming out of the circles in the Figure.

In Figure 3, we present the execution of the tea machine example. The two top panels, \text{coin} and \text{key}, are related to corresponding concrete events. The two panels in the centre, \text{dispense} and \text{refund}, are related to the corresponding events that are present in both concrete and abstract specification. Finally, the two bottom panels, \text{pay} and \text{select}, are related to corresponding abstract events. Initially, only clicks on the \text{coin} panel are successful (the system beeps after the click). After the sum of the clicks on \text{coin} reaches 50, only a click on \text{pay} will be successful. Afterwards, we go to the selection stage on which either clicks on the keys or on \text{refund} are successful. If, however, three clicks on the keys happen, the user might only click on the right button on the \text{select} panel. The right click correspond to the right drink selected: if the keys pressed were \{1,2,3\}, then only \text{tea} can be selected; otherwise, only \text{other} can be selected. After the selection, the system enters the \text{dispense} stage. At this stage, either a click on the right button at the panel \text{dispense} or a click on \text{refund} is successful. Again, the right \text{dispense} corresponds to the selected drink. Successful clicks on the \text{dispense} panel or the \text{refund} panel switch the corresponding light on; subsequent clicks switch all lamps off.

\section{III. Timebands}

The most basic concept in our framework is that of a timeband. We consider the existence of a universal clock that produces \text{tock} events every time unit. The behaviour of this clock is quite simple: it continuously offers the
tock event until it is requested to terminate using channel kill_clock.

channel tock, kill_clock

\[
CLOCK = tock \rightarrow CLOCK \\
[] \text{kill} \_ \text{clock} \rightarrow \text{SKIP}
\]

\(CLOCK\_A = \{ | \text{tock}, \text{kill} \_ \text{clock} | \}\)

A timeband has the following parameters.

- A unique id.
- The granularity that indicates the number of universal time units that a unit on this particular timeband comprises.
- The precision that indicates the length of the window on which events might happen.
- A counter of tocks.
- A boolean variable that indicates if the events might be accepted or not, that is, if the window is opened.
- The alphabet, that is the set of events, that is controlled by this timeband.

In Figure 4 we illustrate the meaning of some of the formal arguments. It is important to notice the timeband window that controls the behaviour of the overall system: events can occur only within the window, which opens at a tick event and closes at a tack event. The length in tocks of the window is determined by the precision and the distance in tocks between each tack event is given by the gran of the timeband.

The process TIMEBAND has a recursive behaviour. On each iteration, it offers: (1) the events from the controlled alphabet if the window is opened (accept is true); (2) the choice of terminating the timeband, in which case it synchronises on kill_clock; and (3) the tock event, in which case it increments the tocks counter.

We consider, for the sake of this example, the existence of three timebands: SEC, MIN, and HOUR.

datatype TIMEBAND_ID = SEC | MIN | HOUR

datatype TIMEBAND = \{ \}

Figure 1. The Tea Machine Class Diagram
The increment behaves in three different ways. First, if the counter is zero, the timeband opens the window by synchronising on `tick` before actually incrementing the counter. However, if the counter has reached the precision, the timeband needs to close the window by synchronising on the `tack` event before incrementing the counter. If the system is not ready for this synchronisation, the universal clock might move forward and synchronise with the timeband on `tock`, in which case a `deadline` event happens before incrementing the counter. This event indicates that the system has not respected the window to achieve its activities that should have happened within the precision of the timeband breaking the specification. It is up to the system that is synchronising with the timeband to deal with the `deadline` event. For instance, the system might be implemented in such a way that the `deadline` event will raise an exception. Finally, if we are in neither of the two special cases, opening or closing the window, the timeband simply increments the counter.

The last process related to the timeband represents the timebands controller, which is defined as the parallel composition of all timebands with the universal clock. All processes synchronise on the alphabet of the clock, which is hidden from the environment.

```
accept & c ->
TIMEBAND(id,gran,prec,tocks,
   accept,alpha))
[] kill_tb.id -> kill_clock -> SKIP
[] tock ->
   INC_TOCKS(id,gran,prec,tocks,
   accept,alpha)
```

The last process related to the timeband represents the timebands controller, which is defined as the parallel composition of all timebands with the universal clock. All processes synchronise on the alphabet of the clock, which is hidden from the environment.

```
TIMEBANDS_CTRL(timebands) =
   (([[ CLOCK_A |] tb:timebands @ tb)
    [[ CLOCK_A |] CLOCK)
  \ CLOCK_A
```

When attached to a timeband, the process may use the auxiliary processes in its definition. Using the first one, `OFFER_EVENTS`, the process may offer a synchronisation on a given set of events instead of forcing this synchronisation to happen. This avoids deadlock in cases where the timeband identifies a timeout and wants to synchronise on `timeout`. In this case, the `OFFER_EVENTS` indicates an exception passing the process id.

```
OFFER_EVENTS(cs, tb, pid) =
```
An auxiliary process can be used to force the process to wait a given number of cycles of the timeband the process is attached to. For that, the process iterates \( n \) times: on each iteration, it synchronises on \( \text{tick} \) (start of cycle) and tries to synchronise on \( \text{tack} \) using the method \text{offerEvent} described next (end of cycle).

\[
\text{WAIT}(\text{cycles}, \text{tb}, \text{pid}) = \\
\text{if } (\text{cycles}==0) \\
\text{then SKIP} \\
\text{else} \\
\text{tick}.\text{tb} \rightarrow \text{OFFER EVENTS}((\text{tack}.\text{tb}),\text{tb},\text{pid}); \\
\text{WAIT}(\text{cycles}-1,\text{tb},\text{pid})
\]

We are now able to define the timebands. For each timeband, we have to define its granularity, precision, and alphabet that it controls. For instance, we present below the definition of the SEC timeband, which is an instantiation of a \text{TIMEBAND} identified as SEC, a granularity of 3, a precision of 3, which initially has 0 tocks and does not accept any event in the controlled alphabet.

\[
\text{SEC}_\text{GRAN} = 3 \\
\text{SEC}_\text{PREC} = 2 \\
\text{SEC}_\text{A} = \{| \ldots |\} \\
\text{SEC}_\text{TB} = \text{TIMEBAND}(\text{SEC},\text{SEC}_\text{GRAN}, \\
\text{SEC}_\text{PREC},0,\text{false},\text{SEC}_\text{A})
\]

The MIN timeband and the HOUR timeband may be defined similarly. However, as there is a dependence between them we might use auxiliary constants that gives the relation between their granularities as we present below for the MIN timeband.

\[
\text{SEC}_\text{IN}_\text{MIN} = 60
\]
MIN_GRAN = SEC_GRAN*SEC_IN_MIN
MIN_PREC = 2
MIN_A = {1, ... , 1}
MIN_TB = TIMEBAND(MIN,SEC_GRAN,
MIN_PREC,0,false,MIN_A)

Similarly, we may define the HOUR timeband and any other
timeband we wish. Finally, we are able to define the process
that represents all timebands using the timebands controlled
previously presented.

TIMEBANDS =
 TIMEBANDS_CTRL({SEC_TB,MIN_TB,HOUR_TB})

This concludes the definition of the basic concepts of our
timeband framework. In the next section, we present the Java
classes that provide an easier way to use these concepts in
the Java implementation of these systems. After the next
section, we will use a case study to illustrate how all these
concepts and implementation can be used in practice.

IV. COMBINING SUPERPOSITION AND TIMEBANDS

The two features provided by our framework, superposi-
tion and timebands, may be combined in modelling a single
system. In this section, we briefly describe a simple example
that illustrates how this can be achieved.

Consider a simple example, modelling the BBC radio
time-check: “the six BBC pips”. Every hour, there is an
announcement like this, “You are listening to programmes
in the BBC World Service. The time now is four o’clock.”
This is then followed by five short pips, followed by a longer
one. The pips are distinctive sounds that used to be generated
by an oscillatory valve at a frequency of 1kHz. They begin
on the 55th second of the 59th minute of the hour. The new
hour begins at the start of the sixth pip, which is marked
out by being a little longer in duration.

In this example, our model has two processes: a high-
level (abstract) one for the hourly announcement and a low-
level (concrete) one for the short pips that antecede the long
pip that indicates that an hour has elapsed. A simplified
specification of the concrete process, PIPS, is presented
below. Initially it offers the longpip and then pips every
second four times before recurring.

PIPS =
tick -> longpip -> tack ->
tick -> pip -> tack ->
tick -> pip -> tack ->
tick -> pip -> tack ->
tick -> pip -> tack ->
PIPS

The simplified version of the abstract process BBC indicates
that an hour has elapsed after the tick event and before
the tack event.

BBC(i) = tick -> hour.i -> tack -> BBC((i+1)%24)

However, it is clear that the first process needs to offer its
ticks and tacks in the second band (with time progressing
irregularly), and the second process needs to offer the same
events in a different timeband, the hours. Furthermore, the

PIPS’ =
tick.SECOND ->
OFFER_EVENTS({longpip},SECOND,PIPS_id);
MASTER_LONGPIP;
OFFER_EVENTS({tack.SECOND},SECOND,PIPS_id);
WAIT(PIPS_WAIT,SECOND,PIPS_id);
tick.SECOND ->
OFFER_EVENTS({pip},SECOND,PIPS_id);
OFFER_EVENTS({tack.SECOND},SECOND,PIPS_id);
tick.SECOND ->
OFFER_EVENTS({pip},SECOND,PIPS_id);
OFFER_EVENTS({tack.SECOND},SECOND,PIPS_id);
tick.SECOND ->
OFFER_EVENTS({pip},SECOND,PIPS_id);
OFFER_EVENTS({tack.SECOND},SECOND,PIPS_id);
tick.SECOND ->
OFFER_EVENTS({pip},SECOND,PIPS_id);
OFFER_EVENTS({tack.SECOND},SECOND,PIPS_id);
PIPS’

The process MASTER_LONGPIP simply indicates to the
controller of the abstract process that it must execute an
hour event and waits for its indication to move on. There
are no conditions on the superposition, so the do synchro-
nisation is always enabled and the donot is never enabled.

MASTER_LONGPIP =
{true} & do.C_HOUR -> done.C_LONGPIP -> SKIP
() (false) & donot.C_HOUR ->
done.C_LONGPIP -> SKIP

Finally, we provide the PIPS process with the capability
to deal with exception handling by giving it to the
MASTERED_PROCESS.

EH_PIPS’ =
MASTERED_PROCESS(PIPS’,PIPS_id,SYSTEM_EXIT_REQ)

As for the previous example, when the deadline expires,
the system terminates. This is indicated by using the previ-
ously described process SYSTEM_EXIT_REQ as argument
to the invocation of MASTERED_PROCESS.

The BBC process works on the HOUR timebands: ticks
and tacks are controlled by this timeband.

BBC_I = BBC(0)
BBC(i) =
tick.HOUR ->
OFFER_EVENTS({hour.i}, HOUR, BBC_id);
OFFER_EVENTS({tack.HOUR}, HOUR, BBC_id);
BBC((i+1) % 24)

The controller enables the hour event only if it receives an indication from the master via event do.C.HOUR. As with the other controllers previously presented, it is terminated using channel kill.

\[ \text{BBC_CTRL} = \]
do.C.HOUR \rightarrow hour?i \rightarrow
done.C_LONGPIP \rightarrow BBC_CTRL
[] donot.C.HOUR \rightarrow done.C_LONGPIP \rightarrow BBC_CTRL
[] kill \rightarrow SKIP

Finally, the controlled BBC is the parallel composition of the original BBC process with its controller. We have also enabled this controlled version of the BBC process to deal with exception handling.

\[ \text{CTRLLED_BBC = BBC_I \[\{|\text{hour}|\}\]}\] \text{BBC_CTRL}
\[ \text{CTRLLED_BBC_EH =}
\text{MASTERED_PROCESS(CTRLLED_BBC,BBC_id, SYSTEM_EXIT_REQ)}\]

The system is the parallel composition of the controlled BBC, CTRLLED_BBC_EH, with the PIPS. For both of them, we consider their versions EH that take exception handling into consideration. As we did for the tea machine, TM, we use the process KILLER(EH_PIPS’,2) that behaves like EH_PIPS’ but synchronises on kill after the termination of EH_PIPS’.

We may now specify the superposed version of the BBC example, which is a parallel composition of the controlled BBC process with the terminating version of the concrete pipes.

\[ \text{UNTIMED_BBC =}
\text{CTRLLED_BBC_EH \[
\text{\{\{MASTERS_A\}\}}\]}
\text{KILLER(EH_PIPS’,2)}\]

Next, we add the timebands by composing them in parallel with the untimed version of the BBC example.

\[ \text{TB_BBC_WCLOCK =}
\text{TIMEBANDS \[
\text{\{\{TIMEBANDS_CTRL_A\}\}}\]}
\text{UNTIMED_BBC}
\text{\} INTERNAL_A}\]

The composition of the system with the exception handling process in next in our specification.

\[ \text{EH_BBC_WCLOCK =}
\text{(TB_BBC_WCLOCK \[
\text{\{\{EXC_HANDLER_A\}\}}\]}
\text{EXCEPTION_HANDLER)}\]
\text{\} EXC_HANDLER_A}\]

Finally, we have the bbc example SYS_BBC_WCLOCK, which is the composition of the BBC example and the timebands with the system termination handler.

\[ \text{SYS_BBC_WCLOCK =}
\text{SYS_PROC(EH_BBC_WCLOCK \[
\text{\{SYS_HANDLER_A\}\}]
\text{SYSTEM_HANDLER}}\]

In what follows, we briefly describe the implementation, using our framework, of this example, which considers both superposition and timebands.

B. Implementing the BBC Example

In Figure 5, we graphically present the overall implementation of the mine pump. As for Figure 2, we indicate the super-classes of each components of the system using different boxes whose meaning are in the left-hand side of the figure. Standard one-to-one channels like do.hour are represented as simple arrows whereas multi-synchronised channels (AltingBarriers) like pip are represented as circles. Finally, multi-synchronisation on channels that communicate values like hour are implemented using one further dimension in the array of AltingBarriers; this is represented using bold lines coming out of the circles in the Figure.

The implementation of this example has been achieved based on the class diagram presented in Figure 6. As well as the previously presented classes related to the timebands and superposition, we also have classes that implement the controlled BBC process and its components, and classes that implement the PIPS. Each local process previously presented in the CSP specification has a corresponding implementing Java class. The overall system BBCWorldClock has its own GUI, and so extends the GUIAbleCSProcess. The superposition is achieved by creating an event master MasterLongPip that interacts with the BBCControl, an EventSlave. The ControlledBBC is implemented as the parallel composition of the original implementation of the BBC with its controller. Finally, the original implementation of the Pips is extended to change the behaviour after the communication on longpip. We present below the constructor of this extension, PipsDash, which adds the master to this class. We also have its overriding version of method that corresponds to the behaviour after the communication on longpip: it simply executes the corresponding master.

```java
public PipsDash(MasterLongPip masterLongPip,
                AltingBarrier pip,
                AltingBarrier longpip) {
    super(pip, longpip);
    this.addMaster(masterLongPip);
}
```

```java
void dealWithLongPip_0() {
    super.dealWithLongPip_0();
    super.getMaster(MASTER_LONG_PIP).run();
}
```

Both processes whose behaviour depend on the timebands, Pips (and hence PipsDash) and BBC, extend the class TBCSProcess. Finally, we have the BBCWorldClock as the parallel composition of its ControlledBBC, its PipsDash, an its TimebandsControl. Their implementation is very similar to that previously presented. The principle for executing the system is the same: instantiate the channels and barriers; instantiate the masters; instantiate the process possibly using the masters; instantiate the timebands; add the processes to the right timebands; configure the
timebands; instantiate the timebands controller; add the
timebands to the controller; configure the controller; execute
the process in parallel with the controller. In our example,
we have provided the BBCWorldClock with a GUI, which
pips on synchronisation on the pip channel and displays the
current hour i on synchronisation on the event hour.i.

V. RELATED WORK

ITL (interval temporal logic) was originally designed for
reasoning about hardware circuits [7], [14], where different
granularities of time arise naturally. ITL has a temporal
projection operator that is used to map between granularities.
The operator is implemented in the Tempura programming
language [15]. The formula

\[\text{len}(4) \land (I = 0) \land (I \text{ gets } I + 1)\]

is true on any interval \(\sigma\) whose length is 4 and in which \(I\)'s
value starts at 0 and increases by 1 from \(\sigma_0\) to \(\sigma_1\), from \(\sigma_1\)
to \(\sigma_2\), and so on. The formula

\[\text{len}(2) \text{ proj } [\text{len}(4) \land (I = 0) \land (I \text{ gets } I + 1)]\]

is true on any interval \(\sigma\) whose length is 8 and in which
\(I\)'s value starts at 0 and increases by 1 from \(\sigma_0\) to \(\sigma_2\), from
\(\sigma_2\) to \(\sigma_4\), 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.

Bettini introduces a glossary of time granularity con-
cepts [2]. Their work is not committed to a particular
model of time, which could be discrete (such as the natural
numbers), dense (such as the rationals), or continuous (such
as the reals). The time domain is discretised into countable
granes of time, and an interpretation function relates the
index of each granule to an interval in the time domain.
Examples of granular time include the following:

- The second timeband: \(\langle 0s, 1s, 2s, 3s, 4s, \ldots \rangle\).
- The hour timeband: \(\langle 1\text{o’clock}, 2\text{o’clock}, 3\text{o’clock}, \ldots \rangle\)
- The Bank Holiday timeband:

\(\langle \text{New Year’s Day, Good Friday, Easter Monday, May Day, Spring Bank Holiday, Summer Bank Holiday, Christmas Day, Boxing Day} \rangle\)

- The Bank Holiday 2009 timeband:

The hour timeband with precision ±5s:

\[\langle 00:59:55..01:00:05, 01:59:55..02:00:05, 02:59:55..03:00:05 \rangle\]

Granularities are either comparable or incomparable; if they are comparable, then one is coarser than the other, or vice versa.

Broy 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 [3]. This set is represented by a function \(\text{time} : E \rightarrow \text{TIME}\), that maps each event to the time of its occurrence. Broy considers time transformers to change the timing of systems. Suppose that \(\text{trans} : \text{Time} \rightarrow \text{TIME}\), then it can be used to transform the system using function composition: \(\text{time}' = \text{trans} \circ \text{time}\). 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

\[ \text{COA}(n) \circ \text{FINE}(n), x = x \]
\[ x \in \text{FINE}(n) \circ \text{COA}(n).x \]

where “\circ” denotes functional image. These functions permit the scaling of a timed system by any rational amount.

VI. CONCLUSION

We have presented timebands, a technique for structuring the architecture of systems that exhibit rich timing behaviour that has to be described using different time units. We have concentrated on two issues: synchronising abstract atomic behaviours (services) with the concrete activities (protocols) that realise them; and loosely synchronising the treatment of different time units in various timebands. We have shown how to implement our approach in a systematic way that avoids intruding on data encapsulated within a timeband, and has minimal disruption of reactive behaviour. Our chosen implementation technology, JCSP, gives a direct way of executing the abstractions used in our specification technique, in this case CSP.

ACKNOWLEDGMENTS

This work was carried out under EPSRC grant EP/E001580/1 on Interdisciplinary Design and Evaluation of Dependability (INDEED). It was conducted while the first author was on a sabbatical visit to the Federal University of Rio Grande do Norte, and he is grateful stimulating intellectual environment provided by the department. The ideas presented in the paper have benefited from discussions with Stuart Anderson, Robin Bloomfield, Ana Cavalcanti, Jin Song Dong, Leo Freitas, Ian Hayes, Cliff Jones, and Ian Sommerville.

REFERENCES