A Relative Timed Semantics for BPMN

Peter Y. H. Wong [1] and Jeremy Gibbons [2]

Computing Laboratory, University of Oxford, United Kingdom

Abstract

We describe a relative-timed semantic model for Business Process Modelling Notation (BPMN). We define the semantics in the language of Communicating Sequential Processes (CSP). This model augments our untimed model by introducing the notion of relative time in the form of delays chosen non-deterministically from a range. We illustrate the application by an example. We also show some properties relating the timed semantics and BPMN's untimed process semantics by exploiting CSP refinement. Our timed semantics allows behavioural properties of BPMN diagrams to be mechanically verified via automatic model-checking as provided by the FDR tool.

Keywords: business process, CSP, refinement, timed semantics, verification, workflow

1 Introduction

Modelling of business processes and workflows is an important area in software engineering. Business Process Modelling Notation (BPMN) allows developers to take a process-oriented approach to modelling of systems. In our previous work [16] we have given an abstract syntax using Z [20] and an untimed process semantics in the language of CSP [14] to a subset of BPMN [12]. However, due to the lack of a notion of time, this semantics is not able to precisely model activities running concurrently when temporality becomes a factor;

For example, Figure 1 shows a simplified breast cancer clinical trial adapted from the Neo-tango trial protocol [4]. This BPMN representation of clinical trial is based on a new observation workflow model and its corresponding transformation to BPMN [18]. Note the clinical trial specification used throughout this paper is by no means not an accurate representation of real trial. In a clinical study it is important that interventions are carried out safely and effectively, and often interventions *must* satisfy a set of oncological safety principles [7]. In this example we will focus on

[1] Email: peter.wong@comlab.ox.ac.uk
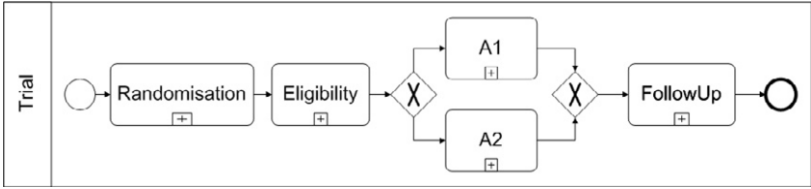[2] Email: jeremy.gibbons@comlab.ox.ac.uk

Fig. 1. A simplified clinical trial

the set of interventions $A2$ denoted as a BPMN *subprocess* state in Figure 1. An expanded version of $A2$ is shown in Figure 2 and below we show the schedule of
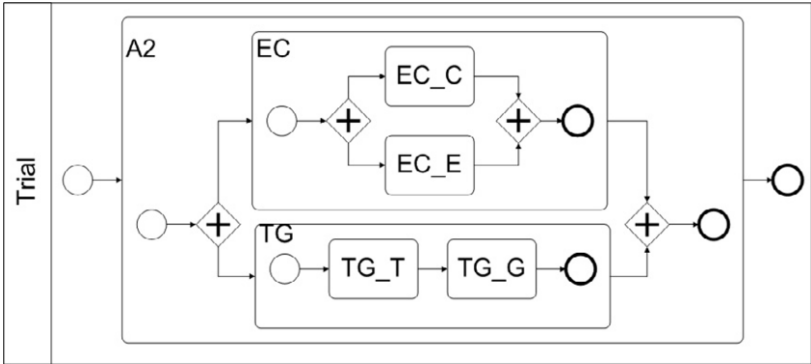

Fig. 2. A set of clinical interventions

each drug administration, we have omitted dosage for simplicity.

- EC_C - Cyclophosphamide, every 14 days to 20 days
- EC_E - Epirubicin, every 18 days to 21 days
- TG - Paclitaxel, every 5 days to 10 days followed by Gemcitabine, upto 10 days

One of the safety principles is *Sequencing* and it ensures each intervention "order(s) (essential) actions temporally for good effect and least harm". Here we are interested in the following particular instance of this principle for interventions $A2$.

No more than one dosage of gemcitabine (TG_G) may be given after the administration of cyclophosphamide (EC_C) and before epirubicin (EC_E).

It is these types of properties that we would like to verify the BPMN representation against, while careful calculation could reveal whether or not this trial specification does indeed satisfy the property and hence is "safe", we are going to show how the semantic model introduced in this paper allows us to mechanically verify the trial specification via automatic model-checking as provided by the FDR tool.

The rest of this paper is structured as follows. Section 2 gives an introduction to BPMN; an introduction to CSP [14] and Z [20], which are used throughout this paper, is given in the Appendix. In Section 3 gives an overview of our syntactic description of BPMN. Section 4 describes briefly our relative timed semantics. In Section 5 we show some properties relating the timed and untimed models based on CSP refinements, and revisit the example to show how the relative-timed model

may be used to verify against the sequencing rule. We conclude this paper with a comparison with related work. The complete formal definition of the timed model may be found in our longer paper [17].

2 BPMN

States in our subset of BPMN, shown in Figure 3, can either be pools, tasks, sub-processes, multiple instances or control gateways, each linked by a normal sequence, an exception sequence flow, or a message flow. A normal sequence flow can be either incoming to or outgoing from a state and have associated guards; an exception sequence flow, depicted by the states labelled *task**, *bpmn**, *task*** and *bpmn***, represents an occurrence of error within the state. While sequence flows represent control flows within individual *local* diagrams, message flows represent unidirectional communication between states in different local diagrams. A *global* diagram hence is a collection of local diagrams connected via message flows.
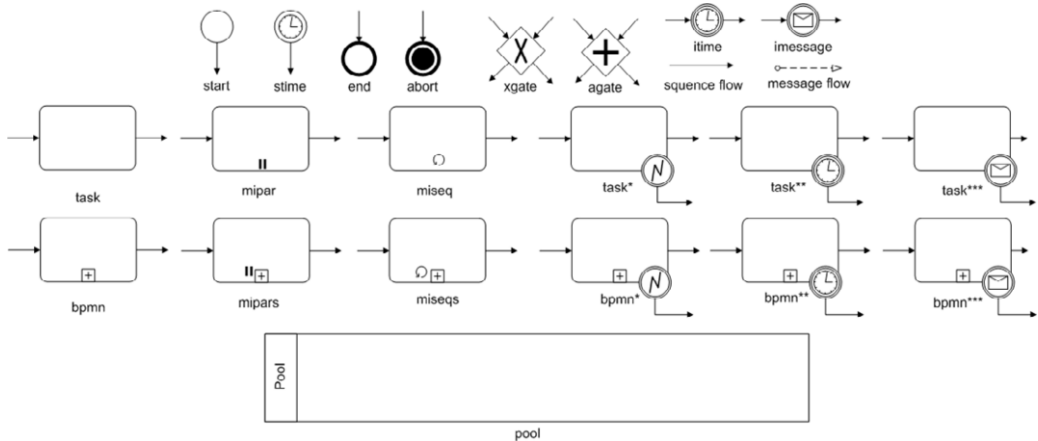

Fig. 3. States of BPMN diagram

In Figure 3, there are two types of start state, *start* and *stime*. A *start* state models the start of the business process in the current scope by initiating its outgoing transition; it has no incoming transition and only one outgoing transition. The *stime* state is a variant start state; it initiates its outgoing transition when a specified duration has elapsed. There are also two types of intermediate state, *itime* and *imessage*. An *itime* state is a delay event; after its incoming transition is triggered, the delay event waits for the specified duration before initiating its outgoing transition. An *imessage* state is a message event; after its incoming transition is triggered, the message event waits until a specified message has arrived before initiating its outgoing transition. Both types of state have a maximum of one incoming transition and one outgoing transition.

There are two types of end state, *end* and *abort*. An *end* state models the successful termination of an instance of the business process in the current scope by initialisation of its incoming transition; it has only one incoming transition with no

outgoing transition. The *abort* state is a variant end state; it models an unsuccessful termination, usually an error of an instance of the business process in the current scope.

Our subset of BPMN contains two types of decision state, *xgate* and *agate*. Each of them has one or more incoming sequence flows and one or more outgoing sequence flows. An *xgate* state is an exclusive gateway, accepting one of its incoming flows and taking one of its outgoing flows; the semantics of this gateway type can be described as an exclusive choice and a simple merge. An *agate* state is a parallel gateway, which waits for all of its incoming flows before initialising all of its outgoing flows.

A *task* state describes an atomic activity, and has exactly one incoming and one outgoing transition. It takes a unique name for identifying the activity. In the environment of the timed semantic model, each atomic task must take a positive amount of time to complete. A *bpmn* state describes a subprocess state. It is a business process by itself and so it models a flow of BPMN states. In this paper, we assume all our subprocess states are expanded [12]; this means we model the internal behaviours of the subprocesses. The state labelled *bpmn* in Figure 3 depicts a collapsed subprocess state where all internal details are hidden; this state has exactly one incoming and one outgoing transition.

Also in Figure 3 there are graphical notations labelled *task**, *bpmn**, *task***, *bpmn***, *task**** and *bpmn****, which depict a task state and a subprocess state with an exception sequence flow. There are three types of exception associated with task and subprocess states in our subset of BPMN states. Both states *task** and *bpmn** are examples of states with an *ierror* exception flow that models an interruption due to an error within the task or subprocess state; the states *task*** and *bpmn*** are examples of states with a timed exception flow, and model an interruption due to an elapse of the specified duration; the states *task**** and *bpmn**** are examples of states with a message exception flow, and model an interruption upon receiving the specified message. Each task and subprocess state can have a maximum of one timed exception flow, although it may have multiple error and message exception flows.

Each task and subprocess may also be defined as *multiple instances*. There are two types of multiple instances in BPMN: the *miseq* state type represents serial multiple instances, where the specified task is repeated in sequence; in the *mipar* state type the specified task is repeated in parallel. The types *miseqs* and *mipars* are their subprocess counterparts.

The graphical notation *pool* in Figure 3 forms the outermost container for each local diagram, representing a single business process; only one execution instance is allowed at any one time. Each local diagram contained in a pool can also be a participant within a business collaboration (global diagram) involving multiple business processes. While *sequence flows* are restricted to an individual pool, *message flows* represent communications between pools.

3 Abstract Syntax

In this section we describe the abstract syntax of BPMN using Z notation [20]. For reasons of space, this section provides partial definitions of BPMN's abstract syntax; readers may refer to our longer paper [17] for full definitions.

We first introduce some maximal sets of values to represent constructs such as *lines*, *task* and *subprocess name*, defined as Z basic types:

$$[PName, Task, Line, Channel, Guard, Msg]$$

where *PName* is the set of diagram's names. In this paper we will only consider the semantics of BPMN timed events describing time cycles (duration) and not absolute time stamps. We define schema type *Time* to record each duration; this schema models a strictly positive subset of the six-dimensional space of the XML schema data type *duration* [21, Section 3.2.6].

$$Time \cong [year, month, day, hour, minute, second : \mathbb{N}]$$

Each type of state shown in Figure 3 is defined using the free type *Type* where each of its constructors describes a particular type of state. For example, the type of an atomic task state is defined by *task t* where *t* is a unique name that identifies that task state. Below is the partial definition.

$$Type ::= start \mid stime\langle\!\langle Time \rangle\!\rangle \mid end\langle\!\langle \mathbb{N} \rangle\!\rangle \mid abort\langle\!\langle \mathbb{N} \rangle\!\rangle \mid task\langle\!\langle Task \rangle\!\rangle \mid$$
$$xgate \mid bpmn\langle\!\langle BName \rangle\!\rangle \mid miseq\langle\!\langle Task \times \mathbb{N} \rangle\!\rangle$$

According to the BPMN specification [12], each state type has other associated attributes describing its properties; our syntactic definition has included only some of these attributes. For example, the number of loops of a sequence multiple instance state type is recorded by the natural number in the constructor function *miseq*. In this paper we call both sequence flows and exception flows 'transitions'; states are linked by transition lines representing flows of control, which may have associated guards. We give the type of a sequence flow or an exception flow, and a message flow by the following schema definitions.

$$Trans \cong [guard : Guard;\ line : Line]$$
$$Mgeflow \cong [msg : Msg;\ chn : Channel]$$

Each atomic task state specifies a delay range, $min \mathrel{..} max$, of type *Range*, denoting a non-deterministic choice of a delay within those bounds. Each task resolves its choice internally when it is being enacted.

$$Range \cong [min, max : Time \mid min \leq_T max]$$

We record the type, transitions and messageflows of each *state* by the schema *State*. Here we show a partial definition of the schema *State*, omitting the inclusion of

schema components for message flows for reasons of space.

$$State \mathrel{\widehat{=}} [type : Type; \ in, out, error : \mathbb{P} \ Trans; \ loop : \mathbb{N}; \ ran : Range]$$

Here we provide the syntactic definition of the *task* state EC_E from the example in Figure 2.

$$\begin{aligned}
&\langle\!\langle \ type \rightsquigarrow task \ EC\_E, in \rightsquigarrow t1, out \rightsquigarrow t2, error \rightsquigarrow \emptyset, loop \rightsquigarrow 0, \\
&\quad ran \rightsquigarrow \langle\!\langle \ min \rightsquigarrow \langle\!\langle year, month, hour, minute, second \rightsquigarrow 0, day \rightsquigarrow 18 \ \rangle\!\rangle, \\
&\qquad\qquad max \rightsquigarrow \langle\!\langle year, month, hour, minute, second \rightsquigarrow 0, day \rightsquigarrow 21 \ \rangle\!\rangle \ \rangle\!\rangle \ \rangle\!\rangle
\end{aligned}$$

Each BPMN diagram encapsulated by a *pool* is a local diagram and represents an individual business participant in a collaboration, built up from a well-configured finite set of well-formed states [17]. While we associate each local diagram with a unique name, a global diagram, representing a business collaboration, is built up from a finite set of names, each associated with its local diagram; we also associate each global diagram with a unique name.

4   Timed Semantics

We define a timed semantic function which takes a syntactic description of a global diagram, describing a collaboration, and returns the CSP process that models the timed behaviour of that diagram. That is, the function takes one or more *pool* states, each encapsulating a local diagram representing an individual participant within a business collaboration, and returns a parallel composition of processes each corresponding to the timed behaviour of one of the individual participants.

For each local diagram, the relative-timed semantics is the partial interleaving of two processes defined by an *enactment* and a *coordination* function. The enactment function returns the parallel composition of processes, each corresponding to the untimed aspect of a state of the local diagram; this is essentially our untimed semantics of local diagrams [16]. The coordination function returns a single process for coordinating that diagram's timed behaviour; it essentially implements a variant of the *two-phase functioning approach* adopted by real-time systems and timed coordination languages [10]. Our timed model permits automatic translation, requiring no user interaction. We will now give a brief overview of the coordination function; again for reasons of space we only present function types accompanied with informal descriptions. The complete formal definition of both the enactment and coordination functions may be found in our longer paper [17].

Informally the coordination process carries out the following steps: branch out and enact all untimed events and gateways until the BPMN process has reached time stability, that is when all *active* BPMN states are timed; order all immediate active states in some sequence $\langle t_1 \mathinner{..} t_n \rangle$ according to their shortest delay; enact all the time-ready states according to their timing information; then remove the enacted states from the sequence. The process implements these steps repeatedly until the enactment terminates.

We define the function *clock* to implement the coordination, where *TimeState* is set of *timed* BPMN states, function *allstates* recursively returns a set of states contained in a local diagram, including those contained within the diagram's subprocess states, and *begin* returns the set of start states of a local diagram.

$$clock : PName \nrightarrow Local \nrightarrow Process$$

This function takes the name of the diagram of type *PName* and its specification environment (a mapping between diagram/subprocess names and their set of states) of type *Local*, and returns a process, which first triggers the outgoing transition of one of the start states, determined by the enactment. The process then behaves as defined by the function *stable*.

$$stable : (\mathbb{P}\,State \nrightarrow Process) \nrightarrow PName \nrightarrow Local \nrightarrow \\ \mathbb{P}\,State \nrightarrow \mathbb{P}\,State \nrightarrow Process$$

The function *stable* is a higher order function; it takes some function f (for example, constructed from the function *timer* below) and a set of *active* states, and returns a process, which recursively enacts all *untimed* active states until the local diagram is *time-stable* [17] i.e. when all active states of a local diagram are timed. Going back to the example in Figure 2, states EC_C, EC_E, TG_T and TG_G are timed and when the function *stable* is applied to the syntax of the diagram initially, the process it returns will enact all states according to the sequence flows until the set of active states are { EC_C, EC_E, TG_T }, that is the diagram being time-stable. After which the function behaves as defined by the function f; in the definition of *clock*, f is the function *timer* applied with its first four arguments where the third and forth arguments are initially empty.

$$timer : PName \nrightarrow Local \nrightarrow \mathbb{P}\,State \nrightarrow \mathbb{P}\,State \nrightarrow \mathbb{P}\,State \nrightarrow Process$$

Generally the function *timer* takes the diagram's name and specification environment, a set of timed states that are active before the previous time stability (initially empty), a set of timed states that have delayed their enactment non-determinisistically (initially empty), and a set of timed states that are active during the current time stability. It orders the set of currently active timed states according to their timing information. Informally the ordering process carries out the following two steps:

- creates a subset of active timed states that has the shortest delay, we denote these states as *time-ready* [17], in our example after the first time being time-stable, the only time-ready state is state TG_T, which has the minimum delay of 5 days;
- subtracts the shortest delay from the delay of all timed states that are not time-ready to represent that at least that amount of time has passed, in our example, as TG_T is the time-ready, other active timed states EC_C and EC_E will have delays 9 to 15 days and 13 to 16 days respectively.

The function then behaves as defined by the function *trun* over the set of time-ready states and the set of active but not time-ready states.

$$trun : PName \nrightarrow Local \nrightarrow \mathbb{P}\, State \nrightarrow \mathbb{P}\, State \nrightarrow Process$$
$$trun' : PName \nrightarrow Local \nrightarrow \mathbb{P}\, State \nrightarrow Process$$
$$record : PName \nrightarrow Local \nrightarrow \mathbb{P}\, State \nrightarrow \mathbb{P}\, State \nrightarrow \mathbb{P}\, State \nrightarrow Process$$

The function *trun* returns a process that recursively enacts a subset of the currently active timed states within a given BPMN process that are time-ready. Coordinating time-ready states is achieved by partially interleaving the *execution process* returned by the function *trun'* with the *recording process* returned by the recording function *record*. The function *trun'* takes the diagram's name, specification environment and its set of time-ready states, and returns a process that interleaves the enactment of a set of processes, corresponding to its set of time-ready state. These processes terminate if either their corresponding states terminate, are cancelled, or are delayed. For each of these situations, the process will communicate a corresponding coordination event to the recording process After all the interleaved processes terminate, the function *trun'* terminates and behaves like the process $run(A) = \Box\, a : A \bullet a \rightarrow run(A)$, over the same set of coordination events, so that if any subsequent coordination contains the same time-ready states due to cycle, this process will not cause blocking. Below we show *trun'* applied to the time-ready state TG_T, where the event $starts.TG\_T$ represents the enactment of state TG_T (administration of Paclitaxel), $init.TG\_G$ represents the control flow from state TG_T to TG_G, and $finish.TG\_T$ and $delayed.TG\_T$ are terminated and delayed events of TG_T.

$$starts.TG\_T \rightarrow init.TG\_G \rightarrow finish.TG\_T \rightarrow Skip$$
$$\sqcap delayed.TG\_T \rightarrow run(\{\, finish.TG\_T, delayed.TG\_T\, \})$$

The function *record* takes the diagram's name, specification environment, its set of time-ready states and set of active timed states, and returns a process that repeatedly waits for coordination events from the execution process and recalculates the set of active states accordingly. The following rules describe the function informally:

 (i) if all time-ready states have delayed their enactments and there are no other currently active states, *record* re-calculates these states so that the states, of which the delay range has the shortest upper bound, are to be enacted;

 (ii) if all time-ready states have either been enacted or delayed, then this completes a cycle of timed coordination, and the process then behaves as defined by *stable* and proceeds with the next cycle;

(iii) if there exist time-ready states that have not been enacted or delayed, *record* waits for coordination events from the execution process.

In our example when the time-ready state TG_T is applied to *record*, the process it returns either waits for TG_T to be enacted or delayed. If TG_T is enacted,

it behaves as *stable* over a empty set of untimed states and the set of timed states $\{ EC\_C, EC\_E, TG\_G \}$ since the immediately succeeding state of TG_T is TG_G, which is a timed state (rule ii). Otherwise it will also behave as *stable* since the set of currently active states are not empty (rule ii). The coordination terminates after it enacts an *end* state of the top level diagram, for a complete definition of the semantic function, it may be found in our longer paper [17].

5 Analysis

We have implemented the semantics described in this paper as a prototype tool using the functional programming language Haskell. Readers may find a copy of the implementation from our web site [3]. The tool inputs a XML serialised representation of BPMN diagram from the JViews BPMN Modeler [9], enriched with timing information as custom properties, and translates it into an ASCII file containing CSP processes representing its behaviours expressed in machine-readable CSP [14].

The following are some results of the timed model. We say a diagram is *timed* if it contains timing information and *untimed* otherwise; every timed diagram is a *timed variant* of another untimed diagram, i.e. an untimed diagram augmented with timing information. Below is an intuitive property about timed variation.

Proposition 5.1 *Untimed Invariance.* *For any untimed local diagram, there exists an (infinite) set of timed variant diagrams such that all of the diagrams in the set are failures-equivalent under the untimed semantics.*

The CSP behaviour models traces ($\mathcal{T}$), stable failures ($\mathcal{F}$) and failures-divergences ($\mathcal{N}$) admit refinement orderings based upon reverse containment [14]. A behavioural specification R can be expressed by constructing the "least" – that is, the most non-deterministic – process satisfying it, called the characteristic process P_R. Any process Q that satisfies specification R has to refine P_R, denoted by $P_R \sqsubseteq Q$. One common behavioural property for any process might be deadlock freedom. We define the process DF to specify a deadlock freedom specification for local diagrams where events *fin.n* and *aborts.n* denote successful execution and interruption respectively [17].

$$DF = (\sqcap i : \Sigma \setminus \{\!| fin, aborts |\!\} \bullet i \to DF)$$
$$\sqcap (\sqcap n : \mathbb{N} \bullet fin.n \to Skip) \sqcap (\sqcap n : \mathbb{N} \bullet aborts.n \to Stop)$$

Definition 5.2 A local diagram is deadlock free iff the process corresponding to the diagram's behaviour failures-refines DF.

One of the results of using a common semantic domain for both timed and untimed models is that we can transfer certain behavioural properties from the untimed to the timed world. We achieve this by showing for any timed variation of any local diagram, the timed coordination process is a *responsive plug-in* [13] to the enactment process. Informally process Q is a responsive plug-in to P if Q is

---

prepared to cooperate with the pattern set out by P for their shared interface. We now formally present Reed et al.'s definition of the binary relation *RespondsTo* over CSP processes using the stable failures model.

Definition 5.3 For any processes P and Q where there exists a set J of shared events, Q *RespondsTo* P iff for all traces $s \in \text{seq}(\alpha P \cup \alpha Q)$ and event sets X

$$(s \restriction \alpha P, X) \in \mathit{failures}(P) \wedge (\mathit{initials}(P/s) \cap J^{\checkmark}) \setminus X \neq \emptyset$$
$$\Rightarrow (s \restriction \alpha Q, (\mathit{initials}(P/s) \cap J^{\checkmark}) \setminus X) \notin \mathit{failures}(Q)$$

where $\mathit{initials}(P/s)$ is the set of possible events for P after trace s and $A^{\checkmark}$ is a set of events $A \cup \{\checkmark\}$; $\checkmark$ denotes successful termination in CSP.

Proposition 5.4 *Responsiveness.* *For any local diagram p under the relative timed model where its enactment and coordination are modelled by processes E and T respectively, T RespondsTo E.*

Proof. (Sketch.) We proceed by considering each of the functions which define the coordination process, and show that for any local diagram p, if there is a set of states which may be performed by p's enactment after some process instance, then the coordination of p must cooperate in at least one of those states. We do this by showing that if the process defined by each function cooperates with p's enactment, then the sequential composition of them also cooperates with p's enactment.   □

A direct consequence of Proposition 5.4 is that deadlock freedom is preserved from the untimed to the timed setting.

Proposition 5.5 *Deadlock Freedom Preservation.* *For any process P, modelling the behaviour of an untimed local diagram, and for any process Q modelling the behaviour of a timed variant of that diagram,*

$$DF \sqsubseteq_{\mathcal{F}} P \Rightarrow DF \sqsubseteq_{\mathcal{F}} Q$$

We say a behavioural property is time-independent if the following holds.

Definition 5.6 Time Independence. A behavioural specification process S is time-independent with respect to some untimed local diagram whose behaviour is given by process P iff for any process Q modelling the behaviour of a timed variant of that diagram,

$$S \sqsubseteq_{\mathcal{F}} P \Rightarrow S \sqsubseteq_{\mathcal{F}} Q$$

As a consequence of Propositions 5.4 and 5.5 and refinements over $\mathcal{T}$, we can generalise time-independent specifications by the following result.

Proposition 5.7 *A specification process S is time-independent with respect to some untimed local diagram whose behaviour is given by the process P iff*

$$S \sqsubseteq_{\mathcal{F}} P \Leftrightarrow \mathit{traces}(S) \supseteq \mathit{traces}(P) \wedge \mathit{deadlocks}(S) \supseteq \mathit{deadlocks}(P)$$

where $traces(P)$ is the set of possible traces of process P and $deadlocks(P)$ is the set of traces on which P can deadlock.

Now we revisit the example given in Figure 2. For reasons of space, both XML representation of the diagram in figure and its corresponding generated CSP script may also be found from our web site. In this section we assume the process $A2$ to be the relative-timed behaviour of the diagram in the figure. Here we use the CSP events $starts.N$ where N is a value over the datatype $Node$ to denote administration of the respective drug.

$$Node ::= TG\_T \mid TG\_G \mid EC\_C \mid EC\_E$$

The CSP events $fin.i$ where i ranging over $\mathbb{N}$ are special events denoting the successful termination of subprocesses and diagrams, in our example we use the event $fin.0$ to denote the successful termination of the diagram.

To verify the set of clinical intervention against the sequencing rule in Section 1, we exploit CSP's stable failures semantics, that is we turn the question of property verification into a question of refinement. The following process S is the most non-deterministic CSP process satisfying the sequencing rule,

$$
\begin{aligned}
S ={}& starts.TG\_G \to S \sqcap starts.EC\_E \to S \\
&\sqcap starts.EC\_C \to T \sqcap fin.0 \to Skip \\
T ={}& starts.EC\_E \to S \sqcap starts.EC\_C \to T
\end{aligned}
$$

and here is the corresponding failures refinement assertion.

$$S \sqsubseteq_{\mathcal{F}} A2 \setminus \{\, fin.1, fin.2, fin.3, starts.TG\_T \,\}$$

We have abstracted the behaviour of the diagram by hiding part of $A2$'s alphabet because the property we are interested in only covers the set of events $\{\, starts.TG\_G, starts.EC\_E, starts.EC\_C, fin.0 \,\}$, i.e. the alphabet of the process S. When we ask FDR to check this assertion the following counterexample in terms of trace is given $\langle starts.EC\_C, starts.TG\_G \rangle$. This tells us that the event $starts.TG\_G$, denoting a dosage of gemcitabine can be given after a dosage cyclophosphamide is given, this trace is sufficient to disprove the correctness of our example against the sequence rule since a dosage of epirubicin must be after gemcitabine according to the syntactic structure of the diagram.

A more detailed analysis reveals that while cyclophosphamide may be administered after 14 days and epirubicin may only be administered after 18 days, paclitaxel may be delayed for as long as 10 days before being administered, and since gemcitabine is allowed to be administered within the 10 days, it may be given after 5 days, that is before epirubicin and after cyclophosphamide. A possible solution to this is by either restricting the duration in which cyclophosphamide and epirubicin may be administered, or delay the administration of gemcitabine. We have chosen the latter as it minimises the change of the overall clinical interventions. We achieve this by including a delay event, an *itime* state, in between states TG_T and TG_G for a delay of 16 days.

As well as describing individual business processes, BPMN may also be used to specify business collaboration where more than one business processes (participants) communicate via message flows, informally we say a participant is compatible with respect to a collaboration if it cooperates on the pattern of message flow communications. Similar to the notion of compatibility defined over untimed model [19], we formalise *time-compatibility* using CSP's responsiveness.

Definition 5.8 Time-Compatibility. Given some collaboration described by the CSP process,

$$C = (\, \| \, i : \{ \, 1 \, .. \, n \, \} \bullet \alpha T_i \circ T_i) \setminus M$$

where n ranges over $\mathbb{N}$ and M is the set of events corresponding to the message flows between its participants, whose **timed behaviour** are modelled by the processes T_i, participant T_i is time-compatible with respect to the collaboration C iff

$$\forall j : \{ \, 1 \, .. \, n \, \} \setminus \{ \, i \, \} \bullet T_i \; RespondsTo \; T_j$$

One result of formalising compatibility under our timed semantics is that, since responsiveness is *refinement-closed* under $\mathcal{F}$ [13], time-compatibility is also refinement-closed.

Proposition 5.9 *Given that the participants P_i, where i ranges over some index set, are time-compatible in some collaboration C, their refinements under $\mathcal{F}$ are also time-compatible in C.*

However, refinement closure does not capture all possible compatible participants within a collaboration. Specifically, for each participant in a collaboration there exists a *time-compatible class* of participants of which any member may replace it and preserve time-compatibility. This class may be formalised via the stable failures equivalence. This notion augments our earlier definitions in the untimed setting [16].

Definition 5.10 Time-Compatible Class. Given some local diagram name p and its specification l, we define its time-compatible class of participants $cf_T(p, l)$ axiomatically as a set of pairs where each pair specifies a BPMN diagram by its environment and the name which identifies it.

$$
\begin{array}{l}
cf_T : (PName \times Local) \nrightarrow \mathbb{P}(PName \times Local) \\
\hline
\forall p : PName; \; l : Local \bullet \\
\quad cf_T(p, l) = \\
\qquad \{ \, p' : PName; \; l' : Local \mid \\
\qquad \quad (((tsem \; p \; l) \setminus (\alpha_{process} \; p \; l \setminus mg \; p \; l)) \\
\qquad \qquad \sqsubseteq_{\mathcal{F}} ((tsem \; p' \; l') \setminus (\alpha_{process} \; p' \; l' \setminus mg \; p' \; l'))) \\
\qquad \quad \vee (tsem \; p' \; l' \setminus (\alpha_{process} \; p' \; l' \setminus mg \; p' \; l')) \\
\qquad \qquad \sqsubseteq_{\mathcal{F}} (tsem \; p \; l \setminus (\alpha_{process} \; p \; l \setminus mg \; p \; l)) \, \}
\end{array}
$$

where the function mg takes a description of a local diagram and returns a set of CSP events corresponding to the message flows of that diagram.

This naturally leads to the definition of the *characteristic* or the most abstract time-compatible participant with respect to a collaboration.

Definition 5.11 Characteristic Participant. Given the time-compatible class cp of some participant p, specified in some environment l, for some collaboration c, the characteristic participant of cp, specified by a pair of name and the environment, is given by the function $char_T$ applied to cp.

$$char_T : \mathbb{P}(PName \times Local) \nrightarrow (PName \times Local)$$

$$
\begin{aligned}
char_T = (&\lambda\, ps : \mathbb{P}(PName \times Local) \bullet \\
&(\mu(p', l') : (PName \times Local) \mid \\
&\quad mg\; p'\; l' = \alpha_{process}\; p'\; l' \wedge (\forall(p, l) : ps \bullet \\
&\quad\quad (tsem\; p'\; l' \sqsubseteq_{\mathcal{F}} (tsem\; p\; l(\alpha_{process}\; p'\; l' \setminus mg\; p'\; l')))))))
\end{aligned}
$$

The following result is a direct consequence of Proposition 5.9, and Definitions 5.10 and 5.11.

Proposition 5.12 *If a characteristic participant p of a time-compatible class cp, specified in some environment l, is time-compatible with respect to some collaboration c, then all participants in cp are also time-compatible with respect to c.*

6   Related Work and Conclusion

In this paper we introduced a relative-timed semantics for BPMN in CSP to model and reason about collaborations described in BPMN. We have adopted a variant of the two-phase functioning approach widely used in real-time systems and timed coordination languages [10]. We showed properties relating the untimed and timed models of BPMN for both local and global diagrams by using CSP's notion of responsiveness, and presented an example to demonstrate the application of the semantic model. We have subsequently implemented a prototype of the semantic function in Haskell.

To the best of our knowledge, this paper describes the first relative-timed model for a collaborative graphical notation like BPMN. Some attempts have been made to provide timed models for similar notations such as UML activity diagrams [6,8] and Workflow nets [11]. Both Guelifi et al. [6] and Eshuis [8] have defined their discrete timed semantic models in Clocked Transition System of which behavioural specifications are expressed as temporal logic formulae and verification are carried out via model checking; in Ling et al.'s work, they defined a formal semantics for a timed extension of van der Aalst's Workflow nets [15] in terms of timed Petri nets. Nevertheless, their semantics do not provide the level of abstraction required to model time explicitly in that they model discrete units of times which we believe may

not be directly applicable to the business process developers whereas our definition captures the six-dimensional space defined by W3C standards [21, Section 3.2.6]. Also unlike BPMN, their target graphical notations and hence their semantic models are not designed for analyses of collaborations where more than one diagram is under consideration. Furthermore, our semantic model has been defined in correspondence to our earlier untimed model [16] so that timed-independent behavioural properties may be preserved across both models.

As in the untimed settings, there exists many approaches in which new process calculi have been introduced to capture the notion of compatibility in collaborations and choreographies. Notable works include Carbone et al.'s End-Point and Glocal Calculi for formalising WS-CDL [2] and Bravetti et al.'s choreography calculus capturing the notion of choreography conformance [1]. Both these works tackled the problem of ill-formed choreographies, a class of choreographies of which correct projection is impossible. While the notion of ill-formed choreographies is similar to our definition of compatibility and the notion of contract refinement defined by Bravetti et al. [1] bears similarity to our definition of compatible class, they have defined their choreographies solely in terms of process calculi with no obvious graphical specification notation that could be more accessible to domain specialists.

Future work will include characterising the class of timed-independent behavioural properties suitable for BPMN; applying Dwyer et al's property specification patterns [3] to assist domain specialists to specify behavioural properties for BPMN processes; and applying the timed model to reason about empirical studies such as clinical trials against safety properties [18].

References

[1] Mario Bravetti and Gianluigi Zavattaro. Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In *Proc. of 6th International Symposium on Software Composition (SC'07)*, 2007.

[2] Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-Talbot. A Theoretical Basis of Communication-Centred Concurrent Programming. Technical report, W3C, 2006.

[3] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420. IEEE Computer Society Press, 1999.

[4] Helena Earl, Carlos Caldas, Helen Howard, Janet Dunn, and Chris Poole. Neo-tAnGo A neoadjuvant study of sequential epirubicin + cyclophosphamide and paclitaxel +/- gemcitabine in the treatment of high risk early breast cancer with molecular profiling, proteomics and candidate gene analysis, 2004.

[5] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement, FDR2 User Manual*, 1998. `www.fsel.com`.

[6] Nicolas Guelfi and Amel Mammar. A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In *APSEC05*, pages 283–290, 2005.

[7] Peter Hammand, Marek J. Sergot, and Jeremy C Wyatt. Formalisation of Safety Reasoning in Protocols and Hazard Regulations. In *19th Annual Symposium on Computer Applications in Medical Care*, October 1995.

[8] Hendrik Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, University of Twente, 2002.

[9] ILOG JViews BPMN Modeler. Available at http://www.ilog.com/products/jviews/diagrammer/bpmnmodeler/.

[10] I. Linden, J.-M. Jacquet, K. De Bosschere, and A. Brogi. On the expressiveness of timed coordination models. *Sci. Comput. Program.*, 61(2):152–187, 2006.

[11] Sea Ling and H. Schmidt. Time petri nets for workflow modelling and analysis. In *Proceedings of 2000 IEEE International Conference on Systems, Man, and Cybernetics*, pages 3039–3044, 2000.

[12] OMG. *Business Process Modeling Notation (BPMN) Specification*, February 2006. www.bpmn.org.

[13] J. N. Reed, J. E. Sinclair, and A. W. Roscoe. Responsiveness of interoperating components. *Form. Asp. Comput.*, 16(4):394–411, 2004.

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

[15] W. M. P. van der Aalst. Verification of Workflow Nets. In *ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, pages 407–426, 1997.

[16] Peter Y. H. Wong and Jeremy Gibbons. A Process Semantics for BPMN. In *Proceedings of the 10th International Conference on Formal Engineering Methods*, October 2008. To appear. Extended version available at http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/bpmnsem.pdf.

[17] Peter Y. H. Wong and Jeremy Gibbons. A Relative-Timed Semantics for BPMN (extended version), 2008. http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/bpmntime.pdf.

[18] Peter Y. H. Wong and Jeremy Gibbons. On Specifying and Visualising Long-Running Empirical Studies. In *Proceedings of 1st International Conference on Model Transformation*, volume 5063 of *LNCS*, July 2008. Extended version available at http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/transext.pdf.

[19] Peter Y. H. Wong and Jeremy Gibbons. Verifying Business Process Compatibility. In *Proceedings of 8th International Conference on Quality Software*. IEEE Computer Society Press, August 2008. To appear.

[20] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[21] XML Schema Part 2: Datatypes Second Edition, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

Appendix

CSP

In CSP [14], a process is a pattern of behaviour; a behaviour consists of events, which are atomic and synchronous between the environment and the process. The environment in this case can be another process. Events can be compound, constructed using the dot operator '.'; often these compound events behave as channels communicating data objects synchronously between the process and the environment. For reasons of space we only describe a subset of the syntax of the language of CSP that is used throughout this paper.

$$P, Q ::= P \llbracket A \rrbracket Q \mid P \llbracket A \mid B \rrbracket Q \mid P \setminus A \mid$$
$$P \square Q \mid P \sqcap Q \mid e \rightarrow P \mid Skip \mid Stop$$
$$e \quad ::= x \mid x.e$$

Process $P \llbracket A \rrbracket Q$ denotes the partial interleaving of processes P and Q sharing events in set A. Process $P \llbracket A \mid B \rrbracket Q$ denotes parallel composition, in which P and Q can evolve independently but must synchronise on every event in the set $A \cap B$;

the set A is the alphabet of P and the set B is the alphabet of Q, and no event in A and B can occur without the cooperation of P and Q respectively.

Process $P \setminus A$ is obtained by hiding all occurrences of events in set A from the environment of P. Process $P \square Q$ denotes the external choice between processes P and Q; the process is ready to behave as either P or Q. Process $P \sqcap Q$ denotes the internal choice between processes P or Q, ready to behave as at least one of P and Q but not necessarily offer either of them. An internal choice over a set of indexed processes is written $\sqcap i : I \bullet P(i)$.

Process $e \rightarrow P$ denotes a process capable of performing event e, after which it will behave like process P. The process *Stop* is a deadlocked process and the process *Skip* is a successful termination.

CSP has three denotational semantics: traces ($\mathcal{T}$), stable failures ($\mathcal{F}$) and failures-divergences ($\mathcal{N}$) models, in order of increasing precision. In this paper our process definitions are divergence-free, so we will concentrate on the stable failures model. The traces model is insufficient for our purposes, because it does not record the availability of events and hence only models what a process can do and not what it must do [14]. Notable is the semantic equivalence of processes $P \square Q$ and $P \sqcap Q$ under the traces model. In order to distinguish these processes, it is necessary to record not only what a process can do, but also what it can refuse to do. This information is preserved in *refusal sets*, sets of events from which a process in a stable state can refuse to communicate no matter how long it is offered. The set *refusals*(P) is P's initial refusals. A failure therefore is a pair (s, X) where $s \in traces(P)$ is a trace of P leading to a stable state and $X \in refusals(P/s)$ where P/s represents process P after the trace s. We write $traces(P)$ and $failures(P)$ as the set of all P's traces and failures respectively.

We write Σ to denote the set of all event names, and CSP to denote the syntactic domain of process terms. We define the semantic function $\mathcal{F}$ to return the set of all traces and the set of all failures of a given process, whereas the semantic function $\mathcal{T}$ returns solely the set of traces of the given process.

$$\mathcal{F} : CSP \rightarrow (\mathbb{P} \operatorname{seq} \Sigma \times \mathbb{P}(\operatorname{seq} \Sigma \times \mathbb{P} \Sigma))$$
$$\mathcal{T} : CSP \rightarrow \mathbb{P} \operatorname{seq} \Sigma$$

These models admit refinement orderings based upon reverse containment; for example, for the stable failures model we have

$$\_ \sqsubseteq_{\mathcal{F}} \_ : CSP \leftrightarrow CSP$$

$$\forall P, Q : CSP \bullet$$
$$\quad P \sqsubseteq_{\mathcal{F}} Q \Leftrightarrow traces(P) \supseteq traces(Q) \wedge failures(P) \supseteq failures(Q)$$

While traces only carry information about *safety* conditions, refinement under the stable failures model allows one to make assertions about a system's *safety* and *availability* properties. These assertions can be automatically proved using a model checker such as FDR [5], exhaustively exploring the state space of a system, either returning one or more counterexamples to a stated property, guaranteeing that no

counterexample exists, or until running out of resources.

Z notation

The Z notation [20] has been widely used for state-based specification. It is based on typed set theory coupled with a structuring mechanism: the schema. A schema is essentially a pattern of declaration and constraint. Schemas may be named using the following syntax:

$$Name \mathrel{\widehat{=}} [declaration \mid constraint]$$

The mathematical language within Z provides a syntax for set expressions, predicates and definitions. Types can either be basic types ($[Type]$), maximal sets within the specification, each defined by simply declaring its name, or be free types ($Type ::= element_1 \mid ... \mid element_n$), introduced by identifying each of the distinct members, introducing each element by name. By using an axiomatic definition we can introduce a new symbol x, an element of S, satisfying predicate p.

$$
\begin{array}{|l}
x : S \\
\hline
p
\end{array}
$$