A Relative Timed Semantics for BPMN

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

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. By using CSP as the semantic domain, we exploit its refinement to show some properties relating the timed semantics and BPMN's untimed process semantics. Our timed semantics allows behavioural properties of BPMN diagrams to be mechanically verified via automatic model-checking provided by the FDR tool.

Key words: business process, compatibility, 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 [11] we have given an untimed process semantics in the language of CSP [10] to a subset of BPMN. 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; this is particularly important when specifying business collaboration where the coordination of one business participant depends on the execution order of another participant's activities. For example, Figure 1 shows a simple business collaboration between participants pl and $p2$. Clearly the temporal order of tasks C and D could affect the execution of this collaboration.

The rest of this paper is structured as follows. Section 2 gives an introduction to BPMN. Section 3 gives an overview of our syntactic description

[1] This work is supported by a grant from Microsoft Research. The authors are grateful to Bill Roscoe for his insightful advice on responsiveness during this work.

[2] Email: {peter.wong,jeremy.gibbons}@comlab.ox.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

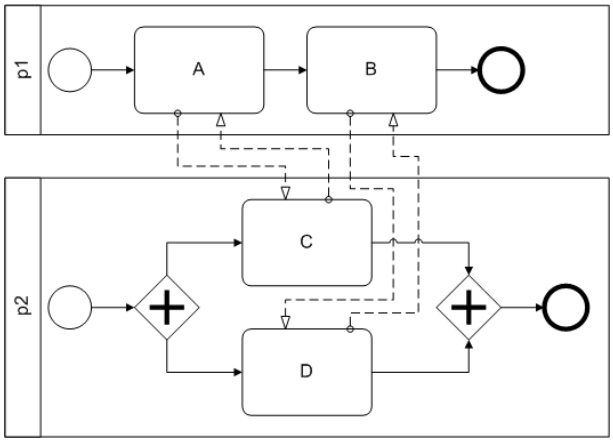

Fig. 1. A simple business collaboration

of BPMN. Section 4 describes briefly our relative timed semantics. In Section 5 we show some properties relating the timed and untimed modes based on CSP refinements, and revisit the example to show how the relative-timed model may be used to verify compatibility between participants. We conclude this paper with a comparison with related work. We assume readers have basic knowledge of the mathematical notations Z [14] and CSP [10]. The complete formal definition of the timed model may be found in our longer paper [12].

2 BPMN

States in our subset of BPMN, shown in Figure 2, can either be pools, tasks, subprocesses, 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.

In Figure 2, 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 and 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

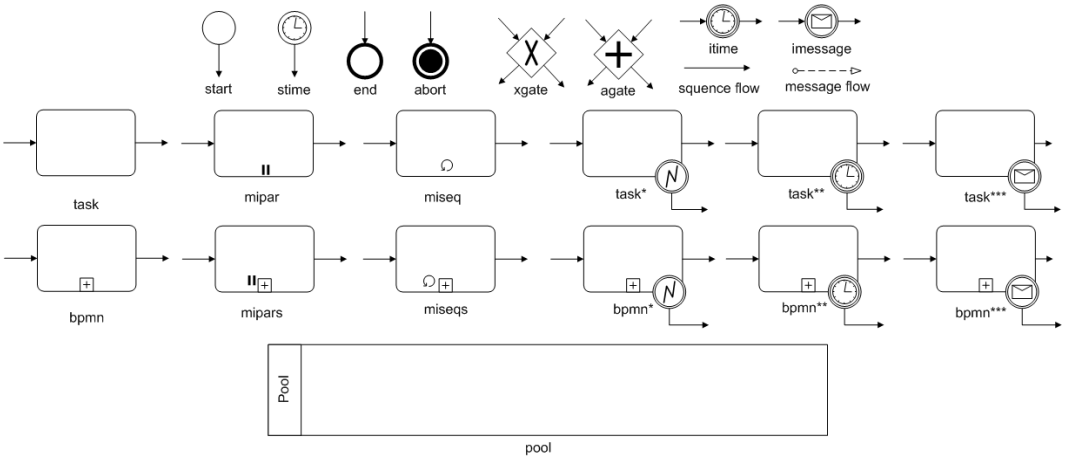

Fig. 2. States of BPMN diagram

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 takes 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 [8], this means we model the internal bebaviours of of the subprocesses. The state labelled *bpmn* in Figure 2 depicts a collapsed subprocess state where all internal details are hidden; this state has exactly one incoming and one outgoing transition.

Also in Figure 2 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 2 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. Our earlier example of a collaboration in Figure 1 illustrates message flow interaction between activities across BPMN pools.

3 Abstract Syntax

In this section we describe the abstract syntax of BPMN using Z notation [14]. For reasons of space, we have omitted certain schema and function definitions and have only concentrated the definition of a smaller subset of the BPMN states than shown in Section 2; readers may refer to our longer paper [12] for their full definitions.

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

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

We then derive subtypes *BName* and *PLName*, *InMsg*, *OutMsg*, *EndMsg* and *LastMsg* axiomatically,

$$\begin{array}{|l} InMsg, OutMsg, EndMsg, LastMsg : \mathbb{P}\, Msg \\ BName, PLName : \mathbb{P}\, PName \\ \hline \langle InMsg, OutMsg, EndMsg, LastMsg\rangle \text{ partition } Msg \\ \langle BName, PLName\rangle \text{ partition } PName \end{array}$$

The sequence of sets $\langle S_1 \,..\, S_n\rangle$ *partitions* some set T iff

$$\bigcup S_1 \,..\, S_n = T \land (\forall\, i, j : 1 \,..\, n \bullet S_i \cap S_j = \emptyset)$$

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* [15].

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

Each type of state shown in Figure 2 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.

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

According to the BPMN specification [8], 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.

$$Trans \mathrel{\widehat=} [guard : Guard;\ line : Line]$$
$$Mgeflow \mathrel{\widehat=} [msg : Msg;\ chn : Channel]$$

Each *state* records the type of its content, its transitions and message flows. Here we show a partial definition of the schema *State*, omitting the inclusion of schema components for message flows for reasons of space.

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

Each atomic task state also specifies a delay range, $min\,..\,max$, of type *Range*; the actual delay will be chosen non-deterministically from this range.

$$Range \mathrel{\widehat=} [min, max : Time \mid min \leq_T max]$$

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 [12]. 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 associates 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 [11]. The coordination function returns a single process for coordinating that diagram's timed behaviour; it essentially implements a variant of *two-phase functioning approach* adopted by coordination languages like Linda [6]. 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 accompained with informal descriptions. The complete formal definition of both the enactment and coordination functions may be found in our longer paper [12].

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 stablility; order all immediate *active* BPMN states, which are also timed by definition, 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.

$$
\begin{array}{|l}
clock : PName \pfun Local \pfun Process \\
\hline
\forall\, p : PName;\ l : Local \bullet \\
\quad clock\ p\ l = \\
\qquad \Box\, i : \bigcup \{\, s : begin\ p\ l \bullet \alpha_{trans}\ s.out \,\} \bullet \\
\qquad\quad \mathbf{let}\ \ os = (\mu\, s : states^{\sim}(l\ p) \mid i \in \alpha_{trans}\ s.in)\ \mathbf{in} \\
\qquad\quad \mathbf{if}\ os \in \{ t : TimeState \mid t \in allstates\ p\ l \} \\
\qquad\quad \mathbf{then}\ i \then (stable\ (timer\ p\ l)\ p\ l\ \emptyset\ \{\, os \,\}) \\
\qquad\quad \mathbf{else}\ i \then (stable\ (timer\ p\ l)\ p\ l\ \{\, os \,\}\ \emptyset)
\end{array}
$$

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*.

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

The function *stable* is a higher order function; it takes some function f and a set of *active* states, and returns a process, which recursively enacts all *untimed* active states until the local diagram is *time-stable* i.e. when all active states of a local diagram are timed. It then behaves as defined by the function f; in the definition of *clock*, f is the function *timer*.

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

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 nondeterminisistically (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* [12];
- 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.

It 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.

$$trun : PName \pfun Local \pfun \mathbb{P}\, State \pfun \mathbb{P}\, State \pfun Process$$

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*, where the function $trun'$ enacts all the time ready states and at the end of each state enactment, the execution process communiates coordination events to the recording process depending on whether the state has terminated, been cancelled, been interrupted or been delayed, while the function *record* receives these coordination events and recalculates the current state of the local diagram.

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

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, defined by *record*. 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.

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 describes the function informally:

- if all time ready states have delayed their enactments, *record* re-calculates these states so that the states, of which the delay range has the shortest upper bound, are to be enacted;
- 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;
- if there exist time ready states that have not been enacted or delayed, *record* waits for coordination events from the execution process.

A complete definition of the semantic function may be found in our longer paper [12].

5 Analysis

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 diagram in the set is 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 [10]. 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. A local diagram is deadlock free when all its process instances are complete. 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 [12].

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

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* [9] 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

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

where $initials(P/s)$ 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, the 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*

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

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 1, which shows a global diagram describing a collaboration between participants $p1$ and $p2$. While $p1$ performs task A then task B, $p2$ performs tasks C and D in an interleaving manner. We write $M1$ and $M2$ to denote the processes corresponding to the untimed behaviour of $p1$ and $p2$ respectively. Here we only show the definition of $M2$. First, we define $I2 = \{ start, as, c, d, aj, end \}$ to index the processes corresponding to the states in the participant $p2$. By applying the untimed semantic function to the syntactic description of $p2$, we obtain the process corresponding to it. Here the events $init.x$ denote incoming transitions of state x, $starts.t$ denote the initialisation of tasks t and $msg.a.b.m$ denote message flows from state a to b with message m. The set αP is the set of possible events performed by P.

$$\begin{aligned} M2 &= M2' \setminus \{|init|\} \\ M2' &= \mathbf{let}\ C = (\Box\, e : (\alpha M2' \setminus \{ fin.2 \}) \bullet e \rightarrow C) \,\Box\, fin.2 \rightarrow Skip \\ &\quad \mathbf{in}\ (\ \|\| i : I2 \bullet \alpha P2(i) \circ P2(i)) \,[\![\alpha M2']\!]\, C \end{aligned}$$

where for each i in $I2$, process $P2(i)$ defines the behaviour of state i. Below shows the behaviour of state c.

$$\begin{aligned} P2(start) &= init.as \rightarrow fin.2 \rightarrow Skip \\ P2(as) &= (init.as \rightarrow (init.c \rightarrow Skip \,|||\, init.d \rightarrow Skip) \fatsemi P2(as)) \end{aligned}$$

$$
\begin{aligned}
&\Box\ \mathit{fin}.2 \rightarrow \mathit{Skip}\\
P2(c) = {} &(\mathit{init}.c \rightarrow \mathit{msg}.a.c.\mathit{mi} \rightarrow \mathit{starts}.c \rightarrow \mathit{msg}.c.a.\mathit{md} \rightarrow \mathit{init}.\mathit{aj}1 \rightarrow P2(c))\\
&\Box\ \mathit{fin}.2 \rightarrow \mathit{Skip}\\
P2(d) = {} &(\mathit{init}.d \rightarrow \mathit{msg}.d.b.\mathit{mi} \rightarrow \mathit{starts}.d \rightarrow \mathit{msg}.b.d.\mathit{md} \rightarrow \mathit{init}.\mathit{aj}2 \rightarrow P2(d))\\
&\Box\ \mathit{fin}.2 \rightarrow \mathit{Skip}\\
P2(\mathit{aj}) = {} &((\mathit{init}.\mathit{aj}1 \rightarrow \mathit{Skip} \ |||\ \mathit{init}.\mathit{aj}2 \rightarrow \mathit{Skip}) \fatsemi \mathit{init}.\mathit{end} \rightarrow P2(\mathit{aj}))\\
&\Box\ \mathit{fin}.2 \rightarrow \mathit{Skip}\\
P2(\mathit{end}) = {} &\mathit{init}.\mathit{end} \rightarrow \mathit{fin}.2 \rightarrow \mathit{Skip}
\end{aligned}
$$

Their collaboration hence is the parallel composition of processes $M1$ and $M2$.

$$UC = (M1[\alpha M1 \ ||\ \alpha M2]M2) \setminus \{|msg|\}$$

CSP's failures refinement allows us to verify the behaviour modelled by a BPMN diagram against another BPMN diagram, specifying the intended behaviour. We can describe such intended behaviour of the collaboration by defining a behavioural specification as the BPMN diagram $s1$ in Figure 3. We

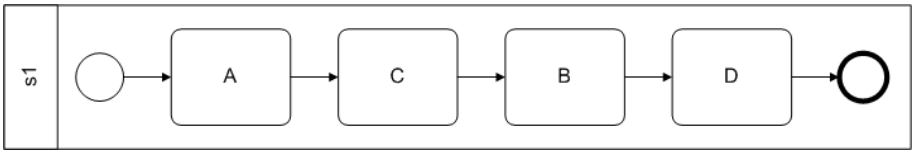

Fig. 3. Specifying the intended behaviour of collaboration between $p1$ and $p2$

write *Spec* to denote the process that models the untimed behaviour of $s1$, and can ask the refinement checker FDR [3] to check the following refinement assertion.

$$\mathit{Spec} \sqsubseteq_{\mathcal{F}} UC \tag{1}$$

This assertion tells us that the behaviour of the collaboration satisfies the specification *Spec*. According to our earlier work on compatibility [11] we can say participants $p1$ and $p2$ are *compatible* with respect to the collaboration.

Now let's suppose tasks A, B and D have a delay range from 30 minutes to 1 hour and task C has a delay range from 45 minutes to 1 hour and 15 minutes. We define the timed behaviour of the collaboration by defining the process corresponding to individual participant's coordination using the coordination function [12]. For example, Process $C2$ defines the coordination of participant $p2$.

$$
\begin{aligned}
C2 &= \mathit{init}.\mathit{as} \rightarrow (\mathit{init}.c \rightarrow \mathit{Skip} \ |||\ \mathit{init}.d \rightarrow \mathit{Skip}) \fatsemi C2_1\\
C2_1 &= ((\mathit{starts}.d \rightarrow \mathit{init}.\mathit{aj}2 \rightarrow \mathit{starts}.c \rightarrow \mathit{init}.\mathit{aj}1 \rightarrow C2_3) \sqcap C2_2)\\
C2_2 &= (\mathit{starts}.c \rightarrow \mathit{init}.\mathit{aj}1 \rightarrow \mathit{Skip} \ |||\ \mathit{starts}.d \rightarrow \mathit{init}.\mathit{aj}2 \rightarrow \mathit{Skip}) \fatsemi C2_3\\
C2_3 &= \mathit{init}.\mathit{end} \rightarrow \mathit{fin}.2 \rightarrow \mathit{Skip}
\end{aligned}
$$

and so we have process $T2$ defining the relative-timed semantics of participant $p2$. Similarly we can define the process $T1$ corresponding to $p1$. The timed

semantics of their collaboration can hence be described by process TC.

$$T2 = (M2' \,[\![\, \alpha M2' \cap \alpha C2 \,]\!]\, C2) \setminus \{\!|init|\!\}$$
$$TC = (T1[\alpha T1 \,||\, \alpha T2]T2) \setminus \{\!|msg|\!\}$$

A similar check to the refinement assertion (1) can be made on TC against *Spec*.

$$Spec \sqsubseteq_{\mathcal{F}} TC$$

When we ask FDR to check this assertion the counterexample $(\langle starts.a \rangle, \Sigma)$ is given. This tells us that the collaboration deadlocks after participant $p1$ performed task A. A more detailed analysis reveals that after starting task A, participant $p1$ sent a message to $p2$'s task C. However, while task C's maximum delay is one minute and fifteen seconds, task D's maximum delay is only one minute. Since delays are chosen internally over a range without the cooperation of the environment, participant $p2$ can choose to perform task D before task C without any agreement with $p1$.

We can now generalise the notion *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

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

As for the example above, to confirm $p1$ and $p2$ are time-incompatible with respect to the collaboration in Figure 1, we need also to show their corresponding processes $T1$ and $T3$ are deadlock-free. This can be achieved by running the following refinement checks on the FDR tool.

$$DF \sqsubseteq_{\mathcal{F}} T1 \wedge DF \sqsubseteq_{\mathcal{F}} T2$$

One result of the generalisation of compatibility under a relative-timed semantics is that, since responsiveness is *refinement-closed* under $\mathcal{F}$ [9], 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 [11].

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.

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

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.

$$
\begin{array}{|l}
char_T : \mathbb{P}(PName \times Local) \pfun (PName \times Local) \\
\hline
char_T = (\lambda\, ps : \mathbb{P}(PName \times Local) \bullet \\
\qquad (\mu(p', l') : (PName \times Local) \mid \\
\qquad\quad mg\ p'\ l' = \alpha_{process}\ p'\ l' \land (\forall (p, l) : ps \bullet \\
\qquad\qquad (tsem\ p'\ l' \sqsubseteq_{\mathcal{F}} (tsem\ p\ l(\alpha_{process}\ p'\ l' \setminus mg\ p'\ l'))))))
\end{array}
$$

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 two-phase functioning approach widely used in real-time systems and coordination languages like Linda [6]. We shown properties relating the untimed and timed models of BPMN for both local and global diagrams by using CSP's notion of responsiveness. We have also illustrated by an example how to use the timed model to verify compatibility between participants within a business collaboration.

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 [4,5,7]. However, neither do their semantics provide the level of abstraction required to model time explicitly nor do their timed models allow analyses of collaborations where more than one diagram is under consideration.

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 the following:

- characterising the class of timed-independent behavioural properties suitable for BPMN;
- automating the semantic function, possibly in Haskell as we already have a representation for BPMN [13];
- applying the timed model to reason about empirical studies against safety properties [13].

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

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

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

[6] 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.

[7] 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.

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

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

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

[11] Peter Y. H. Wong and Jeremy Gibbons. A Process Semantics for BPMN, 2007. Submitted for publication. Extended version available at http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/bpmnsem.pdf.

[12] 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.

[13] Peter Y. H. Wong and Jeremy Gibbons. On Specifying and Visualising Long-Running Empirical Studies, 2008. Submitted for publication.

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

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