A Relative Timed Semantics for BPMN

Peter Y. H. Wong Jeremy Gibbons

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 earlier untimed process semantics by introducing the notion of relative-time in the form of delays and durations over non-deterministic ranges. By using CSP as the semantic domain, we show some properties relating the timed semantics and BPMN's untimed process semantics based on existing refinement orderings defined upon CSP.

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. There are currently around forty implementations of the notation, but the notation specification adopted by the Object Management Group (OMG) [7] does not have a formal behavioural semantics, which we believe is crucial in behavioural specification and verification activities. In our previous work [11] we have given an untimed process semantics to a subset of BPMN in the language of CSP [10]. However due to the lack of notion of time, it is not able to precisely the order of activities running concurrently, this is particularly important when specifying business collaboration where the coordination of one business participant depends on the execution order of another participant's activties. Consider, for example, Figure 1 shows a trivial business collaboration between participants pl and $p2$. Clearly we would like to know what temporal properties are required for $p1$ and $p2$ to be compatible in the collaboration.

The main contribution of our work is to provide a formal relative-timed semantics for a subset of BPMN, in terms of the untimed CSP [10]. Our semantics extends our earlier untimed formalisation by introducing the notion of time via the variant of *two-phase functioning approach* employed by coordination languages such as Linda [6]. We extend the earlier untimed model by the following:

- Formalising a larger subset of BPMN in which timer events are considered. These timer events introduce timing information in the form of delays.
- Introducing duration range into atomic tasks where each task can take choose an interval of time non-deterministically over a bounded range.

By using the language and the behavioural semantics of CSP as the denotational model, we show how the existing refinement orderings defined upon CSP

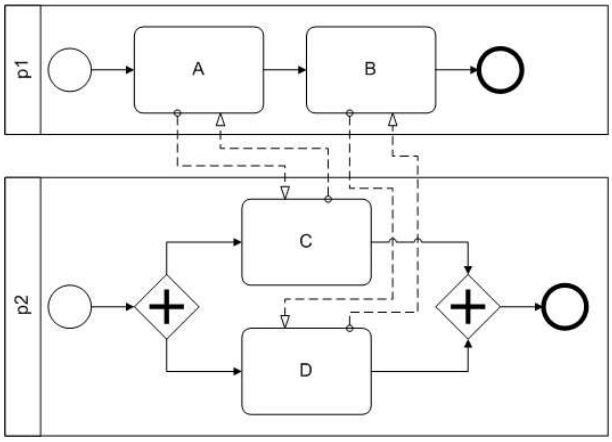

Figure 1: A trivial business collaboration

processes can be applied to the refinement of business process diagrams, and hence demonstrate how to specify both timed and untimed behavourial properties using BPMN. Moreover, our definition of the semantics allows automatic verifcation by the use of a model checker such as FDR [3]. Our semantic construction starts from syntax expressed in Z [13], following our previous work on untimed semantics [11].

This paper begins with an introduction to BPMN and the mathematical notations, Z [13] and CSP [10], that are used throughout the document. Our contribution starts in Section 3, with a Z model of BPMN syntax, and continues in Sections 4 and 5 with a timed behavioural semantics in CSP. In Section 6 we show some initial results relating the timed and untimed model and we then introduce the notion of *timed* and *untimed compatibility* between participants in a business process collaboration. We conclude this paper with a summary.

2 Notation

2.1 BPMN

States in our subset of BPMN [8] 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 state labelled *task**, *bpmn**, *task*** and *bpmn***, represents an occurrence of error within the state. A sequence of flows represents a specific control flow instance of the business process, while message flows represent directional communication between states between different *business process participants* in a collaboration.

In the figure, there are two types of start states *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

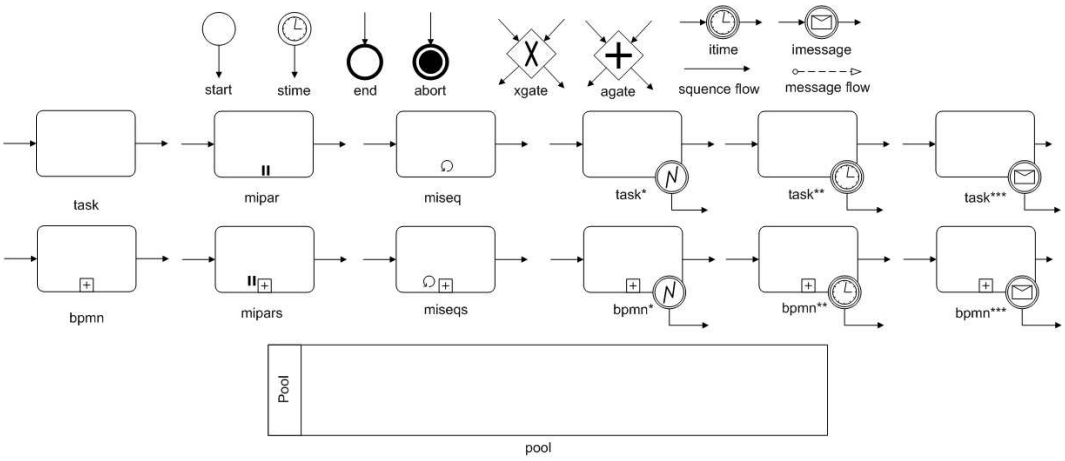

Figure 2: States of BPMN diagram

transition when a specified duration has elapsed. There is also two types of intermediate states *itime* and *imessage*. An *itime* state is a delay event and 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 and after its incoming transition is triggered, the message event waits until a specified message has arrived before initiating its outgoing transition. Both types of states have a maximum of one incoming transition and one outgoing transition

There are two types of end states *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 and its 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 it has a exactly one incoming and one outgoing transitions. 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]. The state labelled *bpmn* in Figure 2 depicts a collapsed subprocess state where all internal details are hidden. this state has a exactly one incoming and one outgoing transitions.

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 it models 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 it models an interruption upon receiving the specified message. Each task and subprocess states can have a maximum of one timed exception flow, although it may have multiples of 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 a single business processe; only one process instance is allowed at any one time. each business process contained in a pool is also a participant within a business collaboration involving multiple business processes. Each pool forms a container for some business processes; only one process instance is allowed at any one time. While *sequence flows* are restricted to an individual pool, *message flows* represent communications between pools. An illustration of message flow between activities across pools is shown in Figure 3. In the figure, task *A sends*

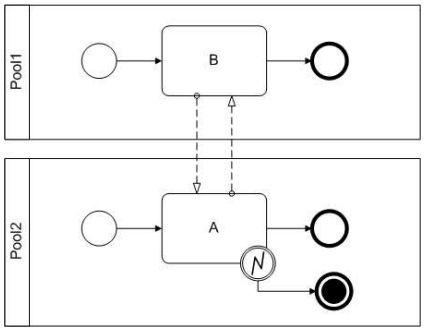

Figure 3: Interaction via message flows

a message; this is *received* by task B, which triggers the start of its activity. As task B completes the necessary activity for A it *replies* with a message for A to *accept*; such message might *break* A's activity flow. Note that each task in the figure is contained in a separate pool.

2.2 Z

The Z notation [13] 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
declaration
constraint

or equivalently

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

If S is a schema then θS denotes the characteristic binding of S in which each component is associated with its current value. Schemas can be used as declarations. For example, the lambda expression $\lambda S \bullet t$ denotes a function from the schema type underlying S, a set of bindings, to the type of term expression t.

The mathematical language within Z provides a syntax for set expressions, predicates and definitions. Types can either be basic types, maximal sets within the specification, each defined by simply declaring its name, or be free types, introduced by identifying each of the distinct members, introducing each element by name. An alternative way to define an object within an specification is by abbreviation, exhibiting an existing object and stating that the two are the same.

$$Type ::= element_1 \mid ... \mid element_n \qquad [Type] \qquad symbol == term$$

By using an axiomatic definition we can introduce a new symbol x, an element of S, satisfying predicate p.

$x : S$
p

2.3 CSP

In CSP [10], 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. Below is the syntax of the language of CSP.

$$\begin{aligned} P, Q ::= {} & P \mid\mid\mid Q \mid P \mid[A]\mid Q \mid P \mid[A \mid B]\mid Q \mid P \setminus A \mid P \triangle Q \mid \\ & P \Box Q \mid P \sqcap Q \mid P \fatsemi Q \mid e \rightarrow P \mid Skip \mid Stop \end{aligned}$$

$$e \quad ::= x \mid x.e$$

Process $P \mid\mid\mid Q$ denotes the interleaved parallel composition of processes P and Q. Process $P \mid[A]\mid Q$ denotes the partial interleaving of processes P and Q sharing events in set A. Process $P \mid[A \mid B]\mid Q$ 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. We write $\Big|\Big|\Big| i : I \bullet P(i)$, $\Big|\Big|[A]\, i : I \bullet P(i)$ and $\Big|\Big| i : I \bullet A(i) \circ P(i)$

to denote an indexed interleaving, partial interleaving and parallel combination of processes $P(i)$ for i ranging over I.

Process $P \setminus A$ is obtained by hiding all occurrences of events in set A from the environment of P. Process $P \triangle Q$ denotes a process initially behaving as P, but which may be interrupted by Q. Process $P \Box Q$ denotes the external choice between processes P and Q; the process is ready to behave as either P or Q. An external choice over a set of indexed processes is written $\Box\, i : I \bullet P(i)$. 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. Similarly an internal choice over a set of indexed processes is written $\sqcap\, i : I \bullet P(i)$.

Process $P \fatsemi Q$ denotes a process ready to behave as P; after P has successfully terminated, the process is ready to behave as Q. 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 [10]. Notable is the semantic equivalence of processes $P \Box Q$ 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 $\mathit{refusals}(P)$ is P's initial refusals. A failure therefore is a pair (s, X) where $s \in \mathit{traces}(P)$ is a trace of P leading to a stable state and $X \in \mathit{refusals}(P/s)$ where P/s represents process P after the trace s. We write $\mathit{traces}(P)$ and $\mathit{failures}(P)$ 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.

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

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

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

and CSP processes are stable-failures equivalent if they refine each other.

$$\_ \equiv_{\mathcal{F}} \_ : (\mathit{CSP} \leftrightarrow \mathit{CSP})$$
$$\forall\, P, Q : \mathit{CSP} \bullet$$
$$P \equiv_{\mathcal{F}} Q \Leftrightarrow P \sqsubseteq_{\mathcal{F}} Q \land Q \sqsubseteq_{\mathcal{F}} P$$

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 [3], 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.

3 Abstract Syntax of BPMN

In this section we describe the abstract syntax of BPMN using Z schemas and the set theory

We intially define the following basic types:

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

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

$$InMsg, OutMsg, EndMsg, LastMsg : \mathbb{P}\, Message$$
$$BName, PLName : \mathbb{P}\, PName$$

$$\langle InMsg, OutMsg, EndMsg, LastMsg\rangle \text{ partition } Message$$
$$\langle BName, PLName\rangle \text{ partition } PName$$

where *BName* represents the set of names for identifying each subprocess state; *PLName* represents the set of names for identifying each participant pool. The type *InMsg* represents a set of *incoming messages*, for example in Figure 3 task *A* receives an incoming message before performing its activities. The type *OutMsg* represents a set of *outgoing messages*, in Figure 3 task *A* sents an outgoing message after performing its activities. The types *LastMsg* and *EngMsg* are sets of messages for signalling while a multiple instance state performs its activities, their semantics will be described in details in Section 4.3.

Each type of state shown in Figure 2 is defined syntactically as follows:

$$\begin{aligned} Type ::= {} & agate \mid xgate \mid start \mid end\langle\!\langle \mathbb{N}\rangle\!\rangle \mid abort\langle\!\langle \mathbb{N}\rangle\!\rangle \mid task\langle\!\langle Task\rangle\!\rangle \mid \\ & stime\langle\!\langle Time\rangle\!\rangle \mid itime\langle\!\langle Time\rangle\!\rangle \mid ierror \mid imessage\langle\!\langle Message\rangle\!\rangle \mid \\ & bpmn\langle\!\langle BName\rangle\!\rangle \mid pool\langle\!\langle PLName\rangle\!\rangle \mid miseq\langle\!\langle Task \times \mathbb{N}\rangle\!\rangle \mid \\ & miseqs\langle\!\langle BName \times \mathbb{N}\rangle\!\rangle \mid mipar\langle\!\langle Task \times \mathbb{N}\rangle\!\rangle \mid mipars\langle\!\langle BName \times \mathbb{N}\rangle\!\rangle \end{aligned}$$

According to the official specification [8], each BPMN state type has associated attributes describing its properties; our syntactic definition has included 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*. We define abbreviations *Inputs*, *NoEnds*, *Starts*, *Tasks*, *Subs* and *Mults* as follows to assist our specification.

$$Inputs == InMsg \cup EndMsg \cup LastMsg$$
$$NoEnds == InMsg \cup LastMsg$$
$$Starts == \{\, start \,\} \cup \operatorname{ran} stime$$
$$Tasks == \operatorname{ran} task \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar$$
$$Subs == \operatorname{ran} bpmn \cup \operatorname{ran} miseqs \cup \operatorname{ran} mipars$$
$$Mults == \operatorname{ran} miseqs \cup \operatorname{ran} mipars \cup \operatorname{ran} miseq \cup \operatorname{ran} mipar$$

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 by the following schema definition,

$$Transition \mathrel{\widehat=} [guard : Guard;\ line : Line]$$

and we give the type of a message flow by the following schema definition.

$$Messageflow \mathrel{\widehat=} [message : Message;\ channel : Channel]$$

If the sequence flow has no guard or the message flow contains an empty message, then the schemas *Transition* and *Messageflow* record the default values tt and *empty* for *guard* and *message* respectively.

$$tt : Guard;\ empty : Message$$

In this paper we will only consider the semantics of BPMN timed events describing time cycles (duration) and not absolute time stamp. 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* [14]

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

We write *zero* to denote zero duration. We write $a_1 \mathinner{..} a_n \rightsquigarrow x$ inside some schema binding s to specify the components $s.a_1 \mathinner{..} s.a_n$, each of same type, to the value x.

$$zero_T == \langle\!| year, month, day, hour, minute, second \rightsquigarrow 0 |\!\rangle$$

Here we define some preliminary functions over *Time*. The function *sec* returns the duration of a given duration in seconds. Note we assume 1 month is 30 days and 1 year is 356 days.

$$sec : Time \to \mathbb{N}$$

$$\begin{aligned} sec = \lambda\, Time \bullet \\ \quad year * 31556926 + month * 2629744 \\ \quad + day * 86400 + hour * 3600 + minute * 60 + second \end{aligned}$$

We also declare some binary relations over *Time*. The following axiomatic definition also includes the definition of the relation $=_T$.

$$\_ =_T \_, \_ \geq_T \_, \_ >_T \_, \_ <_T \_, \_ \leq_T \_, \_ \neq_T \_ : Time \leftrightarrow Time$$

$$\forall\, s, t : Time \bullet s =_T t \Leftrightarrow sec\, s = sec\, t$$

Inequality operators $\geq_T$, $>_T$, $<_T$, $\leq_T$ and $\neq_T$ over duration may be defined similarly.

Each *state* records the type of its content, the sets of incoming, outgoing and error transitions and in the case of a subprocess state, a set of number-transition pairs to align the outgoing transitions of the subprocess within the outgoing transitions within the subprocess. There are also the five sets of message flows; their informal meanings are illustrated by the simple BPMN diagram

in Figure 3. Each state also incorporates the variable *loopMax* to limit the number of state instances each process instance can invoke; the schema *State* records the default value 0 if there is no limit to the number of state instances. The schema component *link* pairs each incoming message flow which initialises or interrupts the execution of the state with either an incoming transition or an exception flow; the component *depend* pairs each incoming message flow which initialise the state's execution with its corresponding outgoing message flow.

$$\begin{aligned} State \mathrel{\widehat=} [&type : Type;\ in, out : \mathbb{P}\ Transition;\ error : \mathbb{P}(Type \times Transition);\\ &send, receive, reply, accept, break : \mathbb{P}\ Messageflow;\\ &link : \mathbb{P}(Transition \times Messageflow);\\ &depend : \mathbb{P}(Messageflow \times Messageflow);\\ &exit : \mathbb{P}(\mathbb{N} \times Transition);\ ran : Range;\ loopMax : \mathbb{N}] \end{aligned}$$

The schema component *ran* is declared with the type *Range*, which is a schema recording a range of durations and is defined as follows.

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

Given some value of the schema component *ran* in some task or multiple instances task states, we say that state takes a non-deterministic duration over the range $ran.min \mathinner{..} ran.max$.

The schema *WFS* describes a subset of *well-formed* states in BPMN.

WFS

State

$$\begin{aligned} &(\exists\, m : \mathbb{N} \bullet \mathbb{N} \bullet m = \#(\bigcup\{\, send, receive, reply, accept, break \,\}) + \#link + \#depend\\ &\land\ type \in \operatorname{ran} pool \Leftrightarrow \#(\bigcup\{\, in, out, error \,\}) = m = 0\\ &\land\ type \notin Tasks \cup Subs \Rightarrow (loopMax = m = \#error = 0))\\ &type \in Tasks \Leftrightarrow ran \neq \langle\!| min, max \rightsquigarrow zero_T |\!\rangle\\ &type \in Starts \Leftrightarrow in = \emptyset \land \#out = 1\\ &type \in (\operatorname{ran} end \cup \operatorname{ran} abort) \Leftrightarrow \#in = 1 \land out = \emptyset\\ &type \notin Subs \Leftrightarrow exit = \emptyset\\ &type \in Subs \Leftrightarrow \{\, e : exit \bullet second\ e \,\} = out\\ &type \in Subs \Rightarrow send \cup accept = \emptyset\\ &(\#receive = \#reply \land \#send = \#accept)\\ &(\{\, e : link \bullet second(e) \,\} = (receive \cup break)) \land (\{\, e : link \bullet first(e) \,\} \subseteq (in \cup error))\\ &(\{\, e : depend \bullet first(e) \,\} = receive) \land (\{\, e : depend \bullet second(e) \,\} = reply)\\ &\forall\, t1, t2 : out \cup in \bullet t1.line \neq t2.line\\ &\forall\, ms : send;\ ns : reply \bullet (ms.message \in InMsg \land ns.message \in OutMsg)\\ &\forall\, ms : receive \cup accept;\ e : error \bullet (ms.message = empty \land e.guard = tt) \end{aligned}$$

We write *WCF* to denote the set of *well-configured* sets of well-formed states in some BPMN diagram. The definition of *well-configuration* can be found in our earlier paper [11]. We define well-formedness for BPMN diagrams/processes.

Definition 3.1 ***Well-formedness*** *A BPMN diagram is* well-formed *if all its constituent states forms a well-configured set of well-formed states and that all the diagram's subprocess states are also well-formed.*

Each BPMN diagram encapsulated by a *pool* represents an individual participant in a collaboration, built up from a well-configured finite set of well-formed

states. We do not allow local states to have type *pool*, since this represents a boundary of a business domain. *Local* is set of all possible specifcation environments; an environment maps each name of a BPMN diagram to its associated diagram, and each diagram is defined by a set of well-configured set of well-formed state.

$$BPD ::= states\langle\langle WCF\rangle\rangle$$
$$Local == PName \pfun BPD$$

A collaboration is then built up from a finite set of names, each of the names is associated with a BPMN diagram and *Global* is set of all possible global specifcation environments and maps each collaboration name to its associated diagram's name.

$$Chor ::= bpmns\langle\langle \mathbb{F}\, PLName\rangle\rangle$$
$$Global == CName \pfun Chor$$

4 A Relative Timed Semantics

To introduce timing information into the semantics of BPMN, we have introduced the following definitions about BPMN states and diagrams:

Definition 4.1 ***Time Active*** *As well as timed events,* stime *and* itime*, all atomic tasks are also* timed*, some strictly positive amount of time must elapse before they terminate. This ensures that each BPMN process that contains tasks must take a positive amount of time to execute.*

We define the schema *TimeState* as a subtype of *State* to capture all timed states.

$$TimeState \mathrel{\widehat=} [State \mid type \in Tasks \cup Subs \cup \operatorname{ran} stime \cup \operatorname{ran} itime]$$

Time Active also implies all other BPMN states such as untimed event and decision gateway states have zero duration. This is particularly important when defining the semantics for parallel split and join states, that is parallel gateways with either multiple outgoing transition or multiple incoming transition respectively. Essentially we would like to ensure a uniform rate of passage of time across all parallel flows.

Definition 4.2 ***Instantaneous Choices*** *Each exclusive choice is resolved instananeously and hence has zero duration, this is notion is compatible with the fact that exclusive gateway states are untimed.*

Definition 4.3 ***Time Stability*** *A BPMN process is time-stable if all its* active *states are timed. A BPMN state is active if its incoming transitions have been triggered and is waiting to engage with the environment.*

Definition 4.4 ***Time Readiness*** *Given a BPMN process is time-stable, there exists a set of timed states that are active, out of those, there are a subset of it which have the shortest delay or its delay range has the shortest lower bound and states from this subset are* time-ready*.*

Definition 4.5 ***Finite Speed*** *No BPMN process can execute an infinite sequence of untimed states in a finite amount of time. A BPMN process is* well-timed *if it satisfies this property.*

Finite Speed ensures it is always possible for a well-timed BPMN process to reach *time stability.*

Definition 4.6 ***Properly-Timed*** *A BPMN process is* properly-timed *iff the following holds for all its timed states:*

- *If the timed state is an atomic task state (task) or a parallel multiple intance state (mipar) then its timed exception flow's expiration can be no longer than the maximum delay of the state.*
- *If the timed state is an atomic sequential multiple intance state (miseq) then its timed exception flow's expiration can be no longer than the summation of maximum delay of its instances.*
- *If the timed state is a subprocess state (bpmn) or a parallel multiple instance subprocess state (mipars) then its timed exception flow's expiration can be no longer than the maximum delay of its longest execution path, an execution path is a sequence of states that begins with the start state and ends with an end state, and it represents the order in which the states contained in a process/subprocess are enacted.*
- *If the timed state is a sequential multiple instance subprocess state (miseqs) then its timed exception flow's expiration can be no longer than the summation of maximum delay of the longest execution path of all its subprocess instances.*

This definition ensures no timed exception within a BPMN process is redundant.

Similar to our definition of BPMN's untimed semantics, our timed semantic function takes a syntactic description of a BPMN diagram and returns a CSP process that models the timed behaviour of that diagram. For each participant in a business collaboration, we define its relative-timed semantics to be the partially interleaving of two processes defined by an enactment and a coordination functions:

- The enactment function returns the parallel composition of processes, each correspond to the untimed aspect of a state of the BPMN diagram defining the participant.
- The coordination function returns a single process for coordinating that diagram's timed behaviour, it essentially implements the variant of two-phase functioning approach adopted by timed coordination languages like Linda [6].

We describe the semantics in two sections, in this section we detail the definition of the enactment function, and in Section 5 we detail the definition of the coordination function, and the semantic function for individual participants and their collaboration.

The rest of this section is as follows: We define functions to associate each transition, state and diagram with their set of events in Section 4.1, Section 4.2

presents the enactment functions for mapping each BPMN diagram to its process describing its behaviour; in Section 4.3 we define the functions for mapping each type of multiple instance states to its process describing its behaviour; in Section 4.4 we present the CSP processes corresponding to the behaviour of each gateways; in Section 4.5 we define processes corresponding to the behaviour of each state types and transitions, and the general functions for mapping each BPMN states to its CSP process describing its behaviour.

4.1 Alphabets

First we define the basic types *Process* and *Event* which correspond to CSP processes and events. We define the basic type *Data* to represent the data which are communicated along CSP channels. The basic type *Channel* in this paper also denotes the set of CSP channels, hence a data object d communicated along a channel c is denoted by the compound event $c.d$.

$$[Process, Event, Data]$$

We define the partial injective function ϵ_{trans} which maps each transition to a pair of a CSP event and a guard. We insist that each transition maps to a unique CSP event. The functions ϵ_{task} and ϵ_{pname} map each task and process name to a unique event respectively. The function ϵ_{msg} maps each message flow to its set of events. The notation $\{| c_1 \ldots c_n |\}$ forms the appropriate set of events from channels $c_1 \ldots c_n$, so $\{| c |\}$ where c communicates data object of type D forms the set $\{ d : D \bullet c.d \}$.

$$
\begin{array}{|l}
\epsilon_{line} : Line \rightarrowtail\!\!\!\!\!\!\;\!+\; Event \\
\epsilon_{task} : Task \rightarrowtail\!\!\!\!\!\!\;\!+\; Event \\
\epsilon_{pname} : PName \rightarrowtail\!\!\!\!\!\!\;\!+\; Event \\
\epsilon_{mg} : Message \rightarrowtail\!\!\!\!\!\!\;\!+\; Data \\
\epsilon_{trans} : Transition \rightarrowtail\!\!\!\!\!\!\;\!+\; (Event \times Guard) \\
\epsilon_{msg} : Messageflow \rightarrowtail\!\!\!\!\!\!\;\!+\; (Channel \times Data) \\
\hline
\mathsf{disjoint}\ \langle \operatorname{ran} \epsilon_{pname}, \operatorname{ran} \epsilon_{task}, \operatorname{ran} \epsilon_{line} \rangle \\
\epsilon_{trans} = \lambda\, Transition \bullet (\epsilon_{line}\ line, guard) \\
\epsilon_{msg} = \lambda\, Messageflow \bullet (channel, \epsilon_{mg}\ message) \\
\forall\, t1, t2 : Transition \bullet (\epsilon_{tran}\ t1).1 = (\epsilon_{tran}\ t2).1 \Leftrightarrow t1 = t2 \\
\bigcup \{ (m, n) : \operatorname{ran} \epsilon_{msg} \bullet \{| m |\} \} \cap (\operatorname{ran} \epsilon_{task} \cup \operatorname{ran} \epsilon_{line}) = \emptyset
\end{array}
$$

In order to define the alphabet for each state, corresponding to the events on which each state must synchronise, we must consider the events associated with each transition, type and messageflow. We define the functions α_{trans} and α_{msg} which map each set of transitions and message flows to the set of associated events respectively. We also define the function α_{chn} which map each set of message flows to its corresponding channels.

$$
\begin{array}{|l}
\alpha_{trans} : \mathbb{P}\, Transition \rightarrowtail \mathbb{P}\, Event \\
\alpha_{msg} : \mathbb{P}\, Messageflow \rightarrowtail \mathbb{P}\, Event \\
\alpha_{chn} : \mathbb{P}\, Messageflow \rightarrowtail \mathbb{P}\, Channel \\
\hline
\forall\, mf : \mathbb{P}\, Messageflow;\ ts : \mathbb{P}\, Transition \bullet \\
\quad \alpha_{trans}\ ts = \{ cp : \epsilon_{trans} (\!|\ ts\ |\!) \bullet cp.1 \} \\
\quad \wedge\ \alpha_{msg}\ mf = \bigcup \{ cd : (\epsilon_{msg} (\!|\ mf\ |\!)) \bullet \{| cd.1 |\} \} \\
\quad \wedge\ \alpha_{chn}\ mf = \{ cd : \epsilon_{msg} (\!|\ mf\ |\!) \bullet cd.1 \}
\end{array}
$$

The alphabet of a given state is the set of events associated with a state with which it must synchronise. A state's alphabet is the union of the events mapped from all the incoming and outgoing transitions, type, exception and message flows. We define α_{state} to be a function mapping each state into its alphabet.

$$
\begin{array}{|l}
\alpha_{state} : State \pfun Local \pfun \mathbb{P}\, Event \\
\hline
\alpha_{state} = (\lambda\, State \bullet (\lambda\, l : Local \bullet \\
\textbf{if } (type \in (Tasks \cup Subs)) \\
\textbf{then } ((\textbf{if } (type \in \operatorname{ran} mipar \cup \operatorname{ran} mipars) \textbf{ then } \bigcup \{\, (t, u) : mipartst\ s \bullet \alpha_{trans}\, \{\, t, u \,\} \,\} \\
\quad \textbf{else } (\textbf{if } (type \in \operatorname{ran} miseq \cup \operatorname{ran} miseqs) \textbf{ then } \alpha_{trans}\, \{\, (miseqtst\ s).1, (miseqtst\ s).2 \,\} \textbf{ else } \emptyset)) \\
\quad \cup (\textbf{if } (type \in Subs) \textbf{ then } \bigcup \{\, s : State \mid s \in states^{\sim}(l(bpmn^{\sim} type)) \bullet \alpha_{state}\, s\, l \} \\
\qquad \textbf{else } (\textbf{if } (type \in Tasks) \textbf{ then } \{\, \epsilon_{task}(task^{\sim} type) \,\} \textbf{ else } \emptyset)) \\
\quad \cup \alpha_{trans}(out \cup in \cup error) \cup \alpha_{msg}(send \cup receive \cup reply \cup accept \cup break)) \\
\textbf{else } (\textbf{if } (type \notin \operatorname{ran} pool) \textbf{ then } \alpha_{trans}(out \cup in) \\
\quad \textbf{else } \bigcup \{\, s : State \mid s \in states^{\sim}(l(pool^{\sim} type)) \bullet \alpha_{state}\ s\ l \,\})))
\end{array}
$$

We also define the function $\alpha_{process}$ to map each diagram to the set of all possible events performed by the process describing an individual l diagram's behaviour.

$$
\begin{array}{|l}
\alpha_{process} : PName \pfun Local \pfun \mathbb{P}\, Event \\
\hline
\forall\, p : PName;\ l : Local \bullet \alpha_{process} = \bigcup \{\, s : states^{\sim}(l\, p) \bullet \alpha_{state}\ s\ l \,\}
\end{array}
$$

4.2 Processes corresponding to Enactment

Our semantics abstracts the internal flow of individual task states and only models the sequence of task initialisations and terminations within a business process. Our enactment function $bsem_T$ takes a syntactic description of a BPMN diagram encapsulated by a state of type *pool* or a BPMN subprocess and returns a parallel composition of processes, each corresponding to one of the diagram's or process's states. The parallel composition, defined by the function bsm, is conjoined via partial interleaving with process X to ensure that the business process either terminates successfully or deadlocks because of an exception flow. We define compound events $fin.i$ and $abt.i$ where i ranges over $\mathbb{N}$ to denote the successful completion and the abortion of a business process.

$$
\begin{array}{|l}
bsem_T : PName \pfun Local \pfun Process \\
\hline
\forall\, p : PName;\ l : Local \bullet \\
\quad bsem_T\ p\ l = \\
\qquad \textbf{let } \ AE = \alpha_{process}\ p\ l \cup \{\, a : \epsilon_{abort}\ p\ l;\ e : \epsilon_{end}\ p\ l \bullet fin.e, abt.a \,\} \\
\qquad\quad X = \Box\, i : \alpha_{process}\ p\ l \bullet \\
\qquad\qquad (i \rightarrow X \,\Box\, (\Box\, e : \epsilon_{abort}\ p\ l \bullet abt.e \rightarrow Stop) \\
\qquad\qquad\ \Box\, (\Box\, e : \epsilon_{end}\ p\ l \bullet fin.e \rightarrow Skip)) \\
\qquad \textbf{in } \ (bsm\ p\ l\ [\![\, AE \,]\!]\ X)
\end{array}
$$

$$
\begin{array}{l}
bsm : PName \pfun Local \pfun Process \\
\hline
\forall\, p : PName;\ l : Local \bullet \\
bsm\ p\ l = (\ \Vert\ s : \{\, s : (states^{\sim}(l\ p)) \mid s.type \notin Starts \,\} \bullet \\
\quad (\alpha_{state}\ s\ l \cup \{\, i : \epsilon_{end}\ p\ l \bullet fin.i \,\} \cup \\
\quad (\textbf{if}\ (s.type \in \operatorname{ran} abort)\ \textbf{then}\ \{\, abt.(abort^{\sim} s.type) \,\}\ \textbf{else}\ \emptyset)) \circ \\
\quad \textbf{if}\ (s.type \in \operatorname{ran} end) \\
\quad \textbf{then}\ ((\rho_{state}\ s \fatsemi fin.(end^{\sim} s.type) \rightarrow Skip) \\
\qquad \Box\ (\Box\ e : \epsilon_{end}\ p\ l \setminus \{\, end^{\sim} s.type \,\} \bullet fin.e \rightarrow Skip)) \\
\quad \textbf{else if}\ (s.type \in \operatorname{ran} abort) \\
\qquad \textbf{then}\ ((\rho_{state}\ s \fatsemi abt.(abort^{\sim} s.type) \rightarrow Stop) \Box\ \rho_{end}\ p\ l) \\
\qquad \textbf{else let}\ X = ((\rho_{state}\ s \fatsemi X) \Box\ \rho_{end}\ p\ l) \\
\qquad\quad \textbf{in}\ \ (\textbf{if}\ (s.loopMax = 0)\ \textbf{then}\ X \\
\qquad\qquad \textbf{else}\ (X \lbag\!\lbag \alpha_{msgtype}\ s.receive\ NoEnds \\
\qquad\qquad\qquad \cup\, \alpha_{trans}\ s.in \cup \{\, i : \epsilon_{end}\ p\ l \bullet fin.i \,\} \rbag\!\rbag \\
\qquad\qquad\quad \rho_{loop}\ p\ s\ l)))) \\
\quad \lbag\!\lbag\, \alpha_{start}\ p\ l \cup \{\, i : \epsilon_{end}\ p\ l \bullet fin.i \,\} \rbag\!\rbag \\
\quad \Box\ s : \{\, s : states^{\sim}(l\ p) \mid s.type = start \,\} \bullet (\rho_{state}\ s \fatsemi \rho_{end}\ p\ l))
\end{array}
$$

We observe that the processes corresponding to a start, an end or an abort state are the only non-recursive processes; a start, an end or an abort activity can occur only once, while it is possible for all other states to occur many times within a single process instance. The function ϵ_{end} returns the set of numbers defined by each of the *end* states within the diagram's syntax, while ϵ_{abort} returns the set of numbers defined by each of the *abort* states. We apply external choice over the processes corresponding to states with a terminating process synchronising on all *end* states. This ensures that all processes terminate at the end of the business process execution. The function α_{start} returns the set of events corresponding to all outgoing transitions of all *start* and *stime* states within the diagram's syntax.

$$
\begin{array}{l}
\alpha_{start} : PName \pfun Local \pfun \mathbb{P}\, Event \\
\epsilon_{end} : PName \pfun Local \pfun \mathbb{P}\, \mathbb{N} \\
\rho_{end} : PName \pfun Local \pfun Process \\
\epsilon_{abort} : PName \pfun Local \pfun \mathbb{P}\, \mathbb{N} \\
\hline
\forall\, p : PName;\ l : Local \bullet \\
\quad \alpha_{start}\ p\ l = \bigcup \{\, s : states^{\sim}(l\ p) \mid s.type \in Starts \bullet \alpha_{trans}(s.out) \,\} \\
\quad \land\ \epsilon_{end}\ p\ l = \{\, s : states^{\sim}(l\ p) \mid s.type \in \operatorname{ran} end \bullet end^{\sim} s.type \,\} \\
\quad \land\ \rho_{end}\ p\ l = (\Box\ e : \epsilon_{end}\ p\ l \bullet fin.e \rightarrow Skip)) \\
\quad \land\ \epsilon_{abort}\ p\ l = \\
\qquad \{\, s : states^{\sim}(l\ p) \mid s.type \in \operatorname{ran} abort \bullet abort^{\sim} s.type \,\} \\
\qquad \cup \bigcup \{\, s : states^{\sim}(l\ p) \mid s.type \in \operatorname{ran} bpmn \bullet \\
\qquad\qquad \epsilon_{abort}\ (bpmn^{\sim} s.type)\ l \,\}
\end{array}
$$

The function ρ_{loop} maps each state of type *task* and *bpmn* to a process which limits the number of iterations of the state.

$$\rho_{loop} : PName \nrightarrow State \nrightarrow Local \nrightarrow Process$$

$$\begin{array}{l}
\forall\, p : PName;\ s : State;\ l : Local \bullet \\
\rho_{loop}\ p\ s\ l = \\
\quad \textbf{let}\ Y = \Box\, i : \alpha_{trans}\ s.in \bullet i \rightarrow Skip \\
\qquad M = \rho_{extmsg}\ s.in\ NoEnds \\
\qquad X(n) = n > 0\ \&\ (Y \fatsemi X(n-1) \Box (M \fatsemi Y \fatsemi X(n-1)) \Box \rho_{end}\ p\ l) \\
\qquad\qquad \Box\ n \leq 0\ \&\ \rho_{end}\ p\ l \\
\quad \textbf{in}\ X(loopMax)
\end{array}$$

4.3 Processes corresponding to Multiple Instances

We define the function ρ_{mipar} which return the process corresponding to the behaviour of the state of type *mipar* or *mipars*.

$$\rho_{mipar} : State \nrightarrow Local \nrightarrow Process$$

$$\begin{array}{l}
\forall\, s : State;\ l : Local \bullet \\
\quad (\exists_1\ ts : \mathbb{P}(Transition \times Transition) \bullet ts = mipartst\ s) \Rightarrow \\
\rho_{mipar}\ s\ l = \\
\qquad \textbf{let} \\
\qquad\quad miparSet = \bigcup\{\, \{\, i : \alpha_{chn} \bigcup\{\, s.send, s.break, s.receive \,\};\ j : NoEnds \bullet i.j \,\}, \\
\qquad\qquad\qquad \alpha_{trans}\ (s.out \cup s.error), \{\, (i,j) : TP \bullet i, j \,\}\, \} \\
\qquad\quad TP = \{\, (i,j) : ts \bullet ((\epsilon_{trans}\ i).1, (\epsilon_{trans}\ j).1) \,\} \\
\qquad\quad Con = s.receive = \emptyset\ \&\ (\rho_{intermsg}\ s.send\ EndMsg \fatsemi XS(s.out)) \\
\qquad\qquad\quad \Box\ (Cn(TP, s) \triangle (\textbf{if}\ s.break = \emptyset\ \textbf{then}\ AJ(s.error)\ \textbf{else}\ \rho_{mierror}\ s)) \\
\qquad \textbf{in} \\
\qquad\quad XJ(s.in) \fatsemi (MTask(ts, s, l) \,[\![\, miparSet \,]\!]\, Con)
\end{array}$$

The function ρ_{mipar} is constructed by partially interleaving the control process *Con* and the process *MTask*, where *MTask* is the partial interleaving of n copies of processes each corresponding to an instance of a task or a subprocess specified by the constructor function. Each copy of the processes in *MTask* is synchronised on the outgoing transitions of the multiple instance state.

$$\begin{array}{l}
MTask(ts, s, l) = \\
\quad \|\ [\alpha_{trans}\ s.out]\ (i,j) : ts \bullet \\
\qquad (\rho_{depend}\ s\ [\![\, \alpha_{msg}(s.receive \cup s.reply)]\!] \\
\qquad (((\rho_{state}\ (\!|in \rightsquigarrow \{\, i \,\}, type \rightsquigarrow s.type, out \rightsquigarrow \{\, j \,\}, exit \rightsquigarrow s.exit, send \rightsquigarrow s.send, \\
\qquad\qquad ran \rightsquigarrow s.ran, error \rightsquigarrow s.error, reply \rightsquigarrow s.reply, accept \rightsquigarrow s.accept, \\
\qquad\qquad break \rightsquigarrow s.break, receive, link, depend \rightsquigarrow \emptyset |\!)\ l) \fatsemi XS(s.out)) \Box XS(s.out)))
\end{array}$$

On receiving a trigger by one of the incoming transitions, the control process *Con* either decides not to execute any instance, if there is no message flow dependency from another state, or behaves like process *Cn*.

$$\begin{array}{l}
Cn(T, s) = \\
\quad (s.receive = \emptyset\ \&\ (\#T > 1\ \&\ IC(T, true, s) \Box \#T = 1\ \&\ EC(T, s) \Box s.send = \emptyset\ \&\ XS(s.out)) \\
\quad \Box\ (\rho_{extmsg}\ s.receive\ EngMsg \fatsemi \rho_{intermsg}\ s.send\ EndMsg \fatsemi XS(s.out)) \\
\quad \Box\ ((\rho_{extmsg}\ s.receive\ LastMsg \fatsemi EC(T, s)) \Box (\rho_{extmsg}\ s.receive\ InMsg \fatsemi IC(T, false, s))))
\end{array}$$

The process *Cn* takes the set of event-pairs, each corresponding to the incoming and outgoing transitions of an instance defined in *MTask*, and returns the

process that controls the multiple instances in *MTask*. If the multiple instance state's *receive* component is empty, then it internally controls the number of instances to trigger, otherwise it controls the number of instances according to the message received through the set of message flows specified by the component *receive*. The control process keeps a counter of the number of instances triggered.

The process *CL* takes a set of transition-pairs and a set of message flows specified in *send* and recursively sends messages of type either *InMsg* or *LastMsg* along the channels specified by the component *send*. If all of the messages are of type *LastMsg*, then *CL* triggers one of the outgoing transitions and the whole multiple instance state terminates, otherwise it behaves as the process *Cn*.

$$
\begin{aligned}
CL(T, D, s) = \textbf{if } & D = \emptyset \textbf{ then } XS(s.out) \\
& \textbf{else } (((\Box\, r : D \bullet r?x : InMsg \rightarrow Skip) \fatsemi \\
& \quad (|||\, q : ((\alpha_{chn}\, D) \setminus \{\, r \,\}) \bullet q?i : \epsilon_{mg}(\!|\, NoEnds \,|\!) \rightarrow Skip) \fatsemi Cn(T, s)) \\
& \quad \Box\, (\rho_{extmsg}\, D\, LastMsg \fatsemi CL(T, (D \setminus \{\, r \,\}), s)))
\end{aligned}
$$

The process *EC* triggers one instance of a task or subprocess, during which it sends a message of type *LastMsg* along each of the message flow channels specified in *send*. It then triggers one of the outgoing transitions. The process *IC* triggers one instance of a task or subprocess by synchronising on its incoming and outgoing transitions, during which it behaves as process *CL* to monitor the type of messages sent along each of the message flow channels specified in *send*.

$$
\begin{aligned}
EC(T, s) &= \Box(i, j) : T \bullet (i \rightarrow \rho_{intermsg}\, s.send\, LastMsg \fatsemi j \rightarrow XS(s.out)) \\
IC(T, b, s) &= \Box(i, j) : T \bullet i \rightarrow (j \rightarrow Skip \;|||\; \\
&\qquad (\textbf{if } (b \wedge s.send \neq \emptyset) \textbf{ then } CL((T \setminus \{\, (i, j) \,\}), s.send, s) \\
&\qquad \textbf{else } \rho_{intermsg}\, s.send\, InMsg \fatsemi Cn((T \setminus \{\, (i, j) \,\}), s)))
\end{aligned}
$$

The following is a set of rules which governs how the control process triggers the multiple instances process.

- The control process can trigger up to N processes, where N is a natural number specified by the constructor function argument.
- If the state schema component *receive* is empty, then the control process triggers up to N instances nondeterministically.
- If the state schema component *receive* is not empty and the message received is of type *LastMsg*, then the control process must only trigger one more instance.
- If the state schema component *send* is not empty, then during the execution of the last instance the control process must send a message of type *LastMsg* along each of the message flow channels specified in *send*.
- If the state schema component *receive* is not empty and the message received is of type *EndMsg*, then the control process must send a message of type *EndMsg* along each of the message flow channels specified in *send*, and terminate.
- After all triggered multiple instances have terminated, the whole multiple instance state terminates and triggers one of its outgoing transitions.

- On receiving an error message flow specified in the component *error*, the control process triggers an exception flow and the whole multiple instance state deadlocks.
- If the state schema component *error* is not empty, the control process can trigger an exception flow from the set *error* at any time, and the whole multiple instance state deadlocks.

We define the function $\alpha_{msgtype}$, which returns the set of events corresponding to the given message flows passing the given messages. The functions $\rho_{intermsg}$ and ρ_{extmsg} return the process corresponding to the interleaving and exclusive choice of the given set of message flows passing the given set of messages.

$$
\begin{array}{l}
\rho_{intermsg}, \rho_{extmsg} : \mathbb{P}\, Messageflow \pfun \mathbb{P}\, Message \pfun Process \\
\alpha_{msgtype} : \mathbb{P}\, Messageflow \pfun \mathbb{P}\, Message \pfun \mathbb{P}\, Event \\
\hline
\forall\, mf : \mathbb{P}\, Messageflow;\ ms : \mathbb{P}\, Message \bullet \\
\rho_{intermsg} = \Vert\, r : \alpha_{chn}\, mf \bullet r?x : ms \rightarrow Skip \\
\land\ \rho_{extmsg} = \Box\, r : \alpha_{chn}\, mf \bullet r?x : ms \rightarrow Skip \\
\land\ \alpha_{msgtype} = \{\, c : \alpha_{chn} mf;\ d : \epsilon_{mg}(\!|\ ms\ |\!) \mid c.d \in \{|c|\} \bullet c.d \,\}
\end{array}
$$

The function $\rho_{mierror}$ returns the process that synchronises with the exception flows of individual instances of the multiple instances states.

$$
\begin{array}{l}
\rho_{mierror} : State \pfun Process \\
\hline
\rho_{mierror} = (\lambda\, State \bullet \\
\quad (\Box (i,j) : \{\, (e,f) : link \mid e \in error \bullet ((\epsilon_{trans}\ e).1, (\epsilon_{msg}\ f).1) \,\} \bullet j?x : InMsg \rightarrow i \rightarrow Stop) \\
\quad \Box\ (\Box\, i : \{\, g : error \mid g \notin \{\, (e,f) : link \bullet e \,\} \bullet (\epsilon_{trans}\ g).1 \,\} \bullet i \rightarrow Stop))
\end{array}
$$

The function *mipartst* maps each state of type *mipar* or *mipars* to a set of transition pairs used to connect the state's parallel instances of task or subprocess state. The function *miseqtst* maps each state of type *miseq* or *miseqs* to a transition pair used to connect the state's task or subprocess state.

$$
\begin{array}{l}
mipartst : State \rightarrowtail \mathbb{P}(Transition \times Transition) \\
miseqtst : State \rightarrowtail (Transition \times Transition) \\
\hline
\forall\, s : State \bullet \\
\quad ((s.type \in \operatorname{ran} mipar \Rightarrow \\
\qquad mipartst\ s = \\
\qquad\quad (\mu\, ts : \mathbb{P}(Transition \times Transition) \mid \\
\qquad\qquad \mathsf{disjoint}\ \langle first(\!|\ ts\ |\!), second(\!|\ ts\ |\!)\rangle \\
\qquad\qquad\quad \land\ \#ts = (mipar^{\sim} s.type).2 = \#first(\!|\ ts\ |\!) = \#second(\!|\ ts\ |\!)) \\
\quad \land \\
\quad ((s.type \in \operatorname{ran} mipars \Rightarrow \\
\qquad mipartst\ s = \\
\qquad\quad (\mu\, ts : \mathbb{P}(Transition \times Transition) \mid \\
\qquad\qquad \mathsf{disjoint}\ \langle first(\!|\ ts\ |\!), second(\!|\ ts\ |\!)\rangle \\
\qquad\qquad\quad \land\ \#ts = (mipars^{\sim} s.type).2 = \#first(\!|\ ts\ |\!) = \#second(\!|\ ts\ |\!)) \\
\quad \land \\
\quad ((s.type \in \operatorname{ran} miseq \cup \operatorname{ran} miseqs \Rightarrow \\
\qquad miseqtst\ s = (\mu(t1,t2) : (Transition \times Transition))
\end{array}
$$

The function ρ_{miseq} returns the process corresponding to the behaviour of the state of type *miseq* or *miseqs*.

$$
\begin{array}{|l}
\rho_{miseq} : State \pfun Local \pfun Process \\
\hline
\forall\, s : State;\ l : Local \mid (\exists\, t1, t2 : Transition;\ e1, e2 : Event;\ n : \mathbb{N} \bullet \\
\quad (t1, t2) = miseqtst\ s \land (e1, e2) = ((\epsilon_{trans}\ t1).1, (\epsilon_{trans}\ t2).1) \\
\quad \land\ (\textbf{if}\ s.type \in \operran miseq\ \textbf{then}\ n = (miseq^\sim s.type).2\ \textbf{else}\ n = (miseqs^\sim s.type).2)) \bullet \\
\quad \rho_{miseq}\ s\ l = \\
\quad \textbf{let} \\
\qquad SY = \bigcup \{\, \alpha_{trans}(s.out \cup s.error), \{\, e1, e2 \,\}, \{\, i : \alpha_{chn}\ s.receive;\ j : NoEnds \bullet i.j \,\}, \\
\qquad\qquad \{\, i : \alpha_{chn}(s.send \cup s.break);\ j : InMsg \bullet i.j \,\} \,\} \\
\quad \textbf{in} \\
\qquad (Cq(n, n, s, e1, e2) \Delta\ (\textbf{if}\ s.break = \emptyset\ \textbf{then}\ AJ(s.error)\ \textbf{else}\ \rho_{mierror}\ s)) \\
\qquad [\![\, SY \,]\!]\ Seq(n, s, l)
\end{array}
$$

Similar to ρ_{mipar} the function ρ_{miseq} is constructed by partially interleaving a control process *Cq* with process *Seq*, which models the multiple instances of task or subprocess, specified by the contructor function, executing sequentially.

$$
\begin{array}{l}
Seq(i, s, l) = i > 0\ \& \\
\quad ((\rho_{state}\ (\!|\, receive \sim\!> s.receive, in \sim\!> \{\, t1 \,\}, type \sim\!> s.type, out \sim\!> \{\, t2 \,\}, send \sim\!> s.send, \\
\qquad\qquad accept \sim\!> s.accept, reply \sim\!> s.reply, error \sim\!> s.error, break \sim\!> s.break, \\
\qquad\qquad ran \sim\!> s.ran, link \sim\!> s.link, depend \sim\!> s.depend \,|\!)\ l)\ \fatsemi\ Seq(i - 1, s, l)) \Box\ XS(s.out)
\end{array}
$$

The process *Cq* is triggered initially by one of the incoming transitions of the multiple instance state. Similar to the control process of ρ_{mipar}, the interaction between the control process and the multiple instance process is governed by the same set of rules. However, whereas the control process for ρ_{mipar} triggers instances in parallel, process *Cq* triggers instances in sequence.

$$
\begin{array}{l}
Cq(n, nm, s, e, f) = \\
\quad ((XJ(s.in) \Box f \rightarrow Skip)\ \fatsemi \\
\quad ((\rho_{extmsg}\ s.receive\ EndMsg\ \fatsemi\ \rho_{intermsg}\ s.send\ EndMsg\ \fatsemi\ XS(s.out)) \\
\quad \Box\ (\rho_{extmsg}\ s.receive\ InMsg\ \fatsemi\ e \rightarrow \rho_{intermsg}\ s.send\ InMsg\ \fatsemi\ Cq(n - 1, nm, s, e, f)) \\
\quad \Box\ (\rho_{extmsg}\ s.receive\ LastMsg\ \fatsemi\ e \rightarrow \rho_{intermsg}\ s.send\ LastMsg\ \fatsemi\ f \rightarrow XS(s.out)) \\
\quad \Box\ s.receive = \emptyset\ \& \\
\qquad ((n > 1)\ \& (e \rightarrow (\textbf{if}\ s.send = \emptyset\ \textbf{then}\ Cq(n - 1, nm, s, e, f)\ \textbf{else}\ CLs(s.send, n, nm, s, e, f))) \\
\qquad \Box\ n = 1\ \& (e \rightarrow (\textbf{if}\ s.send = \emptyset\ \textbf{then}\ Skip\ \textbf{else}\ \rho_{intermsg}\ s.send\ LastMsg)\ \fatsemi\ f \rightarrow XS(s.out)) \\
\qquad \Box\ s.send = \emptyset\ \&\ XS(s.out) \\
\qquad \Box\ n = nm\ \& (\rho_{intermsg}\ s.send\ EndMsg\ \fatsemi\ XS(s.out)))))
\end{array}
$$

The process *CLs* behaves similarly to *CL* in that it recursively sends messages of type either *InMsg* or *LastMsg* along the channels specified by the component *send*. If all of the messages are of type *LastMsg* then *CLs* triggers one of the outgoing transitions and the whole multiple instance state terminate, otherwise it behaves as the process *Cq*.

$$
\begin{array}{l}
CLs(S, n, nm, s, e, f) = \\
\quad \textbf{if}\ S = \emptyset\ \textbf{then}\ f \rightarrow XS(s.out) \\
\quad \textbf{else}\ \rho_{extmsg}\ S\ InMsg)\ \fatsemi \\
\qquad (|||\ q : (S \setminus \{\, r \,\}) \bullet q?i : \epsilon_{mg}(\!|\ NoEnds\ |\!) \rightarrow Skip)\ \fatsemi\ Cq(n - 1, nm, s, e, f))) \\
\qquad \Box\ (\rho_{extmsg}\ S\ LastMsg\ \fatsemi\ CLs(S \setminus \{\, r \,\}, n, nm, s, e, f))
\end{array}
$$

4.4 Processes corresponding to Gateways and Message flows

We now define some CSP processes that correspond to the behaviour of each of the gateway states and message flows.

Exclusive Choice Gateway Processes $XS(tn)$ and $XJ(tn)$ model the behaviour of outgoing and incoming transitions of the state type *xgate*. Note although each outgoing transition of the state type *xgate* is guarded, the choice of its incoming transitions is determined by the behaviour of the preceding states.

$$\begin{aligned} XS(tn) &= \Box\, e : \epsilon_{trans}(\!|\ tn\ |\!) \bullet \ (\text{if } e.2 \text{ then } e.1 \rightarrow \mathit{Skip} \text{ else } \mathit{Skip}) \\ XJ(tn) &= \Box\, e : \alpha_{trans}\ tn \bullet e \rightarrow \mathit{Skip} \end{aligned}$$

We also define the process $AJ(tn)$ to model the behaviour of incoming transitions of the state type *abort* and exception flow within state of type *task* and *bpmn*.

$$AJ(tn) = \Box\, e : \alpha_{trans}\ tn \bullet e \rightarrow \mathit{Stop}$$

Parallel Gateway Process $ASJ(tn)$ models the behaviour of outgoing and incoming transitions of the state type *agate*. Note that all outgoing transitions are enabled and all incoming transition are required in this state type.

$$ASJ(tn) = |||\, e : \alpha_{trans}\ tn \bullet e \rightarrow \mathit{Skip}$$

We also define CSP processes that correspond to the behaviour of message flows.

Message Flow Interaction Processes $RC(ms)$, $SD(ms)$, $AC(ms)$ and $RE(ms)$ model the behaviour of a task or a subprocess *receiving*, *sending*, *accepting* and *replying* a message respectively. While it only takes an activity to receive any one of the message flows to initiate or to abort its execution and one corresponding message flow to notify about its completion, other message flows within its execution must all be completed.

$$\begin{aligned} &RC(ms) = \rho_{extmsg}\ ms\ \mathit{NoEnds} \\ &SD(ms) = |||(s,n) : \{\, (p,k) : \epsilon_{msg}(\!|\ ms\ |\!) \mid k \in \epsilon_{mg}(\!|\ \mathit{NoEnds}\ |\!)\,\} \bullet s!n \rightarrow \mathit{Skip} \\ &AC(ms) = \rho_{intermsg}\ ms\ \mathit{OutMsg} \\ &RE(ms) = \Box(s,n) : \{\, (p,k) : \epsilon_{msg}(\!|\ ms\ |\!) \mid k \in \epsilon_{mg}(\!|\ \mathit{OutMsg}\ |\!)\,\} \bullet s!n \rightarrow \mathit{Skip} \end{aligned}$$

4.5 Processes corresponding to Transitions, Types and States

Functions ρ_{out} and ρ_{in} take a state and return the process describing the behaviour of all outgoing and incoming transitions, respectively.

$$
\begin{array}{l}
\rho_{out} : State \pfun Process \\
\rho_{in} : State \pfun Process \\
\hline
\rho_{out} = (\lambda\, State \bullet \\
\quad \textbf{if } (type = asplit) \textbf{ then } ASJ(out) \textbf{ else } XS(out)) \\
\rho_{in} = (\lambda\, State \bullet \\
\quad \textbf{if } (type \in \operran abort) \textbf{ then } AJ(in) \\
\quad \textbf{else if } (type = ajoin) \textbf{ then } ASJ(in) \textbf{ else } XJ(in))
\end{array}
$$

The function ρ_{type} maps the type of a given state to its corresponding process. Since our semantics abstracts internal flow of task states, we only model the initialisation, the termination, message flows and any exception flow of each task, note exception flows do to terminate its state.

$$
\begin{array}{l}
\rho_{exit} : State \pfun Process \\
\rho_{type} : State \pfun Local \pfun Process \\
\hline
\rho_{exit} = (\lambda\, State \bullet \\
\quad \textbf{let} \quad X = (\textbf{if } reply = \emptyset \textbf{ then } Skip \textbf{ else } RE(reply)) \\
\qquad\quad Y = \{\, (e, f) : exit \bullet (fin.e, (\epsilon_{trans}\, f).1) \,\} \\
\quad \textbf{in } (\Box (i, j) : Y \bullet i \then X \fatsemi j \then Skip) \extchoice XS(error)) \\
\rho_{type} = (\lambda\, State \bullet (\lambda\, l : Local \bullet \\
\quad \textbf{if } (type \in \operran task) \\
\quad \textbf{then if } (error = \emptyset) \\
\quad\quad \textbf{then } \epsilon_{task}(task^{\sim} type) \then (SD(send) \fatsemi AC(accept) \fatsemi RE(reply)) \\
\quad\quad \textbf{else } \epsilon_{task}(task^{\sim} type) \then (((SD(send) \fatsemi AC(accept)) \\
\qquad\qquad\qquad \interrupt (\textbf{if } break = \emptyset \textbf{ then } XS(error) \\
\qquad\qquad\qquad\quad \textbf{else } (\rho_{link}\, \{\, (e, f) : link \mid e \in error \,\}\, error \\
\qquad\qquad\qquad\qquad [\![\alpha_{trans}\, in \cup \alpha_{msgtype}\, break\, NoEnds]\!] \\
\qquad\qquad\qquad\qquad RC(break) \fatsemi XS(error)))) \fatsemi RE(reply)) \\
\quad \textbf{else if } (type \notin \operran task \cup \operran bpmn) \textbf{ then } Skip \\
\quad\quad \textbf{else } (\textbf{if } (error = \emptyset) \textbf{ then } \epsilon_{pname}(bpmn^{\sim} type) \then bsem\, (bpmn^{\sim} type)\, l \\
\qquad\quad \textbf{else } \epsilon_{pname}(bpmn^{\sim} type) \then (bsem\, (bpmn^{\sim} type)\, l \\
\qquad\qquad\qquad \interrupt (\textbf{if } break = \emptyset \textbf{ then } XS(error) \textbf{ else } RC(break) \fatsemi XS(error))))))
\end{array}
$$

The function ρ_{link} returns a process which pairs each incoming message flow with its corresponding incoming transition or exception flow, according to the component *link* of the schema *State*. The function ρ_{depend} returns a process which pairs each incoming message flow with its corresponding outgoing message flow, according to the component *depend* of the schema *State*.

$$
\begin{array}{l}
\rho_{link} : \power(Transition \times Messageflow) \pfun \power\, Transition \pfun Process, \\
\rho_{depend} : State \pfun Process \\
\hline
\forall (t, m) : \power(Transition \times Messageflow);\ ts : \power\, Transition;\ s : State \bullet \\
\rho_{link}\, (t, m)\, ts = ((\Box (i, j) : \{\, (e, f) : t1 \bullet ((\epsilon_{trans}\, e).1, (\epsilon_{msg}\, f).1) \,\} \bullet \\
\qquad\qquad\qquad j?x : NoEnds \then i \then Skip) \extchoice (\Box\, i : ts \bullet i \then Skip) \\
\land\ \rho_{depend}\, s = (\Box (i, (j, k)) : \{\, (e, f) : s.depend \bullet ((\epsilon_{msg}\, e).1, \epsilon_{msg}\, f) \,\} \bullet \\
\qquad\qquad\qquad i?x : NoEnds \then j.k \then Skip) \extchoice s.depend = \emptyset \mathbin{\&} Skip
\end{array}
$$

We define the function ρ_{state} which returns the process corresponding to the behaviour of a given state; this function essentially maps each state to the

sequential composition of the processes corresponding to the state's incoming transitions, type, message flows and outgoing transitions.

$$
\begin{array}{l}
\rho_{state} : State \pfun Local \pfun Process \\
\hline
\rho_{state} = (\lambda\, State \bullet (\lambda\, l : Local \bullet \\
\textbf{if } (type \in \operatorname{ran} task) \\
\textbf{then } (\rho_{depend}\, \theta State \,[\![\, \alpha_{msgtype}\, receive\, NoEnds \cup \alpha_{msg}\, reply]\!] \\
\quad (\rho_{link}\, \{\, (e, f) : link \mid e \in in \,\}\, in \,[\![\, \alpha_{trans}\, in \cup \alpha_{msgtype}\, receive\, NoEnds]\!] \\
\quad (\rho_{in}\, \theta State \fatsemi \rho_{type}\, \theta State\, l \fatsemi \rho_{out}\, \theta State))) \\
\textbf{else if } (type \in \operatorname{ran} bpmn) \\
\quad \textbf{then } (\rho_{depend}\, \theta State \,[\![\, \alpha_{msgtype}\, receive\, NoEnds \cup \alpha_{msg}\, reply]\!] \\
\quad\quad (\rho_{link}\, \theta State \,[\![\, \alpha_{trans}\, (in \cup error) \cup \alpha_{msgtype}\, receive\, NoEnds]\!] \\
\quad\quad (\rho_{in}\, \theta State \fatsemi ((\rho_{type}\, \theta State\, l \,[\![\, \{\, e : exit \bullet fin.(e.1) \,\} \cup \alpha_{trans}\, error]\!] \\
\quad\quad \rho_{exit}\, \theta State\, l) \,[\![\, \{\, o : out \bullet (\epsilon_{trans}\, e).1 \,\} \,]\!]\, \rho_{out}\, \theta State)))) \\
\quad \textbf{else if } (type \in \operatorname{ran} miseq \cup \operatorname{ran} miseqs) \textbf{ then } \rho_{miseq}\, \theta State\, l \\
\quad\quad \textbf{else if } (type \in \operatorname{ran} mipar \cup \operatorname{ran} mipars) \textbf{ then } \rho_{mipar}\, \theta State\, l \\
\quad\quad\quad \textbf{else if } (type = start) \textbf{ then } \rho_{out}\, \theta State \\
\quad\quad\quad\quad \textbf{else if } (type \in \operatorname{ran} end \cup \operatorname{ran} abort) \textbf{ then } \rho_{in}\, \theta State \\
\quad\quad\quad\quad\quad \textbf{else } \rho_{in}\, \theta State \fatsemi \rho_{out}\, \theta State))
\end{array}
$$

5 Coordination Function

In this section we define processes corresponding to the coordination of the timed behaviour of BPMN diagrams. We first define the CSP events for representing a state "delaying", "terminating", "being cancelled" and "being interrupted". We write $g \circ f$ to denote the backward relational composition of f with g.

$$
\begin{array}{l}
\epsilon_{wait} : State \rightarrowtail Event \\
\epsilon_{fin}, \epsilon_{can} : State \rightarrow Event \\
\epsilon_{int} : State \rightarrowtail \mathbb{P}(Transition \times Event) \\
\hline
\mathsf{disjoint}\, \langle \operatorname{ran} \epsilon_{wait}, \operatorname{ran} \epsilon_{fin}, \operatorname{ran} \epsilon_{can}, (second \circ unzip) \bigcup (\operatorname{ran} \epsilon_{int}) \rangle \\
\operatorname{dom} \epsilon_{wait} = \{\, State \bullet type \in Tasks \,\} \\
\operatorname{dom} \epsilon_{int} = \{\, State \bullet type \in Tasks \cup Subs \land error \neq \emptyset \,\} \\
\forall\, s : State \bullet \epsilon_{int}\, s = \{\, (t, e) : (Transition \times Event) \mid \exists_1\, e : s.error \bullet second\, e = t \,\}
\end{array}
$$

The generic function *unzip* takes some set of pairs $\{\, (a, b) .. (a_n, b_n) \,\}$ and returns the pair of sets $(\{\, a \,..\, a_n \,\}, \{\, b \,..\, b_n \,\})$.

$$
\begin{array}{l}
[X, Y] \\
\hline
unzip : \mathbb{P}(X \times Y) \to (\mathbb{P}\, X \times \mathbb{P}\, Y) \\
\hline
unzip = (\lambda\, zp : \mathbb{P}(X \times Y) \bullet (\{\, (x, y) : zp \bullet x \,\}, \{\, (x, y) : zp \bullet y \,\}))
\end{array}
$$

The overall semantic function of each individual business participant *tsem* is defined by partially interleaving the enactment process defined by the function *bsem* with the coordination process defined by the function *clock*. The function *hide* is defined to conceal the control flow events from the environment outside the specification of the BPMN diagram ,and the function *sync* returns the set of events to be coordinated by *clock* and the function α_{clock} returns the

set of events, hidden from the enactment function, performed internally in the coordination function.

$$hide, sync : PName \pfun Local \pfun \mathbb{P}\, Event$$

$$\begin{aligned}
&\forall\, p : PName;\ l : Local \bullet \\
&\quad hide\ p\ l = \bigcup \{\, s : states^{\sim}(l\ p) \bullet \alpha_{trans}(s.in \cup s.out \cup s.error)\,\} \\
&\quad \land\ sync\ p\ l = \bigcup \{\, s : states^{\sim}(l\ p) \mid \alpha_{state}\ s \setminus mfs\ s\,\}
\end{aligned}$$

The function *mfs* takes a BPMN state and returns a set of CSP events corresponding to the state's messageflows.

$$mfs : State \pfun \mathbb{P}\, Event$$

$$\begin{aligned}
&\forall\, s : State \bullet \\
&\quad s.type \notin Subs \cup Tasks \Rightarrow mfs\ s = \emptyset \\
&\quad \land\ s.type \in \operatorname{ran} task \Rightarrow mfs\ s = \alpha_{trans}(s.receive \cup s.send \cup s.reply \cup s.accept \cup s.break) \\
&\quad \land\ s.type \in \operatorname{ran} bpmn \Rightarrow \\
&\qquad mfs\ s = \alpha_{trans}(s.receive \cup s.send \cup s.reply \cup s.accept \cup s.break) \\
&\qquad\qquad \cup \bigcup mfs\ (\!|\ states^{\sim}(l\ (bpmn^{\sim}\ s.type))\ |\!) \\
&\quad \land\ s.type \in \operatorname{ran} mipar \Rightarrow \\
&\qquad mfs\ s = \alpha_{trans}(s.receive \cup s.send \cup s.reply \cup s.accept \cup s.break) \\
&\qquad\qquad \cup \bigcup \{\, (t, u) : mipartst\ s \bullet \alpha_{trans}\ \{\, t, u\,\}\,\} \\
&\quad \land\ s.type \in \operatorname{ran} mipars \Rightarrow \\
&\qquad mfs\ s = \alpha_{trans}(s.receive \cup s.send \cup s.reply \cup s.accept \cup s.break) \\
&\qquad\qquad \cup \bigcup \{\, (t, u) : mipartst\ s \bullet \alpha_{trans}\ \{\, t, u\,\}\,\} \\
&\qquad\qquad \cup \bigcup mfs\ (\!|\ states^{\sim}(l\ ((first \circ mipars^{\sim})\ s.type))\ |\!) \\
&\quad \land\ s.type \in \operatorname{ran} miseq \Rightarrow \\
&\qquad mfs\ s = \alpha_{trans}(s.receive \cup s.send \cup s.reply \cup s.accept \cup s.break) \\
&\qquad\qquad \cup \alpha_{trans}\ \{\, (miseqtst\ s).1, (miseqtst\ s).2\,\} \\
&\quad \land\ s.type \in \operatorname{ran} miseqs \Rightarrow \\
&\qquad mfs\ s = \alpha_{trans}(s.receive \cup s.send \cup s.reply \cup s.accept \cup s.break) \\
&\qquad\qquad \cup \alpha_{trans}\ \{\, (miseqtst\ s).1, (miseqtst\ s).2\,\} \\
&\qquad\qquad \cup \bigcup mfs\ (\!|\ states^{\sim}(l\ ((first \circ miseqs^{\sim})\ s.type))\ |\!)
\end{aligned}$$

$$\alpha_{clock} : PName \pfun Local \pfun \mathbb{P}\, Event$$

$$\begin{aligned}
&\alpha_{clock} = (\lambda\, p : PName \bullet (\lambda\, l : Local \bullet \\
&\qquad internals(\!|\ \{\, s : allstates\ p\ l \mid s.type \in Tasks \cup Subs \cup \operatorname{ran} itime \cup \operatorname{ran} stime\,\}\ |\!)
\end{aligned}$$

The function *internals* takes a BPMN state and returns a set of its corresponding events to be used internally in the coordination function.

$$internals : State \pfun \mathbb{P}\, Event$$

$$\begin{aligned}
&internals = (\lambda\, State \bullet \\
&\qquad (\textbf{if}\ type \in Tasks\ \textbf{then}\ \{\, \epsilon_{wait}\ s\,\}\ \textbf{else}\ \emptyset) \\
&\qquad \cup\ (second \circ unzip)\ \epsilon_{int}\ s \cup \{\, \epsilon_{fin}\ s, \epsilon_{can}\ s\,\}
\end{aligned}$$

$$tsem : PName \pfun Local \pfun Process$$

$$\begin{aligned}
&\forall\, p : PName;\ l : Local \bullet \\
&\quad \textbf{let}\ E = \Box\ i \in \epsilon_{end}\ p\ l \bullet fin.i \rightarrow Skip \\
&\qquad \Box\ \Box\ e \in \epsilon_{abort}\ p\ l \bullet abort.e \rightarrow Stop \\
&\quad \textbf{in}\ tsem\ p\ l = (bsem_T\ p\ l\ [\![\ sync\ p\ l\ ]\!]\ ((clock\ p\ l \triangle E) \setminus \alpha_{clock}\ p\ l)) \setminus (hide\ p\ l)
\end{aligned}$$

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

The function *begin* returns a set of starting states of a given BPMN diagram.

$$
\begin{array}{|l}
begin : PName \pfun Local \pfun \mathbb{P}\, State \\
\hline
begin = (\lambda\, p : PName \bullet (\lambda\, l : Local \bullet \{\, s : allstates\ l\ p \mid s.type \in Starts \,\}))
\end{array}
$$

The coordination function implements a variant of the two-phase functioning approach. The classical two-phased functioning approach can be described as follows: In the first phase, all untimed elementary actions are executed, and in the second phase, when all actions are blocked by an obligation of duration, time progresses by one unit. Our variation does not model time explicitly but rather uses timing information to coordinate between each state's enactment. Informally the function carries out the following steps:

1. Branch out and enact all untimed events and gateways until the BPMN process has reached time stablility, this will be described in Section 5.1;
2. Order all immediate *active* BPMN states, which are also timed by definition, in some sequence $\langle t_1 \,..\, t_n \rangle$ according to their shortest delay, this will be described in Section 5.2;
3. Enact all the time-ready states according to their timing information, after which remove the enacted states from the sequence. This will be described in Section 5.3;
4. Repeat Steps 1 to 3 until the BPMN process terminates.

The overall semantic function of a business collaboration is defined by the function $csem_T$, which takes a syntactic description of one or more states of type *pool*, each encapsulating a separate BPMN diagram representing an individual participant within a business collaboration, and returns a parallel composition of processes, each corresponding to an individual participant.

$$
\begin{array}{|l}
csem_T : CName \pfun Global \pfun Local \pfun Process \\
chide : CName \pfun Global \pfun Local \pfun \mathbb{P}\, Event \\
\hline
\forall\, l : Local;\ c : CName;\ g : Global \bullet \\
\quad csem\ c\ g\ l = \\
\qquad (\ \|\ ps : \{\, b : bpmns^{\sim}(g\ c) \,\} \bullet \alpha_{process}\ ps\ l \circ tsem\ ps\ l) \setminus chide\ c\ gl\ l \\
\quad \wedge \\
\quad chide\ c\ g\ l = \\
\qquad \bigcup \{\, ps : bpmns^{\sim}(g\ c);\ s : states^{\sim}(l\ ps) \bullet \\
\qquad\qquad \alpha_{msg}(s.send \cup s.receive \cup s.reply \cup s.accept \cup s.break) \,\}
\end{array}
$$

5.1 Reaching Time Stability

The function *stable* is higher order function; it takes a function f and a set of active states, and returns a process which recursively enacts all active states until the BPMN process is time-stable. The function then behaves like the supplied function f, in the definition of *clock*, f is the function *timer*.

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

$$\begin{array}{l}
\forall f : \mathbb{P}\, State \pfun Process;\ p : PName;\ l : Local;\ us, st : \mathbb{P}\, State \bullet \\
stable\ f\ p\ l\ us\ st = \\
\quad \textbf{let}\ uw = \{\, u : us \mid u.type = agate \land (\exists\, k : allstates\ p\ l \bullet u.in \subseteq k.in \land k.out > 1) \\
\qquad\qquad \land\ st \cap (prec\ p\ l\ (\mu\, k : allstates\ p\ l \mid u.in \subseteq k.in)) \neq \emptyset \,\}\ \textbf{in} \\
\quad \textbf{if}\ us = \emptyset \lor u = uw\ \textbf{then}\ f\ st \\
\quad \textbf{else}\ (\Box\, i : \bigcup(((flip(\alpha_{state}))\ l)(\!|\ u \setminus uw\ |\!)) \bullet \\
\qquad \textbf{if}\ i \notin \bigcup\{\, t : State \mid t \in allstates\ p\ l \bullet \alpha_{trans}\ (t.in \cup t.out \cup second(\!|\ t.error\ |\!)) \,\} \\
\qquad \textbf{then}\ i \rightarrow stable\ f\ p\ l\ us\ st \\
\qquad \textbf{else let}\ \ b = (\mu\, s : State \bullet s \in allstates\ p\ l \land i \in \alpha_{trans}(s.out \cup second(\!|\ s.error\ |\!))) \\
\qquad\qquad\qquad a = (\mu\, t : State \mid t \in allstates\ p\ l \land i \in \alpha_{trans} t.in)\ \textbf{in} \\
\qquad\quad \textbf{if}\ a \in \{\, t : TimeState \mid t \in allstates\ p\ l \,\} \\
\qquad\quad \textbf{then if}\ (b.type = agate \land \#b.out > 1 \lor b.type \neq agate) \\
\qquad\qquad\quad \textbf{then let}\ o = (\mu\, t : Transition \mid t \in b.out \land i = \epsilon_{line}\ t.line) \\
\qquad\qquad\qquad\quad \textbf{in}\ i \rightarrow (stable\ f\ p\ l\ ((us \setminus \{\, b \,\}) \cup \{\, (rmtrans\ b\ o) \,\})\ (st \cup \{\, a \,\})) \\
\qquad\qquad\quad \textbf{else}\ i \rightarrow (stable\ f\ p\ l\ (us \setminus \{\, b \,\})\ (st \cup \{\, a \,\})) \\
\qquad\quad \textbf{else if}\ b.type = agate \land \#b.out > 1 \\
\qquad\qquad\quad \textbf{then let}\ o = (\mu\, t : Transition \mid t \in b.out \land i = \epsilon_{line}\ t.line) \\
\qquad\qquad\qquad\quad \textbf{in}\ (i \rightarrow stable\ f\ p\ l\ ((us \setminus \{\, b \,\}) \cup \{\, (rmtrans\ b\ o), a \,\})\ st \\
\qquad\qquad\quad \textbf{else}\ i \rightarrow stable\ f\ p\ l\ (us \setminus \{\, b \,\}) \cup \{\, a \,\})\ st
\end{array}$$

The function *rmtrans* removes a transition from a given BPMN state.

$$rmtrans : State \pfun Transition \pfun State$$

$$\begin{array}{l}
rmtrans = (\lambda\, s : State \bullet (\lambda\, t : Transition \bullet \\
\qquad (\!|\ type \rightsquigarrow s.type, in \rightsquigarrow s.in, exit \rightsquigarrow s.exit, \\
\qquad\quad out \rightsquigarrow s.out \setminus \{\, t \,\}, loopMax \rightsquigarrow s.loopMax, \\
\qquad\quad receive \rightsquigarrow s.receive, send \rightsquigarrow s.send, reply \rightsquigarrow s.reply, \\
\qquad\quad accept \rightsquigarrow s.accept, break \rightsquigarrow s.break, dur \rightsquigarrow s.dur\ |\!)
\end{array}$$

The function *allstates* recursively returns a set of states contained in a BPMN diagram, including those contained within the diagram's subprocess states,

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

$$\begin{array}{l}
allstates = (\lambda\, p : PName \bullet (\lambda\, l : Local \bullet \\
\qquad \bigcup\{\, s : states^{\sim}(l\ p) \mid s \in Subs \bullet allstates\ (gn\ s)\ l \,\} \cup states^{\sim}(l\ p)
\end{array}$$

where the function *gn* returns the unique name of a given subprocess state.

$$gn : State \pfun PName$$

$$\begin{array}{l}
\forall\, s : State \bullet \\
\quad s.type \in \mathrm{ran}\ miseqs \Rightarrow gn\ l\ s = first\ (miseqs^{\sim} s.type) \\
\quad \land\ s.type \in \mathrm{ran}\ mipars \Rightarrow gn\ l\ s = first\ (mipars^{\sim} s.type) \\
\quad \land\ s.type \in \mathrm{ran}\ bpmn \Rightarrow gn\ l\ s = bpmn^{\sim} s.type
\end{array}$$

The generic function *flip* applied to the function f takes its first two arguments in the reverse order of f and is declared as follows:

$$[X, Y, Z]$$
$$flip : (X \leftrightarrow Y \leftrightarrow Z) \to (Y \leftrightarrow X \leftrightarrow Z)$$

The functions *succ* and *prec* take a BPMN state and returns a set of BPMN states, of which control flows succeed and precede the given state respectively.

$$succ, prec : \mathbb{P}\, State \nrightarrow \mathbb{P}\, State \nrightarrow State \nrightarrow \mathbb{P}\, State$$

$$\begin{array}{l}
\forall\, ss, tt : \mathbb{P}\, State;\ s : State \bullet \\
\quad succ\ ss\ tt\ s = \\
\qquad \textbf{let}\ ts = \{ t : ss \mid (s.out \cup second\, (\!|\ s.error\ |\!)) \cap t.in \neq \emptyset \land t \notin tt \} \\
\qquad \textbf{in}\ \ (\textbf{if}\ ts = \emptyset\ \textbf{then}\ tt\ \textbf{else}\ \bigcup (succ_I\ (ss \setminus ts)\ (tt \cup ts) (\!|\ ts\ |\!))) \\
\quad \land\ prec\ ss\ tt\ s = \\
\qquad \textbf{let}\ ts = \{ t : ss \mid (s.in \cap (t.out \cup second\, (\!|\ t.error\ |\!))) \neq \emptyset \land t \notin tt \} \\
\qquad \textbf{in}\ \ (\textbf{if}\ ts = \emptyset\ \textbf{then}\ tt\ \textbf{else}\ \bigcup (prec_I\ (ss \setminus ts)\ (tt \cup ts) (\!|\ ts\ |\!)))
\end{array}$$

5.2 Ordering Timed Sequence

Initially after branching and enacting all necessary untimed states to reach time stability, function *stable* calls the function *timer*. We define *timer* to order the set of currently active timed states (Step 2) according to their delays.

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

$$\begin{array}{l}
\forall\, p : PName;\ l : Local;\ ss : \mathbb{P}\, State \bullet \\
\quad timer\ p\ l\ ss = \\
\qquad \textbf{let}\ \ ss' = order_T \bigcup (sep\, (\!|\ ss\ |\!) \cup subexc\ p\ l\, (\!|\ ss\ |\!)) \\
\qquad\quad ts = \{\, n \mapsto s : ss' \bullet n \mapsto time\ s \,\} \\
\qquad\quad num = \#\{\, t : \mathrm{ran}\ ts \mid t =_T head\ ts \,\} \\
\qquad\quad cur = (take\ num\ ss') \\
\qquad\quad ss'' = \{\, s : ss \mid s \notin cur \land (s.type \in \mathrm{ran}\ miseq \\
\qquad\qquad\qquad\qquad \Rightarrow \neg\, \exists\, c : allstates\ p\ l \bullet \\
\qquad\qquad\qquad\qquad\qquad c.type \in \mathrm{ran}\ miseq \land c \approx_S s \\
\qquad\qquad\qquad\qquad\qquad \land\ (second \circ miseq^{\sim})\ s.type <_T (second \circ miseq^{\sim})\ c.type) \,\} \\
\qquad\quad ns = subt\ head\ ts\, (\!|\ ss''\ |\!) \\
\qquad \textbf{in}\ trun\ p\ l\ ns\ cur
\end{array}$$

For states that are task states with a timed exception flow, we applied the function *sep*, which returns a set containing the task state itself and a timed event state that contains the syntactic description of the task state's timed exception flow. We write $\langle\!| c_1 \rightsquigarrow v_1 \,..\, c_n \rightsquigarrow v_n |\!\rangle$ to define an object of some schema type with components $c_1 \,..\, c_n$ declared.

$$sep : State \to \mathbb{P}\, State$$

$$\begin{array}{l}
sep = (\lambda\, State \bullet \\
\qquad \textbf{if}\ (type \notin Tasks \cup Subs \lor \forall\, t : error \bullet first\ t \notin \mathrm{ran}\ itime)\ \textbf{then}\ \{\, \theta State \,\} \\
\qquad \textbf{else}\ \{\, \langle\!|\ type \rightsquigarrow (\mu\, t : error \mid first\ t \in \mathrm{ran}\ itime \bullet first\ t), in, exit \rightsquigarrow \emptyset, \\
\qquad\qquad out \rightsquigarrow \{\, (\mu\, t : error \mid first\ t \in \mathrm{ran}\ itime \bullet second\ t) \,\}, loopMax \rightsquigarrow 0, \\
\qquad\qquad receive, send, reply, accept, break \rightsquigarrow \emptyset \\
\qquad\qquad dur \rightsquigarrow \langle\!| min, max \rightsquigarrow zero_T |\!\rangle\ |\!\rangle, \theta State \,\})
\end{array}$$

We define a binary relation $\approx_S$ over *State* below to associate two states that are *flow-equivalent* - that is, having the same incoming and outgoing transitions. Note two states must be the same if they have the same transitions in the context of the same specification environment.

$$\_ \approx_S \_ : (State \leftrightarrow State)$$

$$\forall s, t : State \bullet s \approx_S t \Leftrightarrow s.in = t.in \land a.out = t.out$$

For states that are embedded in some (nested) subprocess state that has a timed exception flow, we applied the function *subexc*, which returns a set, possibly empty, of states representing all the timed exception flows attached to all subprocess states into which the state is embedded. For example, Figure 4 shows a BPMN subprocess state, if function *subexc* is applied over the state A, it will return a set of states representing timed exception flows attached to subprocess states B and C, while the function *sep* will return state A and a timed event state representing A's own the timed exception flow.

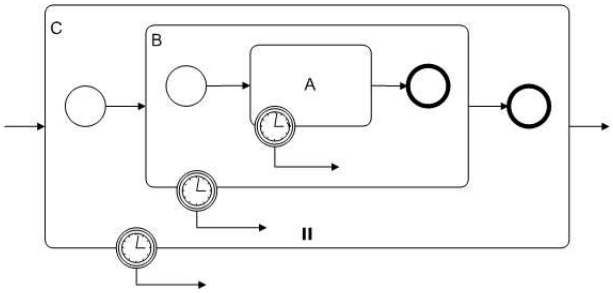

Figure 4: A BPMN subprocess state

$$subexc : PName \nrightarrow Local \nrightarrow State \nrightarrow \mathbb{P}\, State$$
$$finexc : Local \nrightarrow State \nrightarrow State \nrightarrow \mathbb{P}\, State$$
$$makexc : (Type \times Transition) \rightarrow State$$

$$\begin{array}{l}
\forall\, t : Type;\ ts : Transition;\ p : PName;\ l : Local;\ s, v : State \bullet \\
\quad subexc\ p\ l\ s = \\
\qquad \textbf{let}\ \ su = (\mu\, u : State \mid (s, u) \in tpsub(p, l)) \\
\qquad\qquad e = (\mu\, te : su.error \mid first\ te \in \mathrm{ran}\ itime)\ \textbf{in} \\
\qquad \textbf{if}\ \forall\, u : State \bullet (s, u) \notin tpsub(p, l)\ \textbf{then}\ \emptyset \\
\qquad \textbf{else}\ \{\, makexc(first\ e, second\ e)\,\} \cup finexc\ l\ s\ su \\
\quad \land \\
\quad finexc\ l\ s\ v = \\
\qquad (\textbf{if}\ (s, v) \notin insubs\ l\ v\ \textbf{then}\ \emptyset \\
\qquad \textbf{else}\ \{\, (\mu\, e : v.error \mid first\ e \in \mathrm{ran}\ itime \bullet makexc(first\ e, second\ e))\,\}) \\
\qquad \cup \bigcup (finexc\ l\ s\ (\!|\ \{\, x : states^{\sim}((l \circ gn)v) \mid x.type \in Subs\,\}\ |\!)) \\
\quad \land \\
\quad makexc(t, ts) = \langle\!|\, type \rightsquigarrow t, in, exit \rightsquigarrow \emptyset, out \rightsquigarrow \{\, ts\,\}, loopMax \rightsquigarrow 0, \\
\qquad\qquad dur \rightsquigarrow \langle\!| min, max \rightsquigarrow zero_T |\!\rangle, receive, send, reply, accept, break \rightsquigarrow \emptyset\, |\!\rangle
\end{array}$$

We define the function *allsub*, which recursively returns the set of states, each being contained in the given state.

$$
\begin{array}{l}
allsub : Local \pfun State \pfun \mathbb{P}\, State \\
\hline
\forall\, l : Local;\ su : State \bullet \\
\quad allsub\ l\ su = \\
\qquad \textbf{if}\ su.type \notin Subs\ \textbf{then}\ \emptyset \\
\qquad \textbf{else}\ \bigcup(allsub\ l\ (\!|\ \{\ s : states^{\sim}((l \circ gn)su) \mid s.type \in Subs\ \}\ |\!)) \cup states^{\sim}((l \circ gn)su)
\end{array}
$$

The parameterised relation *tpsub* maps each state contained in some nested subprocess state to its topmost subprocess state with a timed exception flow given a BPMN process via its name and its specification environment. The function *insub* takes a subprocess state and returns a binary relation associating each state contained in some nested subprocess state with a timed exception flow to that subproces state.

$$
\begin{array}{l}
tpsub : (PName \times Local) \fun (State \rel State) \\
insubs : Local \pfun State \pfun (State \rel State) \\
\hline
\forall\, p : PName;\ l : Local;\ su : State \bullet \\
\quad tpsub(p, l) = \bigcup\{\ s : states^{\sim}(l\ p) \mid s.type \in Subs \land (\exists\, e : s.error \bullet first\ e \in \operatorname{ran} itime) \\
\qquad\qquad \bullet\ \{\ t : allsub\ l\ s \bullet t \mapsto s\ \}\ \} \\
\qquad\quad \cup \bigcup\{\ s : states^{\sim}(l\ p) \mid s.type \in Subs \\
\qquad\qquad \land\ (\forall\, e : s.error \bullet first\ e \notin \operatorname{ran} itime) \bullet insub\ l\ s\ \} \\
\quad \land \\
\quad insubs\ l\ su = \\
\qquad \textbf{if}\ \exists\, e : su.error \bullet first\ e \in \operatorname{ran} itime\ \textbf{then}\ \{\ t : allsub\ l\ su \bullet t \mapsto su\ \} \\
\qquad \textbf{else}\ \bigcup(insubs\ l\ (\!|\ \{\ s : states^{\sim}((l \circ gn)su) \mid s.type \in Subs\ \}\ |\!))))
\end{array}
$$

The function *timer* also employs the function $order_T$ to order a set of timed states into a timed sequence. For task states which have delays over a specfied range, we assume their lower bounds as their delay initially. The function *timer* then calls the function *trun*, which is defined in Section 5.3, with a set of time-ready states *cu*, which is the range of the initial segment of the timed sequence and a set of states that belongs the range of the remaining seqment of the timed sequence, with which the states' delays are subtracted by the currently shortest delay. We write $s \frown t$ to denote concatenation of sequences s and t.

$$
\begin{array}{l}
order_T : \mathbb{P}\ TimeState \pfun \operatorname{seq} TimeState \\
\hline
\forall\, ss : \mathbb{P}\ TimeState \bullet \\
\quad ss = \emptyset \Rightarrow order_T\ ss = \langle\rangle \\
\quad \land\ ss \neq \emptyset \Rightarrow order_T\ ss = \ \ \textbf{let}\ s = (\mu\, s : ss \mid \forall\, t : ss \bullet time\ s \leq_T time\ t) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \langle s\rangle \frown order_T(ss \setminus \{\ s\ \})
\end{array}
$$

$$
\begin{array}{l}
time : TimeState \fun Time \\
\hline
\forall\, t : TimeState \bullet \\
\quad t.type \in \operatorname{ran} stime \Rightarrow time\ t = stime^{\sim} t.type \\
\quad \land\ t.type \in \operatorname{ran} itime \Rightarrow time\ t = itime^{\sim} t.type \\
\quad \land\ t.type \in Tasks \cup Subs \Rightarrow time\ t = t.ran.min
\end{array}
$$

The function *take* takes the a natural number n and a sequence of states ss and returns a set of states containinig the first n elements of ss.

$$take : \mathbb{N} \to \text{seq}\, State \to \mathbb{P}\, State$$

$$\begin{array}{l}
\forall\, n : \mathbb{N};\ ss : \text{seq}\, State \bullet \\
\quad n \neq 0 \land ss \neq \langle\rangle \Rightarrow take\ n\ ss = \{\, head\ ss \,\} \cup take\ (n-1)\ (tail\ ss) \\
\quad \land\ n = 0 \lor ss = \langle\rangle \Rightarrow take\ n\ ss = \emptyset
\end{array}$$

The function *subt* subtracts the duration of a timed state with a specified duration.

$$\begin{array}{l}
subt : Time \nrightarrow TimeState \nrightarrow TimeState \\
\hline
\forall\, t : Time;\ s : TimeState \bullet \\
\quad type \in \text{ran}\, task \Rightarrow \\
\qquad subt\ t\ s = \\
\qquad\quad \langle\!|\, type \rightsquigarrow s.type, in \rightsquigarrow s.in, exit \rightsquigarrow s.exit, out \rightsquigarrow s.out, loopMax \rightsquigarrow s.loopMax, \\
\qquad\qquad dur \rightsquigarrow \langle\!|min \rightsquigarrow (\textbf{if}\ s.ran.min \leq_T t\ \textbf{then}\ zero_T\ \textbf{else}\ s.ran.min -_T t), \\
\qquad\qquad\qquad\quad max \rightsquigarrow (\textbf{if}\ s.ran.max \leq_T t\ \textbf{then}\ zero_T\ \textbf{else}\ s.ran.max -_T t)|\rangle \\
\qquad\qquad receive \rightsquigarrow s.receive, send \rightsquigarrow s.send, reply \rightsquigarrow s.reply, accept \rightsquigarrow s.accept, \\
\qquad\qquad break \rightsquigarrow s.break,\ |\rangle \\
\quad \land \\
\quad type \in \text{ran}\, itime \Rightarrow \\
\qquad subt\ t\ s = \\
\qquad\quad \langle\!|\, type \rightsquigarrow itime(\textbf{if}\ (itime^{\sim} s.type) \leq_T t\ \textbf{then}\ zero_T\ \textbf{else}\ (itime^{\sim} s.type) -_T t), \\
\qquad\qquad in \rightsquigarrow s.in, exit \rightsquigarrow s.exit, out \rightsquigarrow s.out, loopMax \rightsquigarrow s.loopMax, dur \rightsquigarrow s.dur \\
\qquad\qquad receive \rightsquigarrow s.receive, send \rightsquigarrow s.send, reply \rightsquigarrow s.reply, accept \rightsquigarrow s.accept, \\
\qquad\qquad break \rightsquigarrow s.break,\ |\rangle \\
\quad \land \\
\quad type \in \text{ran}\, stime \Rightarrow \\
\qquad subt\ t\ s = \\
\qquad\quad \langle\!|\, type \rightsquigarrow stime(\textbf{if}\ (stime^{\sim} s.type) \leq_T t\ \textbf{then}\ zero_T\ \textbf{else}\ (stime^{\sim} s.type) -_T t), \\
\qquad\qquad in \rightsquigarrow s.in, exit \rightsquigarrow s.exit, out \rightsquigarrow s.out, loopMax \rightsquigarrow s.loopMax, dur \rightsquigarrow s.dur \\
\qquad\qquad receive \rightsquigarrow s.receive, send \rightsquigarrow s.send, reply \rightsquigarrow s.reply, accept \rightsquigarrow s.accept, \\
\qquad\qquad break \rightsquigarrow s.break,\ |\rangle
\end{array}$$

We define the subtraction operator $-_T$ over durations Similar operator $+_T$ for addition may also be defined.

$$\begin{array}{l}
_ -_T _ : (Time \times Time) \to Time \\
\hline
\forall\, s, t : Time \bullet \\
\quad s\ -_T\ t = \langle\!|year \rightsquigarrow s.year - t.year, month \rightsquigarrow s.month - t.month, \\
\qquad\qquad day \rightsquigarrow s.day - t.day, hour \rightsquigarrow s.hour - t.hour, \\
\qquad\qquad minute \rightsquigarrow s.minute - t.minute, second \rightsquigarrow s.second - t.second|\rangle
\end{array}$$

5.3 Coordinating Timed States

We now define the function *trun*, which 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 partial 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 BPMN process. Before enacting the time-ready states, the following rules are applied.

- For all time-ready states that are sequential multiple instance task states, we instantiate one instance of the task modelled by each of the multiple instance states via the function *splitsep* and reduce the number of multiple instance recorded by one. The *instantiated task* states are time-ready and each multiple instance states will becoming time-ready upon the termination of their respective instantiated state;
- For all time-ready states that are parallel multiple instance task states, we instantiate all instances of the task modelled by each of the multiple instance states via the function *splitpar*. Each *instantiated task* states is time-ready.

$$
\begin{array}{|l}
trun : PName \pfun Local \pfun \power State \pfun \power State \pfun Process \\
\hline
\forall\, p : PName;\ l : Local;\ ss, cu : \power State \bullet \\
\quad trun\ p\ l\ ss\ cu = \\
\qquad \textbf{let}\ (ms, is) = unzip\,(splitseq(\!|\ \{\, c : cu \mid c.type \in \ran miseq \,\}\ |\!)) \\
\qquad\quad ps = splitpar(\!|\ \{\, c : cu \mid c.type \in \ran mipar \,\}\ |\!) \\
\qquad\quad cu' = \{\, c : cu \mid c.type \notin Mults \,\} \cup ps \cup is \\
\qquad\quad sy = \bigcup(internal(\!|\ (\epsilon_{can}\ \{\, c : cu \mid c.type \in Mults \,\}) \cup cu'\ |\!))\ \textbf{in} \\
\qquad ((trun'\ p\ l\ cu' \fatsemi run(sy))\ [\![\ sy\ ]\!]\ record\ p\ l\ (ss \cup ms)\ cu'\ \emptyset) \setminus sy
\end{array}
$$

The execution process is described in Section 5.3.1 and the recording process is described in Section 5.3.2. We define the function *splitseq* for Step 5.3 above and this function is defined to take a sequential multiple instance state and returns a pair where the first element is a task state representing one instance of the multiple instance state and the second element is the multiple instance state with number of instance reduced by one.

$$
\begin{array}{|l}
splitseq : State \pfun (State \times State) \\
\hline
splitseq = \\
\quad (\lambda\, s : State \mid type \in \ran miseq \bullet \\
\qquad \{\, (\!|\, type \sim (task\,(first \circ miseq^\sim)\ s.type), loopMax \sim (second \circ miseq^\sim)\ s.type, dur \sim s.dur, \\
\qquad\quad in, out, error, exit \sim \emptyset, receive, send, reply, accept, break \sim \emptyset \,|\!), reduce\ s \,\})
\end{array}
$$

We define the function *splitpar* for Step 5.3 above and this function is defined to take a parallel multiple instance state with n multiple instances and returns n task states containing no incoming and outgoing transitions, each representing one instance of the multiple instance state, we use the schema component *loopMax* to differentiate each state.

$$splitpar : State \pfun \mathbb{P}\, State$$

$$\begin{array}{l}
splitpar = \\
\quad (\lambda\, s : State \mid type \in \operatorname{ran} mipar \bullet \\
\qquad \textbf{if } (second \circ mipar^{\sim})\, type = 0 \textbf{ then } \emptyset \\
\qquad \textbf{else } \{ \langle type \mapsto (task\, (first \circ mipar^{\sim})\, s.type), in, out, error, exit \mapsto \emptyset, dur \mapsto s.dur, \\
\qquad\qquad loopMax \mapsto (second \circ mipar^{\sim})\, s.type, receive, send, reply, accept, break \mapsto \emptyset \rangle \} \\
\qquad\quad \cup\, (splitpar \circ reduce)\, s)
\end{array}$$

The function *reduce* reduces the number of instance specfied in the supplied multiple instance task state by one.

$$reduce : State \pfun State$$

$$\begin{array}{l}
\forall\, s : State \bullet \\
\quad type \in \operatorname{ran} miseq \land (second \circ miseq^{\sim}) s.type > 0 \Rightarrow \\
\qquad reduce\, s = \langle type \mapsto miseq((first \circ miseq^{\sim})\, s.type, (second \circ miseq^{\sim})\, s.type - 1), \\
\qquad\qquad in \mapsto s.in, out \mapsto s.out, error \mapsto s.error, \\
\qquad\qquad receive \mapsto s.receive, send \mapsto s.send, reply \mapsto s.reply, accept \mapsto s.accept, \\
\qquad\qquad break \mapsto s.break, loopMax \mapsto s.loopMax, dur \mapsto s.dur \rangle \\
\quad \land \\
\quad type \in \operatorname{ran} mipar \land (second \circ mipar^{\sim}) s.type > 0 \Rightarrow \\
\qquad reduce\, s = \langle type \mapsto mipar((first \circ mipar^{\sim})\, s.type, (second \circ mipar^{\sim})\, s.type - 1), \\
\qquad\qquad in \mapsto s.in, out \mapsto s.out, error \mapsto s.error, \\
\qquad\qquad receive \mapsto s.receive, send \mapsto s.send, reply \mapsto s.reply, accept \mapsto s.accept, \\
\qquad\qquad break \mapsto s.break, loopMax \mapsto s.loopMax, dur \mapsto s.dur \rangle
\end{array}$$

5.3.1 Execution Process

The function $trun'$ returns a process that interleaves the enactment of the processes corresponding to the subset of the currently active timed states described above. Informally each process corresponding to each states behaves according to the following rules:

- if some state s is a timed event of type *itime* and *stime* and it is not a representation of a timed exception flow then s will be enacted, after which the process notifies the the recording process via the event $\epsilon_{fin}\, s$, then the process terminates. State s may be interrupted at any time by an exception flow of a subprocess state that contains s;
- If the corresponding state is a task state then the function *ttrn* is applied to it. Similarly this state may be interrupted at any time by an exception flow of a subprocess state that contains it;
- If the corresponding state is an *instantiated task* state then the function *tmrn* is applied to it. Similarly this state may be interrupted at any time by an exception flow of a subprocess state that contains the multiple instance state which it instantiates.

After the interleaving of the processes corresponding to all time-ready states terminate, the function $trun'$ terminates and behaves like the process *run* which is defined here.

$$run(A) = \Box\, a : A \bullet a \rightarrow run(A)$$

$$trun' : PName \pfun Local \pfun \power State \pfun Process$$

$$\begin{lenv}
\forall\, p : PName;\ l : Local;\ cu : \power State \bullet \\
\quad trun'\ p\ l\ cu = \\
\qquad \LET par == \{\, c : cu \mid c.type \in \ran task \\
\qquad\qquad \land\, (\exists\, t : allstates\ p\ l \bullet t.type \in \ran mipar \\
\qquad\qquad\qquad \land\, (task^\sim c.type) = (first \circ mipar^\sim)\ t.type) \,\} \IN \\
\qquad (\lll (p, ps) : (pars\ \ p\ l\ par\ (getmult\ p\ l\ mipar\ (\!|\ par\ |\!))) \bullet \\
\qquad\quad \lpar [\{\, \epsilon_{fin}\ p, \epsilon_{int}\ p \,\}]\ s : ps \bullet \\
\qquad\qquad ((tmrn\ mipar\ p\ l\ (cu \setminus \{\, s \,\})\ s) \\
\qquad\qquad \catastrophe\ (\epsilon_{int}\ p \then Skip \extchoice \epsilon_{fin}\ p \then Skip \extchoice \epsilon_{can}\ p \then Skip))) \\
\qquad \lll (\lll s : \{\, c : (cu \setminus par) \mid c.type \in \ran itime \implies c.in \neq \emptyset \,\} \bullet \\
\qquad\quad \LET it == \exists\, t : allstates\ p\ l \bullet t.type \in \ran miseq \\
\qquad\qquad\qquad \land\, (task^\sim s.type) = (first \circ miseq^\sim)\ t.type \IN \\
\qquad\quad (\IF s.type \in \ran task \land it \\
\qquad\quad \THEN \LET q == getmult\ p\ l\ miseq\ s \IN \\
\qquad\qquad tmrn\ miseq\ p\ l\ (cu \setminus \{\, s \,\})\ s \\
\qquad\qquad \catastrophe\ (\epsilon_{int}\ q \then Skip \extchoice \epsilon_{fin}\ q \then Skip \extchoice \epsilon_{can}\ q \then Skip) \\
\qquad\quad \ELSE (\IF s.type \in stime \lor (s.type \in itime \land s.in \neq \emptyset) \\
\qquad\qquad \THEN (XS(s.out) \fatsemi \epsilon_{end}\ s \then Skip) \ELSE ttrn\ p\ l\ (cu \setminus \{\, s \,\})\ s) \\
\qquad\qquad \catastrophe\ \epsilon_{can}\ s \then Skip))
\end{lenv}$$

$$pars : PName \pfun Local \pfun \power State \pfun \power State \pfun \power(State \times \power State)$$

$$\begin{lenv}
\forall\, p : PName;\ l : Local;\ ss, ps : \power State \bullet \\
\quad pars\ \ p\ l\ ss\ ps = \{\, p : ps \bullet (p, \{\, s : ss \bullet getmult\ p\ l\ mipar\ s = p \,\}) \,\}
\end{lenv}$$

We define the function *getmult* to take a state representing a task instance of a multiple instance state and returns that multiple instance state.

$$getmult : PName \pfun Local \pfun ((Task \times \nat) \pfun Type) \pfun State \pfun State$$

$$\begin{lenv}
\forall\, p : PName;\ l : Local;\ tc : ((Task \times \nat) \pfun Type);\ s : State \bullet \\
\quad s.type \in \ran task \implies \\
\qquad getmult\ p\ l\ tc\ s = (\mu\, t : allstates\ p\ l \mid t.type \in \ran tc \land (first \circ tc^\sim)\ t.type = task^\sim s.type)
\end{lenv}$$

We define the function *ttrn* the coordination of a time-ready task state. Here we describe the coordination informally.

1. If the task state s is embedded in some subprocess state k, which has a timed exception flow that expires in the current time passage, the timed exception flow interrupts the task state, and all other active states embedded in this subprocess state, the function then triggers the outgoing transition of the timed exception, and notifies the recording process via the event $\epsilon_{int}\ k$ and terminates;

2. If the task state s is not embedded and has been attached with a timed exception flow that expires in the current time passage, the timed exception flow interrupts the task state, the function then triggers the outgoing transition of the timed exception, and notifies the recording process via the event $\epsilon_{int}\ s$ and terminates;

3. If the task state s is not embedded and does not contain a timed exception flow, the function chooses either to enact the state, triggers s's outgoing transition and notifies the recording process via the event $\epsilon_{fin}\ s$ and terminates, or to delay up to the end of the current time passage, notifies the recording process via the event $\epsilon_{wait}\ s$ and terminates,

$$
\begin{array}{|l}
ttrn : PName \pfun Local \pfun \mathbb{P}\, State \pfun State \pfun Process \\
\hline
\forall\, p : PName;\ l : Local;\ cu : \mathbb{P}\, State;\ s : State \bullet \\
\quad ttrn\ p\ l\ cu\ s = \\
\qquad \textbf{if}\ \{\, c : cu \mid \exists\, k : subexc\ p\ l\ s \bullet c \approx_S k \,\} \neq \emptyset \\
\qquad \textbf{then}\ (\Box\, se : \{\, c : cu \mid \exists\, k : subexc\ p\ l\ s \bullet c \approx_S k \,\} \bullet \\
\qquad\qquad \textbf{let}\ et = (\mu\, v : Event \mid \exists\, u : allstates\ p\ l \bullet \\
\qquad\qquad\qquad \exists\, e : u.error \bullet second\ e = strans\ se \\
\qquad\qquad\qquad \land\ \exists\, i : \epsilon_{int}\ u \bullet first\ i = second\ e \bullet second\ i) \\
\qquad\qquad \textbf{in}\ XS(se.out) \fatsemi et \rightarrow Skip \\
\qquad \textbf{else}\ (\textbf{if}\ \exists\, k : cu;\ t : Type \bullet (t, strans\ k) \in s.error \\
\qquad\qquad \textbf{then}\ (\textbf{let}\ tx = (\mu\, k : cu \mid \exists\, t : Type \bullet (t, strans\ k) \in s.error) \\
\qquad\qquad\qquad et = (\mu\, v : Event \mid \exists\, i : \epsilon_{int}\ s \bullet first\ i = strans\ tx \bullet second\ i) \\
\qquad\qquad\qquad \textbf{in}\ XS(tx.out) \fatsemi et \rightarrow Skip) \\
\qquad\qquad \textbf{else}\ (\textbf{let}\ es = \{\, e : s.error \mid first\ e \notin \operatorname{ran} itime \bullet second\ e \,\} \\
\qquad\qquad\qquad Tk = (\textbf{if}\ es = \emptyset\ \textbf{then}\ \epsilon_{task}\ (task^{\sim} s.type) \rightarrow XJ(s.out) \fatsemi \epsilon_{fin}\ s \rightarrow Skip \\
\qquad\qquad\qquad\qquad \textbf{else}\ ((\epsilon_{task}\ (task^{\sim} s.type) \rightarrow Skip \triangle XJ(es)) \\
\qquad\qquad\qquad\qquad\qquad [\![\, \alpha_{trans}\ es \,]\!]\ (except\ s \Box (XJ(s.out) \fatsemi \epsilon_{fin}\ s \rightarrow Skip)))) \\
\qquad\qquad\qquad \textbf{in if}\ s.ran.min =_T s.ran.max\ \textbf{then}\ Tk\ \textbf{else}\ (TK \sqcap \epsilon_{wait}\ s \rightarrow Skip))))
\end{array}
$$

We define the function *strans* to return the outgoing transition of the given state, which has exactly one outgoing transition. This function is called in the function *ttrn*.

$$
\begin{array}{|l}
strans : State \pfun Transition \\
\hline
strans = (\lambda\, State \mid \#out = 1 \bullet (\mu\, t : Transition \mid t \in out))
\end{array}
$$

$$
\begin{array}{|l}
except : State \pfun Process \\
\hline
except = (\lambda\, s : State \bullet \\
\qquad (\Box\, i : \{e : s.error \mid first\ e \notin \operatorname{ran} itime \bullet second\ e\,\} \bullet \\
\qquad\quad \textbf{let}\ \ et = (\mu\, k : \epsilon_{int}\ s \mid i = first\ k \bullet second\ k) \\
\qquad\quad \textbf{in}\ \epsilon_{line}\ i.line \rightarrow et \rightarrow Skip))
\end{array}
$$

We define the function *tmrn* for the coordination of a time-ready task state that instantiates an instance of a multiple instances task state. Here we describe the coordination informally.

1. If the task state instantiates a multiple instance state m, which is contained in some subprocess state q that has an expiring timed exception flow, the timed exception flow interrupts the instantiated task state, the function then triggers the outgoing transition of the timed exception, and notifies the recording process via the event $\epsilon_{int}\ q$ and terminates;

2. If the task state instantiates a multiple instance state m, which is not embedded but has an expiring timed exception flow, the timed exception flow interrupts the instantiated task state, the function then triggers the outgoing transition of the timed exception, and notifies the recording process via the event $\epsilon_{int}\ m$ and terminates;

3. If some task state s instantiates a multiple instance state m, which is not embedded and does not have an expiring timed exception flow, the function may perform one of the following by calling the function *tsrn* and *tprn* for sequential and parallel multiple instance states respectively:

   - The function may enact s, notify the recording process via the event $\epsilon_{fin}\ s$ and terminate;
   - The function may delay s up to the end of the current time passage and notify the recording process via the event $\epsilon_{wait}\ s$ and terminate;
   - The function may enact s and decide to terminates m, it will then notify the recording process via the event $\epsilon_{fin}\ m$ and terminate the whole multiple instance state m;
   - If s is the nth instantiation of m where n is the maximum number of multiple instances, then the function will enact s, trigger the outgoing transition of m and notify recording process via the event $\epsilon_{fin}\ m$ and terminate the whole multiple instance state m.

$$
\begin{array}{l}
tmrn : ((Task \times \mathbb{N}) \rightarrowtail Type) \nrightarrow PName \nrightarrow Local \nrightarrow \mathbb{P}\, State \nrightarrow \mathbb{P}\, State \\
\qquad \nrightarrow State \nrightarrow Process \\
\hline
\forall\, tc : ((Task \times \mathbb{N}) \rightarrowtail Type);\ p : PName;\ l : Local;\ ss, cu : \mathbb{P}\, State;\ s : State \bullet \\
tmrn\ tc\ p\ l\ ss\ cu\ s = \\
\quad \textbf{let}\ ms = getmult\ p\ l\ tc\ s\ \textbf{in} \\
\quad \textbf{if}\ \{\, c : cu \mid \exists\, k : subexc\ p\ l\ ms \bullet c \approx_S k \,\} \neq \emptyset \\
\quad \textbf{then}\ (\Box\, se : \{\, c : cu \mid \exists\, k : subexc\ p\ l\ ms \bullet c \approx_S k \,\} \bullet \\
\qquad \textbf{let}\ et = (\mu\, s : allstates\ p\ l \mid \exists\, e : s.error;\ i : \epsilon_{int}\ s \bullet \\
\qquad\qquad second\ e = strans\ se \wedge first\ i = second\ e \bullet second\ i) \\
\qquad \textbf{in}\ (XS(se.out) \fatsemi et \rightarrow \epsilon_{wait}\ s \rightarrow Skip)) \\
\quad \textbf{else if}\ (\exists\, k : cu;\ t : Type \bullet (t, strans\ k) \in ms.error) \\
\qquad \textbf{then}\ (\textbf{let}\ tx = (\mu\, k : cu \mid \exists\, t : Type \bullet (t, strans\ k) \in ms.error) \\
\qquad\qquad et = (\mu\, e : Event \mid \exists\, i : \epsilon_{int}\ ms \bullet first\ i = strans.tx \bullet second\ i) \\
\qquad\quad \textbf{in}\ XS(tx.out) \fatsemi et \rightarrow Skip) \\
\qquad \textbf{else}\ (tc = miseq\ \&\ tsrn\ p\ l\ ss\ cu\ s \,\Box\, tc = mipar\ \&\ tprn\ p\ l\ ss\ cu\ s)
\end{array}
$$

The function *tsrn* defines the part of the coordination described in Step 3 for sequential multiple instance states.

$$
\begin{array}{l}
tsrn : PName \pfun Local \pfun \mathbb{P}\, State \pfun \mathbb{P}\, State \pfun State \pfun Process \\
\hline
\forall\, p : PName;\ l : Local;\ ss, cu : \mathbb{P}\, State;\ s : State \bullet \\
\quad tsrn\ p\ l\ ss\ cu\ s = \\
\qquad (\textbf{let}\ \ mq = (\mu\, x : ss \mid x \in \operatorname{ran} tc \land (first \circ tc^{\sim})\, x.type = task^{\sim} s.type) \\
\qquad\quad es = \{\, e : mq.error \mid first\ e \notin \operatorname{ran} itime \bullet second\ e \,\} \\
\qquad\quad Sk = \textbf{if}\ es \neq \emptyset \\
\qquad\qquad \textbf{then}\ ((\epsilon_{task}\ (task^{\sim} s.type) \fatsemi Skip \mathbin{\triangle} XJ(es)) \llbracket\, \alpha_{trans}\ es \,\rrbracket \\
\qquad\qquad\qquad ((second \circ miseq^{\sim})\ mq.type > 0\ \&\ \epsilon_{fin}\ s \fatsemi Skip \\
\qquad\qquad\qquad\ \Box\ XJ(mq.out) \fatsemi \epsilon_{fin}\ mq' \fatsemi Skip \Box except\ mq)) \\
\qquad\qquad \textbf{else}\ (\epsilon_{task}\ (task^{\sim} s.type) \fatsemi \\
\qquad\qquad\qquad (second \circ miseq^{\sim})\ mq.type > 0\ \&\ \epsilon_{fin}\ s \fatsemi Skip \\
\qquad\qquad\qquad\ \Box\ XJ(mq.out) \fatsemi \epsilon_{fin}\ mq \fatsemi Skip\ \textbf{in} \\
\qquad\quad \textbf{if}\ s.ran.min =_T s.ran.max\ \textbf{then}\ Sk\ \textbf{else}\ (Sk \sqcap \epsilon_{wait}\ s \fatsemi Skip)))
\end{array}
$$

The function *tprn* defines the part of the coordination described in Step 3 for parallel multiple instance states

$$
\begin{array}{l}
tprn : PName \pfun Local \pfun \mathbb{P}\, State \pfun \mathbb{P}\, State \pfun \mathbb{P}\, State \pfun State \pfun Process \\
\hline
\forall\, p : PName;\ l : Local;\ ss, cu, wt : \mathbb{P}\, State;\ s : State \bullet \\
\quad tprn\ p\ l\ ss\ cu\ wt\ s = \\
\qquad \textbf{let}\ \ mc = \{\, x : cu \mid getmult\ p\ l\ mipar\ x = getmult\ p\ l\ mipar\ s \,\} \\
\qquad\quad mw = \{\, x : wt \mid getmult\ p\ l\ mipar\ x = getmult\ p\ l\ mipar\ s \,\} \\
\qquad\quad mt = getmult\ p\ l\ mipar\ s \\
\qquad\quad es = \{\, e : mt.error \mid first\ e \notin \operatorname{ran} itime \bullet second\ e \,\} \\
\qquad\quad Pk = \textbf{if}\ es \neq \emptyset \\
\qquad\qquad \textbf{then}\ ((\epsilon_{task}\ (task^{\sim} s.type) \fatsemi Skip \mathbin{\triangle} XJ(es)) \llbracket\, \alpha_{trans}\ es \,\rrbracket \\
\qquad\qquad\qquad (mc \cup mw \neq \emptyset)\ \&\ \epsilon_{fin}\ s \fatsemi Skip) \\
\qquad\qquad\qquad\ \Box\ XJ(mt.out) \fatsemi \epsilon_{fin}\ mt \fatsemi Skip \Box except\ mt)) \\
\qquad\qquad \textbf{else}\ ((\epsilon_{task}\ (task^{\sim} s.type) \fatsemi \\
\qquad\qquad\qquad (mc \cup mw \neq \emptyset)\ \&\ \epsilon_{fin}\ s \fatsemi Skip) \\
\qquad\qquad\qquad\ \Box\ XJ(mt.out) \fatsemi \epsilon_{fin}\ mt \fatsemi Skip\ \textbf{in} \\
\qquad\quad \textbf{if}\ s.ran.min =_T s.ran.max\ \textbf{then}\ Pk\ \textbf{else}\ (Pk \sqcap \epsilon_{wait}\ s \fatsemi Skip)))
\end{array}
$$

5.3.2 Recording Process

The function *record* defines the recording process. It receives coordination events from the execution process and recalculates the set of timed-active states. The following describes the function informally:

- At each recursive call to *record*, it is applied with the following three sets of BPMN states
  - The set of states that are currently active, and they can either be timed or untimed (ss);
  - The set of timed states that are currently active and time-ready (cu);
  - The set of states that are currently active and time-ready but have descided to delay their enactment (wt).

  The values of these sets represent the *state* of the BPMN process during timed coordination.

- If both *cu* and *ss* are empty, all time-ready task and multiple instance states have delayed their enactments. The function then re-calculates these states so that the states, of which the delay range has the shortest upper bound, are to be enacted.
- If *cu* is empty and *ss* is not empty, all time-ready states have either been enacted or delayed. The function then branches out and enacts subsequent active untimed states via the functions *mstable* and *stable* defined in Section 5.1, and the function $timer'$ defined below;
- If *cu* is not empty, then wait for the following types of coordination events:
  1. On receiving a terminating event $e \in \operatorname{ran} \epsilon_{fin}$, it applies the function *finish* to the state $\epsilon^{\sim}_{fin}\, e$;
  2. On receiving a delay event $e \in \operatorname{ran} \epsilon_{wait}$, it applies the function *wait* to the state $\epsilon^{\sim}_{wait}\, e$;
  3. On receiving an interrupting event $e \in (second \circ unzip) \bigcup(\operatorname{ran} \epsilon_{int})$, it applies the function *interrupt* to the state representing that exception.

$$
\begin{array}{l}
record : PName \pfun Local \pfun \mathbb{P}\, State \pfun \mathbb{P}\, State \pfun \mathbb{P}\, State \pfun Process \\
\hline
\forall\, p : PName;\ l : Local;\ ss, cu, wt : \mathbb{P}\, State \bullet \\
\quad record\ p\ l\ ss\ cu\ wt = \\
\qquad \textbf{if}\ cu = \emptyset \\
\qquad \textbf{then if}\ ss = \emptyset \\
\qquad\qquad \textbf{then let}\ m = (\mu\, w : wt \mid \forall\, x : ws \bullet w.ran.max \leq_T x.ran.max \bullet w.ran.max) \\
\qquad\qquad\qquad \textbf{in}\ trun\ p\ l\ \emptyset\ (subt\ m(\!|\ wt\ |\!)) \\
\qquad\qquad \textbf{else let}\ pos = (\{\, s : ss \mid (s.type \in \operatorname{ran} itime \wedge s.in = \emptyset)\,\} \cup \{\, s : TimeState \mid s \in ss\,\}) \\
\qquad\qquad\qquad \textbf{in}\ stable\ (timer'\ p\ l\ ss\ wt)\ p\ l\ (ss \setminus pos)\ (pos \cup wt) \\
\qquad \textbf{else}\ (\Box\, i : \alpha_{clock}\ p\ l \bullet \\
\qquad\qquad (\textbf{if}\ i \in \operatorname{ran} \epsilon_{fin}\ \textbf{then}\ finish\ p\ l\ ss\ cu\ wt\ (\epsilon^{\sim}_{fin}\, i) \\
\qquad\qquad\ \textbf{else}\ (\textbf{if}\ i \in \operatorname{ran} \epsilon_{wait}\ \textbf{then}\ wait\ p\ l\ ss\ cu\ wt\ (\epsilon^{\sim}_{wait}\, i) \\
\qquad\qquad\qquad \textbf{else}\ (\textbf{if}\ i \notin (second \circ unzip) \bigcup(\operatorname{ran} \epsilon_{int})\ \textbf{then}\ Stop \\
\qquad\qquad\qquad\qquad \textbf{else}\ (\textbf{if}\ \exists\, s : allstates\ p\ l \bullet (\exists\, e : \epsilon_{int}\ s \bullet second\ e = i \wedge first\ e \in \operatorname{ran} itime) \\
\qquad\qquad\qquad\qquad\qquad \textbf{then let}\ k = (\mu\, s : cu \mid \exists\, t : allstates\ p\ l \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \exists\, e : \epsilon_{int}\ t \bullet second\ e = i \wedge s.type = first\ e) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ interrupt\ p\ l\ ss\ cu\ wt\ k \\
\qquad\qquad\qquad\qquad\qquad \textbf{else let}\ k = (\mu\, a : allstates\ p\ l \mid \exists\, e : a.error;\ f : \epsilon_{int}\ a \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad second\ f = i \wedge second\ e = first\ f \bullet makexc\ e) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ interrupt\ p\ l\ ss\ cu\ wt\ k))))))
\end{array}
$$

Function *finish* defines part of the re-calculation on receiving a terminating event (Rule 1 above). The following describes the function informally:

- If the terminating state is a task state, then the function removes the state from the set of time-ready states *cu*, the function also removes all states representing timed exception flows of this state from the set of active states *ss* and appends all states, which immediately succede this state to *ss*;

- If the terminating state is a sequential multiple instance state, then the function removes the instantiated task state from the set of time-ready states cu, the function also removes all states representing timed exception flows of the multiple instance state and the multiple instance state itself from the set of active states ss, and appends all states, which immediately succede this state to ss;
- If the terminating state is a parallel multiple instance state, then the function removes all its instantiated task states from the set of time-ready states cu, the function also removes all states representing timed exception flows of the multiple instance state and the multiple instance state itself from the set of active states ss, and appends all states, which immediately succede this state to ss.

$$
\begin{array}{l}
finish : PName \pfun Local \pfun \mathbb{P}\, State \pfun \mathbb{P}\, State \pfun \mathbb{P}\, State \pfun State \pfun Process \\
\hline
\forall\, p : PName;\ l : Local;\ ss, cu, wt : \mathbb{P}\, State;\ s : State \bullet \\
\quad finish\ p\ l\ ss\ cu\ wt\ s = \\
\qquad \textbf{let}\ es = \{\, k : ss \mid \exists\, t : Type \bullet (t, strans\ k) \in s.error \,\}\ \textbf{in} \\
\qquad \textbf{if}\ s.type \in \operatorname{ran} task \\
\qquad \textbf{then}\ record\ p\ l\ ((ss \setminus es) \cup next\ p\ l\ (\!|\ s.out\ |\!))\ (cu \setminus \{\, s \,\})\ wt \\
\qquad \textbf{else if}\ s.type \in \operatorname{ran} miseq \\
\qquad\quad \textbf{then let}\ ms = (\mu\, t : ss \mid t \approx_S getmult\ p\ l\ miseq\ s) \\
\qquad\qquad\qquad it = (\mu\, t : cu \mid s.type = ((first \circ splitseq)\ s).type)\ \textbf{in} \\
\qquad\qquad record\ p\ l\ ((ss \setminus (\{\, ms \,\} \cup es)) \cup next\ p\ l\ (\!|\ s.out\ |\!))\ (cu \setminus \{\, it \,\})\ wt \\
\qquad\quad \textbf{else if}\ s.type \in \operatorname{ran} mipar \\
\qquad\qquad \textbf{then let}\ ip = \{\, i : ss \cup cu \mid i.type \in \operatorname{ran} task \land s = getmult\ p\ l\ mipar\ i \,\}\ \textbf{in} \\
\qquad\qquad\quad record\ p\ l\ ((ss \setminus (\{\, ms \,\} \cup es)) \cup next\ p\ l\ (\!|\ s.out\ |\!))\ (cu \setminus \{\, it \,\})\ wt \\
\qquad\qquad \textbf{else}\ record\ p\ l\ (ss \cup next\ p\ l\ (\!|\ s.out\ |\!))\ (cu \setminus \{\, s \,\})\ wt
\end{array}
$$

The function *wait* defines part of the re-calculation on receiving a delay event (Rule 2 above). It removes the delaying state from ss and appends it to the set of delayed states wt, subtracting the amount of delay from the state's delay range.

$$
\begin{array}{l}
wait : PName \pfun Local \pfun \mathbb{P}\, State \pfun \mathbb{P}\, State \pfun \mathbb{P}\, State \pfun State \pfun Process \\
\hline
\forall\, p : PName;\ l : Local;\ ss, cu, wt : \mathbb{P}\, State;\ s : State;\ \bullet \\
\quad wait\ p\ l\ ss\ cu\ wt\ s = record\ p\ l\ ss\ (cu \setminus \{\, s \,\})\ (wt \cup \{\, subt\ s.ran.min\ s \,\})
\end{array}
$$

The function *interrupt* defines part of the re-calculation on receiving an interrupt event (Rule 3 above). The following describes the function informally:

- If the state s is a representation of an exception of a time-ready task state, the function applies *inttask* to s.
- If the state s is a representation of an exception of a time-ready sequential multiple instance state, the function applies *intmiseq* to s.
- If the state s is a representation of an exception of a time-ready sequential multiple instance state, the function applies *intmipar* to s.

- If the state s is a representation of an exception of a subprocess state, the function removes states, contained in the subprocess state, and their timed exception representations from the set of active states ss, the set of time-ready state cu and the set of delayed states wt.

$$
\begin{array}{|l}
interrupt : PName \pfun Local \pfun \power State \pfun \power State \pfun \power State \pfun State \pfun Process \\
\hline
\forall\, p : PName;\ l : Local;\ ss, cu, wt : \power State;\ s : State \bullet \\
\quad interrupt\ p\ l\ ss\ cu\ wt\ s = \\
\qquad \textbf{if}\ \exists\, c : cu;\ a : allstates\ p\ l \bullet (s.type, strans\ s) \in a.error \land c \approx_S a \\
\qquad \textbf{then let}\ k = (\mu\, a : allstates\ p\ l \bullet (s.type, strans\ s) \in a.error) \\
\qquad\qquad \textbf{in}\ \ (\textbf{if}\ \exists\, c : cu \bullet k \approx_S c\ \textbf{then}\ inttask\ p\ l\ ss\ cu\ wt\ s \\
\qquad\qquad\qquad \textbf{else if}\ \exists\, c : cu \bullet k = getmult\ p\ l\ miseq\ c\ \textbf{then}\ intmiseq\ p\ l\ ss\ cu\ wt\ s \\
\qquad\qquad\qquad\qquad \textbf{else}\ intmipar\ p\ l\ ss\ cu\ wt\ s) \\
\qquad \textbf{else let}\ all = allexc\ p\ l\ (ss \cup cu)\ s \\
\qquad\qquad\qquad all' = \{\, w : all \mid (\exists\, c : cu \bullet c \approx_S w \lor w = getmult\ p\ l\ c) \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\, \neg(w.type \in \mathrm{ran}\ itime \land w.in = \emptyset)\,\} \\
\qquad\qquad \textbf{in}\ \ (|||\ a : all' \bullet \epsilon_{can}\ a \rightarrow Skip)\ \fatsemi \\
\qquad\qquad\qquad (record\ p\ l\ (rmsub\ ss\ all \cup next\ p\ l\ (\!|\ s.out\ |\!))\ (rmsub\ cu\ all)\ (rmsub\ wt\ all))
\end{array}
$$

The function *inttask* is called when *interrupt* receives an exception of a time-ready task state. It removes the state representing timed exception of the task state from the set of active states ss and the set of time-ready states cu. It then "activates" the states that are triggered by the exception flow by appending them to ss, after which the function behaves as *record*.

$$
\begin{array}{|l}
inttask : PName \pfun Local \pfun \power State \pfun \power State \pfun \power State \pfun State \pfun Process \\
\hline
\forall\, p : PName;\ l : Local;\ ss, cu, wt : \power State;\ s : State \bullet \\
\quad inttask\ p\ l\ ss\ cu\ wt\ s = \\
\qquad \textbf{let}\ k = (\mu\, c : cu \mid \exists\, a : allstates\ p\ l \bullet (s.type, strans\ s) \in a.error \land c \approx_S a) \\
\qquad \textbf{in}\ record\ p\ l\ (\{\, s : ss \mid \neg \exists\, e : k.error \bullet strans\ s = second\ e\,\} \cup next\ p\ l\ (\!|\ s.out\ |\!)) \\
\qquad\qquad\qquad (\{\, c : cu \mid \neg \exists\, e : k.error \bullet strans\ c = second\ e\,\} \setminus \{\, k\,\})\ wt
\end{array}
$$

The function *intmiseq* is called when *interrupt* receives an exception of a time-ready sequential multiple instance state. It removes its flow-equivalent multiple instance state from the set of active states ss, its instantiated task state from the set of time-ready states cu, and the states representing the multiple instance state's timed exception flows from both ss and cu. It then "activates" the states that are triggered by the exception flow by appending them to ss, after which the function behaves as *record*.

$$
\begin{array}{|l}
intmiseq : PName \pfun Local \pfun \power State \pfun \power State \pfun \power State \pfun State \pfun Process \\
\hline
\forall\, p : PName;\ l : Local;\ ss, cu, wt : \power State;\ s : State \bullet \\
\quad intmiseq\ p\ l\ ss\ cu\ wt\ s = \\
\qquad \textbf{let}\ k = (\mu\, c : cu \mid \exists\, a : allstates\ p\ l \bullet (s.type, strans\ s) \in a.error \land a = getmult\ p\ l\ miseq\ c) \\
\qquad\qquad\ l = (\mu\, s : ss \mid \exists\, a : allstates\ p\ l \bullet a = getmult\ p\ l\ miseq\ k \land s \approx_S a) \\
\qquad \textbf{in}\ record\ p\ l\ ((\{\, s : ss \mid \neg \exists\, e : l.error \bullet strans\ s = second\ e\,\} \setminus \{\, l\,\}) \cup next\ p\ l\ (\!|\ s.out\ |\!)) \\
\qquad\qquad\qquad (\{\, c : cu \mid \neg \exists\, e : l.error \bullet strans\ c = second\ e\,\} \setminus \{\, k\,\})\ wt
\end{array}
$$

The function *intmipar* is called when *interrupt* receives an exception of a time-ready parallel multiple instance state. It removes its instantiated task states

from the set of active states ss, the set of time-ready states cu and the set of delayed states wt. It also removes states representing the multiple instance state's timed exception flows from both ss and cu. It then "activates" the states that are triggered by the exception flow by appending them to ss, after which the function behaves as *record*.

$$intmipar : PName \pfun Local \pfun \power State \pfun \power State \pfun \power State \pfun State \pfun Process$$

$$\begin{aligned}
&\forall\, p : PName;\ l : Local;\ ss, cu, wt : \power State;\ s : State \bullet \\
&\quad intmipar\ p\ l\ ss\ cu\ wt\ s = \\
&\qquad \textbf{let}\ ks = \{\, c : cu \mid \exists\, a : allstates\ p\ l \bullet (s.type, strans\ s) \in a.error \land a = getmult\ p\ l\ mipar\ c \,\} \\
&\qquad\quad l = (\mu\, a : allstates\ p\ l \mid (s.type, strans\ s) \in a.error) \\
&\qquad \textbf{in}\ record\ p\ l\ (\{\, s : ss \mid \neg \exists\, e : l.error \bullet strans\ s = second\ e \,\} \cup next\ p\ l\ (\!|\ s.out\ |\!)) \\
&\qquad\qquad\qquad (\{\, c : cu \mid \neg \exists\, e : l.error \bullet strans\ c = second\ e \,\} \setminus ks)\ (wt \setminus ks)
\end{aligned}$$

The function *allexc* takes a set of states and a state representing an exception flow and returns all the states that are contained in the subprocess state that has the given exception flow, and the states that represent all timed exceptions flows embedded in that subprocess state.

$$allexc : PName \pfun Local \pfun \power State \pfun State \pfun \power State$$

$$\begin{aligned}
&\forall\, p : PName;\ l : Local;\ ss : \power State;\ s : State \bullet \\
&\quad allexc\ p\ l\ ss\ s = \\
&\qquad (\mu\, u : allstates\ p\ l \mid u.type \in Subs \land \exists\, e : u.error \bullet second\ e = strans\ s \\
&\qquad\quad \bullet \{\, w : ss \mid \exists\, x : allsub\ l\ u \bullet \exists\, y : x.error \bullet strans\ w = second\ y \,\} \cup allsub\ l\ u)
\end{aligned}$$

The function *rmsub* performs a variant set difference operation over two sets of states such that it removes some state s of the first set based on the following rules:

- if s is an instantiated task state of some multiple instance state in the second set;
- if s is not an instantiated task state and there exists a flow-equivalent state in the second set.

$$rmsub : \power State \pfun \power State \pfun \power State$$

$$\begin{aligned}
&rmsub = \\
&\quad (\lambda\, ss : \power State \bullet \lambda\, tt : \power State \bullet \\
&\qquad ss \setminus \{\, s : ss \mid \exists\, t : tt \bullet \\
&\qquad\qquad (t.type \in \operatorname{ran} mipar \land s.type \in \operatorname{ran} task \\
&\qquad\qquad\qquad \Rightarrow task^{\sim} s.type = (first \circ mipar^{\sim})\ t.type \\
&\qquad\qquad \land\ (t.type \in \operatorname{ran} miseq \land s.type \in \operatorname{ran} task \\
&\qquad\qquad\qquad \Rightarrow s.type = ((first \circ splitseq)\ t).type) \\
&\qquad\qquad \land\ (t.type \in \operatorname{ran} miseq \land s.type \in \operatorname{ran} miseq) \lor t.type \notin Mults \\
&\qquad\qquad\qquad \Rightarrow s \approx_S t \,\}
\end{aligned}$$

The function *next* is defined to return the State of which the set of incoming transitions contains the given transition.

$$next : PName \pfun Local \pfun Transition \pfun State$$

$$\forall\, p : PName;\ l : Local;\ t : Transition \bullet next\ p\ l\ t = (\mu\, s : allstates\ p\ l \mid t \in s.in)$$

After all active states in the current time interval have either been enacted or decided to delay its enactment, the function *trun*, defined at the beginning of this section, branches and enacts all active untimed states until reaching time stability. We define the function $timer'$ to re-calculate the timing information defined in each active timed states. The function takes the following sets of states as arguments.

- The set of timed states that are active before time stability has been reached (as).
- The set of timed states that are active before time stability has been reached and have non-deterministically delayed its enactment (ws).
- The set of all timed states that are active after time stability has been reached (ss).

$$timer' : PName \pfun Local \pfun \power State \pfun \power State \pfun \power State \pfun Process$$

$$\begin{array}{l}
\forall\, p : PName;\ l : Local;\ ss, as, ws : \power State \bullet \\
\quad timer'\ p\ l\ as\ ws\ ss = \\
\qquad \textbf{let}\ \ ns = \{\, s : ss \mid \neg\, \exists\, a : as \cup ws \bullet s \approx_S a \,\} \cup as \\
\qquad\qquad fs = (time \circ head)\ (order_T \bigcup (sep\ (\!|\ (ns \cup \bigcup (subexc\ p\ l\ (\!|\ ss\ |\!)))\ |\!))) \\
\qquad\qquad nws = subt\ fs\ (\!|\ ns\ |\!) \cup \{\, s : ws \mid s.ran.max \leq_T fs \bullet update\ zero_T\ zero_T\ s \,\} \\
\qquad\qquad\qquad \cup \{\, s : ws \mid s.ran.max >_T fs \bullet update\ zero_T\ (s.ran.max -_T fs)\ s \,\} \\
\qquad \textbf{in}\ timer\ p\ l\ nws
\end{array}$$

The function creates a set of timed states nws using the three sets described above. This set represents the set of currently active timed states. The set contains states in as and states in a subset of ss that are not already defined in as and ws as their duration either have changed or would have changed from the original syntactic description. This set also contains all the states from ws according the following rules:

- If a state in ws, of which the maximum delay (schema component $ran.max$) is less than or equal to the shortest duration of states that are currently active, the state's duration range will be changed into instantaneous enactment ($ran.min =_T ran.max =_T zero_T$) ;
- If a state in ws, of which the maximum delay (schema component $ran.max$) is larger than the shortest duration of states that are currently active, the state's duration range's upper bound will be subtracted from that shortest duration.

The function then calls *timer* defined in Section 5.2 to order nws according to Step 2 defined on Page 23. We define the function *update* to update the duration of a timed task state with specified lower bound and upper bound.

$$update : Time \pfun Time \pfun TimeState \pfun TimeState$$

$$\begin{array}{l}
\forall\, t, u : Time;\ s : TimeState \bullet \\
\quad type \in \mathrm{ran}\ task \Rightarrow \\
\qquad update\ t\ u\ s = \\
\qquad\qquad (\!|\ type \mapsto s.type, in \mapsto s.in, exit \mapsto s.exit, out \mapsto s.out, loopMax \mapsto s.loopMax, \\
\qquad\qquad\quad receive \mapsto s.receive, send \mapsto s.send, reply \mapsto s.reply, accept \mapsto s.accept, \\
\qquad\qquad\quad break \mapsto s.break, dur \mapsto (\!|\ min \mapsto t, max \mapsto u\ |\!)\ |\!)
\end{array}$$

6 Analysis

We recall from our earlier work [11] the untimed process semantics for a local BPMN diagram (single participant with no message flow) is given by the $bsem\, p\, l$ where p and l are the diagram's name and local specificaion environments respectively. As a result of using the untimed CSP as the semantic domain for both BPMN's relative-timed and untimed semantics, we are able to show several properties relating the untimed semantics with the relative-timed semantics over the syntax of BPMN with timing information.

First to differentiate between BPMN diagrams with and without timing information, we say a BPMN diagram is *timed* if it contains timing information and *untimed* otherwise. We then define an abstraction function *abstract* on the syntax of BPMN with timing information so that we may apply the untimed semantic function to it. The abstraction may be defined as follows:

- If the BPMN state is a timed task or subprocess state (of type *task*, *bpmn*, *miseq*, *mipar*, *miseqs*, *mipars*), then untimed semantic function will simply ignore the timing information given by the state's schema component *dur*, and any timed exception flow defined upon the state will be abstracted into an untimed internal exception flow of type *ierror*.
- If the BPMN state is a timed start state (of type *stime*) then it is abstracted into an untimed start state of type *start*.
- If the BPMN state is an intermediate delay state (of type *itime*), we remove it from the BPMN diagram and join the state directly precedes the delay state to the state directly succedes it.

$$abstract : PName \pfun Local \pfun PName$$

$$\forall\, p : PName;\ l : Local \bullet$$
$$abstract\, p\, l = (\mu\, r : PName \mid allstate\, r\, l = abs\, ((head \circ \mathrm{seq})(chg(\!|\, allstate\, p\, l \,|\!)))\, \emptyset$$

The function *abs* takes a set of BPMN states representing a local diagram with timing information and returns a set of BPMN states with that timing information abstracted.

$$abs : \mathbb{P}\, State \pfun \mathbb{P}\, State \pfun \mathbb{P}\, State$$

$$\begin{array}{l}
\forall\, ss, tt : \mathbb{P}\, State \bullet \\
\quad abs\, ss\, tt = \\
\qquad \textbf{if } ss = \emptyset \textbf{ then } tt \\
\qquad \textbf{else if } head(ss).type \in \mathrm{ran}\, itime \\
\qquad\quad \textbf{then let } p = (\mu\, s : State \mid s \in ss \cup tt \land s.out \cap head(ss).in \neq \emptyset) \\
\qquad\qquad\qquad\quad s = (\mu\, s : State \mid s \in ss \cup tt \land s.in \cap head(ss).out \neq \emptyset) \\
\qquad\qquad\qquad\quad pt = (\mu\, t : Transition \mid t \in (p.out \cap head(ss).in)) \\
\qquad\qquad\qquad\quad s' = cht\, s\, ((strans \circ head)\, ss)\, pt \\
\qquad\qquad\quad \textbf{in } abs\, (squash(ss \rhd \{\, p, s, head(ss) \,\}))\, ((tt \setminus \{\, s \,\}) \cup \{\, p, s' \,\}) \\
\qquad\quad \textbf{else } abs\, tail(ss)\, (tt \cup head(ss))
\end{array}$$

The function *chg* essentially maps each timed variant start, task, multiple instance and subprocess state to its untimed variant.

$$chg : State \pfun State$$

$$\begin{array}{l} chg = (\lambda\, State \bullet \\ \quad \textbf{let}\ \ tp = \textbf{if}\ type \in \operatorname{ran} stime\ \textbf{then}\ start\ \textbf{else}\ type \\ \qquad\ \ er = \{\, e : error \mid first\ e \in \operatorname{ran} itime \bullet (ierror, second\ e) \,\} \\ \qquad\qquad \cup \{\, e : error \mid first\ e \notin \operatorname{ran} itime \,\}\ \textbf{in} \\ \quad (\!|\, type \mapsto tp, in \mapsto in, exit \mapsto exit, out \mapsto out, loopMax \mapsto loopMax, \\ \qquad receive \mapsto receive, send \mapsto send, reply \mapsto reply, \\ \qquad accept \mapsto accept, break \mapsto break, dur \mapsto dur \,|\!) \end{array}$$

The function *cht* replaces either an incoming or an outgoing/error transition with the transition specified by the argument.

$$cht : State \pfun Transition \pfun Transition \pfun State$$

$$\begin{array}{l} chi = (\lambda\, State \bullet (\lambda\, s : Transition \bullet (\lambda\, t : Transition \bullet \\ \quad \textbf{let}\ \ in' = \textbf{if}\ s \in in\ \textbf{then}\ (in \setminus \{\, s \,\}) \cup \{\, t \,\}\ \textbf{else}\ in \\ \qquad\ \ out' = \textbf{if}\ s \in out\ \textbf{then}\ (out \setminus \{\, s \,\}) \cup \{\, t \,\}\ \textbf{else}\ out \\ \qquad\ \ err' = \textbf{if}\ \forall\, t : Type \bullet (t, s) \notin error\ \textbf{then}\ error \\ \qquad\qquad \textbf{else let}\ e' = (\mu\, e : error \mid second\ e = s) \\ \qquad\qquad\qquad \textbf{in}\ (error \setminus \{\, e \,\}) \cup \{\, (first\ e, t) \,\}\ \textbf{in} \\ \quad (\!|\, type \mapsto tp, in \mapsto in', exit \mapsto exit, out \mapsto out', loopMax \mapsto loopMax, \\ \qquad receive \mapsto receive, send \mapsto send, reply \mapsto reply, \\ \qquad accept \mapsto accept, break \mapsto break, dur \mapsto dur \,|\!) \end{array}$$

We also define the function *refine* as the "inverse" of *abstract*

$$refine : PName \pfun Local \pfun \mathbb{P}\, PName$$

$$refine = (\lambda\, p : PName;\ l : Local \bullet \{\, q : PName \mid p = abstract\ q\ l \,\})$$

The following defines timed and untimed process instances.

Definition 6.1 ***Process Instance*** *A process instance of a BPMN diagram represents one possible execution and it is a sequence of BPMN states being triggered, starting from a start state.*

We say a process instance is *complete* if the sequence begins from a start state and ends in either an end state or an abort state. We may augment this definition with the relative-timed semantics.

Definition 6.2 ***Timed Process Instance*** *A process instance of a BPMN diagram is timed if it represents one possible execution of the diagram under the relative-timed semantics and it is a sequence of BPMN states being triggered, starting from a start state.*

We say a process instance is *timed-complete* if it is timed and the sequence begins from a start state and ends in either an end state or an abort state.

Proposition 6.3 ***Untimed Invariance*** *For all BPMN diagram where the only differences are their timing information, their untimed semantics are failures-equivalent.*

One common behavioural specification any process would like to satisfy is 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.

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

Definition 6.4 *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 preserve certain behavioural properties from the untimed to the timed world. We achieve this by showing for any local diagram, such that for all its timed variation, the timed coordination process is a *responsivness plug-in* [9] to the enactment process. We first formally present Reed et al.'s definition of the binary relation *RespondsTo* over CSP using the stable failures model.

Definition 6.5 *For any process P and Q where there exists a set of shared events J, Q RespondsTo P iff for all trace $s \in \mathrm{seq}(\alpha P \cup \alpha Q)$ and event set X*

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

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 6.6 ***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 sequential composition of them also cooperates with p's enactment. ∎

A direct consequence of Proposition 6.6 is that deadlock freedom is preserved from the untimed to the timed setting.

Proposition 6.7 ***Deadlock Freedom Preservation*** *For any local diagram p and environment l such that for all diagrams q where $p = abstract\ q\ l$*

$$(DF \sqsubseteq_{\mathcal{F}} bsem\ p\ l \Rightarrow DF \sqsubseteq_{\mathcal{F}} tsem\ q\ l)$$

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

Definition 6.8 ***Time Independence*** *A behavioural specification Spec is time-independent with respect to some local diagram p and environment l iff for all diagram q such that $p = abstract\ q\ l$*

$$Spec \sqsubseteq_{\mathcal{F}} bsem\ p\ l \Rightarrow \forall\, q \bullet Spec \sqsubseteq_{\mathcal{F}} tsem\ q\ l$$

As a consequence of Propositions 6.6 and 6.7, and refinements over $\mathcal{T}$ and we can generalise timed-independent specifications by the following result.

Proposition 6.9 *A specification process Spec is time-independent with respect to some untimed local diagram p and environment l iff*

$$\begin{aligned} &Spec \sqsubseteq_{\mathcal{F}} bsem\ p\ l \setminus S \Leftrightarrow \\ &\quad traces(Spec) \supseteq traces(bsem\ p\ l \setminus S) \\ &\quad \wedge\ deadlocks(Spec) \supseteq traces(bsem\ p\ l \setminus S) \end{aligned}$$

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

The notion of time independence may be augmented to the behavioural semantics itself.

Definition 6.10 *A BPMN diagram, specified by the name p and environment l, is* time-independent *iff for all diagrams q where p = abstract q l*

$$bsem\ p\ l \equiv_{\mathcal{F}} tsem\ q\ l$$

As intuitively expected, all sequential untimed BPMN diagrams (without parallel gateway states of type *agate*) with no timed exception flows are failures-equivalent to their untimed counterparts.

We can now turn to the relationship of compatibility of participants in a business collaboration between their untimed and timed semantics, note we use the term global diagram to represent the syntactic description of a collaboration and local diagram to represent the syntactic description of individual participants. First we revisit the example given in Figure 1, which shows a trivial BPMN diagram describing a collaboration between participant $p1$ and $p2$. While $p1$ performs task A then task B, $p2$ performs tasks C and D in a interleaving manner.

We define $I1$ to index the processes corresponding to the states in the participant $p1$.

$$I1 = \{\, start, a, b, end \,\}$$

By applying the untimed semantic function upon the syntactic description of $p1$, we obtain the process corresponding to it.

$$\begin{aligned} M1 &= M1' \setminus \{\!|init|\!\} \\ M1' &= \ \mathbf{let}\ C = \Box\, x : (\alpha M1' \setminus \{\, fin.1 \,\}) \bullet (x \rightarrow C \,\Box\, fin.1 \rightarrow Skip) \\ &\quad\ \ \mathbf{in}\ (\ \|\ i : I1 \bullet \alpha P1(i) \circ P1(i)\ [\![\, \alpha M1' \,]\!]\ C) \end{aligned}$$

where for each i in $I1$, the process $P1(i)$ is as defined below and $\alpha P1(i)$ is the set of possible events performed by $P1(i)$.

$$\begin{aligned} P1(start) &= init.a \rightarrow fin.1 \rightarrow Skip \\ P1(a) &= (init.a \rightarrow starts.a \rightarrow msg.a.c.mi \rightarrow msg.c.a.md \rightarrow init.b \rightarrow P1(a)) \\ &\quad\ \ \Box\, fin.1 \rightarrow Skip \\ P1(b) &= (init.b \rightarrow starts.b \rightarrow msg.d.b.mi \rightarrow msg.b.d.md \rightarrow init.end \rightarrow P1(b)) \\ &\quad\ \ \Box\, fin.1 \rightarrow Skip \\ P1(end) &= init.end \rightarrow fin.1 \rightarrow Skip \end{aligned}$$

Similarly we define $I2$ to index the processes corresponding to the states in the participant $p2$.

$$I2 = \{\, start, as, c, d, aj, end \,\}$$

By applying the untimed semantic function upon the syntactic description of $p2$, we obtain the process corresponding to it.

$$\begin{array}{l} M2 = M2' \setminus \{\!|init|\!\} \\ M2' = \quad \textbf{let}\ C = \Box\, x : (\alpha M2' \setminus \{\, fin.2 \,\}) \bullet (x \rightarrow C \,\Box\, fin.2 \rightarrow Skip) \\ \qquad\quad \textbf{in}\ (\ \|\ i : I2 \bullet \alpha P2(i) \circ P2(i))\ [\![\alpha M2']\!]\ C \end{array}$$

where for each i in $I2$, the process $P2(i)$ is as defined below and $\alpha P2(i)$ is the set of possible events performed by $P2(i)$.

$$\begin{array}{l} P2(start) = init.as \rightarrow fin.2 \rightarrow Skip \\ P2(as) = (init.as \rightarrow (init.c \rightarrow Skip \ |||\ init.d \rightarrow Skip) \fatsemi P2(as)) \,\Box\, fin.2 \rightarrow Skip \\ P2(c) = (init.c \rightarrow msg.a.c.mi \rightarrow starts.c \rightarrow msg.c.a.md \rightarrow init.aj1 \rightarrow P2(c)) \\ \qquad\quad \Box\, fin.2 \rightarrow Skip \\ P2(d) = (init.d \rightarrow msg.d.b.mi \rightarrow starts.d \rightarrow msg.b.d.md \rightarrow init.aj2 \rightarrow P2(d)) \\ \qquad\quad \Box\, fin.2 \rightarrow Skip \\ P2(aj) = ((init.aj1 \rightarrow Skip \ |||\ init.aj2 \rightarrow Skip) \fatsemi init.end \rightarrow P2(aj)) \,\Box\, fin.2 \rightarrow Skip \\ P2(end) = init.end \rightarrow fin.2 \rightarrow Skip \end{array}$$

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

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

As described in our earlier on BPMN's untimed semantics [11], CSP's stable-failures refinement ordering allows us to verifying 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 5. If

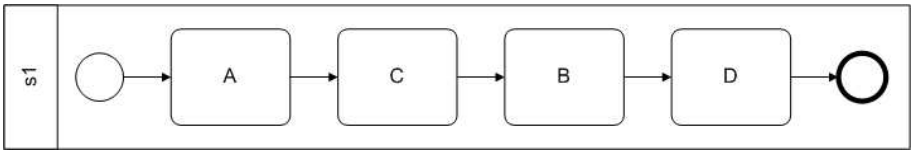

Figure 5: A specification of the intended behaviour of collaboration between $p1$ and $p2$

the CSP process *Spec* models the untimed semantics of $s1$, we run FDR to check the following refinement assertion.

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

This assertion tells us that the behaviour of the collaboration the specification described by *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 all the task states (A, B, C and D) have the following delay range (i.e. a duration between 30 minutes to 1 hour),

$$\begin{array}{l}\langle\!|\, min \rightsquigarrow \langle\!|\, year, month, day, hour, second \rightsquigarrow 0, minute \rightsquigarrow 30 \,|\!\rangle, \\ \quad max \rightsquigarrow \langle\!|\, year, month, day, minute, second \rightsquigarrow 0, hour \rightsquigarrow 1 \,|\!\rangle \,|\!\rangle\end{array}$$

we may define the timed semantics of the collaboration by defining the process corresponding to individual participant's coordination using the coordination function *clock* defined in Section 5. Process $C1$ defines the coordination of participant $pool1$.

$$\begin{array}{l} C1 = init.a \rightarrow C1_1 \\ C1_1 = starts.a \rightarrow init.b \rightarrow C1_2 \\ C1_2 = starts.b \rightarrow init.end \rightarrow C1_3 \\ C1_3 = fin.1 \rightarrow Skip \end{array}$$

The process corresponding to the timed semantics of participant $pool1$ therefore is the partial interleaving of the enactment process $M1'$ (i.e. process corresponding to its the untimed behaviour without hiding events corresponding to the diagram's control flows) and the coordination process $C1$.

$$T1 = (M1' \,[\![\, \alpha M1 \cap \alpha C1 \,]\!]\, C1) \setminus \{\!|init|\!\}$$

Similarly, process $C2$ defines the coordination of participant $pool2$,

$$\begin{array}{l} C2 = init.as \rightarrow C2_1 \\ C2_1 = (init.c \rightarrow Skip \;|||\; init.d \rightarrow Skip) \fatsemi C2_2 \\ C2_2 = (starts.c \rightarrow init.aj1 \rightarrow Skip \;|||\; starts.d \rightarrow init.aj2 \rightarrow Skip) \fatsemi C2_3 \\ C2_3 = init.end \rightarrow fin.2 \rightarrow Skip \end{array}$$

and process $T2$ defines the timed semantics of participant $poo2$.

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

Hence the timed semantics of their collaboration is the parallel composition of processes $T1$ and $T2$.

$$TC = (T1[\alpha M1 \;||\; \alpha M2]T2) \setminus \{\!|msg|\!\}$$

Since all timed states within the collaboration have uniform delay range, process TC (the timed model), should be semantically equivalent to process UC (the untimed model) under the stable-failures refinement ordering. This can be proved if we run FDR to check the following assertion.

$$UC \equiv_{\mathcal{F}} TC$$

Now let's suppose the delay range is not uniform across the collaboration and that task C has the following delay ranges,

$$\begin{array}{l}\langle\!|\, min \rightsquigarrow \langle\!|\, year, month, day, hour, second \rightsquigarrow 0, minute \rightsquigarrow 45 \,|\!\rangle, \\ \quad max \rightsquigarrow \langle\!|\, year, month, day, second \rightsquigarrow 0, hour \rightsquigarrow 1, minute \rightsquigarrow 15 \,|\!\rangle \,|\!\rangle\end{array}$$

and the following process $C3$ describes the coordination of participant $pool2$.

$$C3 = init.as \rightarrow C3_1$$
$$C3_1 = (init.c \rightarrow Skip \;|||\; init.d \rightarrow Skip) \fatsemi C3_2$$
$$C3_2 = ((starts.d \rightarrow init.aj2 \rightarrow C3_3) \sqcap C3_4)$$
$$C3_3 = starts.c \rightarrow init.aj1 \rightarrow C3_5$$
$$C3_4 = (starts.c \rightarrow init.aj1 \rightarrow Skip \;|||\; starts.d \rightarrow init.aj2 \rightarrow Skip) \fatsemi C3_5$$
$$C3_5 = init.end \rightarrow fin.2 \rightarrow Skip$$

and so we have process $T3$ defining the relative-timed semantics of participant $poo2$, and process TC' describing the timed semantics of their collaboration.

$$T3 = (M2' \;[\![\, \alpha M2 \cap \alpha C3 \,]\!]\; C3) \setminus \{\!|init|\!\}$$
$$TC' = (T1[\alpha M1 \;||\; \alpha M2]\, T3) \setminus \{\!|msg|\!\}$$

The following refinement assertion checks whether the collaboration behaves as specified by the diagram $s1$.

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

When we ask FDR to check this assertion the following counterexample in the form of a failure is given

$$(\langle starts.a \rangle, \Sigma)$$

This tells us that after 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 delay are chosen internally over a range without the coorperation 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 *timed-compatibility* using CSP's responsiveness.

Definition 6.11 ***Timed-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 timed-incompatible with respect to the collaboration C iff for any process T_i*

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

As for the example in above, to confirm $p1$ and $p2$ are timed-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.

$$DF \sqsubseteq_{\mathcal{F}} T1 \wedge DF \sqsubseteq_{\mathcal{F}} T3$$

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

Proposition 6.12 *Given the participants P_i, where i ranges over some index set, are timed-compatible in some collaboration C, their refinements under $\mathcal{F}$ are also timed-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 *timed-compatible class* of participants of which any member may replace it and preserves timed-compatibility. This class may be formalised via the stable failures equivalence. This notion augments our earlier definitions in the untimed setting [11].

Definition 6.13 ***Timed-Compatible Class*** *Given a collaboration participant named p, specified in some environment l, we define its timed-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.*

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

where the function mg returns a set of CSP events describing the alphabet of the states of a given BPMN diagram, which defines message flows.

$$
\begin{array}{|l}
mg : PName \pfun Local \pfun \mathbb{P}\, Event \\
\hline
mg = (\lambda\, p : PName \bullet (\lambda\, l : Local \bullet \\
\qquad \bigcup \{\, s : State \mid s \in (states^{\sim}(l\ p)) \\
\qquad\quad \land \bigcup (\alpha_{msg}(\!|\ s.send \cup s.receive \cup s.reply \cup s.accept \cup s.break\ |\!)) \neq \emptyset \bullet \alpha_{state}\ s\ l\})) \\
\end{array}
$$

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

Definition 6.14 ***Characteristic Participant***. *Given the timed-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 6.12, and Definitions 6.13 and 6.14.

Proposition 6.15 *If a characteristic participant p of a timed-compatible class cp, specified in some environment l, is timed-compatible with respect to some collaboration c, then all participants in cp are also timed-compatible with respect to c.*

7 Related Work

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 model for similar notation such as UML activity diagrams [4, 5]. However, neither do their semantics provide the level of abstraction required to model the six-dimensional space defined by W3C standards [14] nor do their timed model allow analyses of collaborations where more than one diagram is under consideration.

As in the untimed setting 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.

8 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 timed coordination langauges 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.

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 [12];
- applying the timed model to reason about empirical studies against safety properties [12].

References

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

[2] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. 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] N. Guelfi and A. 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. D. Bosschere, and A. Brogi. On the expressiveness of timed coordination models. *Sci. Comput. Program.*, 61(2):152–187, 2006.

[7] Object management group. `www.omg.org`.

[8] OMG. *Business Process Modeling Notation (BPMN) Specification*, Feb. 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] P. Y. H. Wong and J. 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] P. Y. H. Wong and J. Gibbons. On Specifying and Visualising Long-Running Empirical Studies (extended version). `http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/transext.pdf`, 2007.

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

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

A Proofs

Proposition A.1 ***(Responsiveness 6.6)*** *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 *clock* and show that for any local diagram, its coordination process is *a responsive plug-in* to its enactment process, i.e. 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 sequential composition of them also cooperates with p's enactment. Note at any point if the enactment of p terminates or aborts by behaving like $fin?n \rightarrow Skip$ and $aborts?n \rightarrow Stop$, by definition of the coordination process, it can also behave like those processes.

case *clock*: By definition every local diagram must start by triggering one of its *start* or *stime* states outgoing transitions, and the definition of *clock* begins by doing exactly that - performing external choice over all *start* and *stime* states, after which it behaves as the process defined by *stable* over the singleton set containing the state triggered after executing one of the start states.

case *stable*: If p's enactment can trigger an untimed state then *stable* can by definition also trigger an untimed state, If p's enactment can only trigger timed states and nothing else then p is time stable and by definition the set *us* is either empty or only contains states of type *agate* which are preceded by timed states that have not yet been triggered. In both case the *stable* will then behave as $f\ st$ where f is either the function *timer* or $timer'$.

On the other hand if p's enactment can trigger both timed and untimed states then all timed states should be blocked until p is timed stable. By definition all timed states will be in the set *st* and untimed states in set *us*, this means *stable* will be able to trigger those untimed states by performing external choice over them recursively.

Consequently stable will always be able to perform at least a subset of states that can be performed by p until p is time stable or terminates.

case *timer*: When diagram p is time stable, by defintion there exists a set of active timed state which p's enactment is able to execute in a interleaving manner. During time stability, *timer* returns a process that orders the set of active timed states and then behaves like *trun*, which coordinate the current time ready states, which is by definition a non-empty subset of the active timed states. Since ordering states does not engage in any CSP events at the semantic level, and *timer* will always behave as *trun* after ordering, the function does not cause deadlock at the semantic level if and only if *trun* does not cause deadlock at the semantic level.

case *trun*: When all timed states are ordered, this function coordinates a set of time ready states, this being a non-empty subset of active timed states which p can still trigger in a interleaving manner at time stability. The function *trun* returns a process which takes the shape

$$((P \fatsemi run(X)) \llbracket X \rrbracket Q) \setminus X$$

where process P (the execution process) is defined by the function $trun'$ over the set of timed ready states, and the process Q (the recording process) is defined by the function *record* over the set of timed states. The set X is the set of coordination events, which are only communicated between P and Q, and not to the enactment. Similar to *timer*, since *trun* is defined in terms of $trun'$ and *record*, it will only cause the enactment to deadlock at the semantic level if and only if either $trun'$ or *record* deadlocks or they each have a set of refusals that are disjoint from each other after they cooperate on some trace.

case $trun'$: While by definition, p's enactment may either execute and terminate or interrupt and cancel timed states according to their syntactic descriptions in an interleaving manner when time stable, $trun'$ defines the process which takes the shape

$$|||\, i : I \bullet P_i$$

where I is the set of time-ready states (a non-empty subset of the active timed states), and for any $i \in I$, P_i is a process that will either execute or interrupt or may non-deterministically delayed state i according to i's timing information. After which P_i terminates. When all P_i of I terminate, $trun'$ terminates and by definition of *trun*, it will then behave as $run(X)$ where X is the set of internal coordination events for this set of timed-ready states.

By definition $trun'$ must delay execution of state i if i's minimum delay is $>_T zero_T$; it may delay i's execution if i's minimum delay is $=_T zero_T$ and maximum delay is not, and it must execute i if i's maximum delay is $=_T zero_T$. At any time it must cooperate with enactment on any interrupt of type *ierror* and *imessage* upon i, and it must interrupt i if it has a timed exception that expires at current duration.

Therefore $trun'$ will cooperate on at non-empty set of time ready states and will not cause deadlock to p's enactment. By definition of *record* and $timer'$, States which have been delayed will be guaranteed to be executed and terminated or interrupt and cancelled before p terminates.

case *record*: While $trun'$ cooperating with p's enactment, It synchronises with *record* over a set of coordination events, and engaging them recursively until the set of time-ready states is empty where each state is either terminated, cancelled or delayed. On engaging in these events, *record* can either insert a set of new active states into the set of currently active states if a time-ready state terminates successfully or is interrupted, or insert a time-ready state into the set of delayed states if that state has delayed its execution.

When each time-ready state has either terminated, cancelled or delayed, if the set of active states are empty, this means all time ready states have been delayed, the function then assumes the maximum of the shortest delay over the set of delayed states ss i.e.

$$(\mu\, s : ss \mid \forall\, t : ss \bullet s.ran.max \leq_T t.ran.max \bullet s.ran.max)$$

has passed and coordinates the set of delayed states by returning a process defined by *trun* over the set of delayed states. This ensures that any least one time-ready state is either executed and terminated or interrupt and cancelled, and consequently ensures *record* does not cause deadlock.

On the other hand if the set of active states is not empty, then *record* returns a process defined by *stable* over the the function $timer'$, a variant of the function *timer* and the set of active untimed states. By definition an iteration of the two phase functioning approach has completed and *record* will only cause deadlock if *stable* causes deadlock (already shown not the case) or the function $timer'$ causes deadlock.

case $timer'$: This is a variant of *timer* and returned a process when p has reached time stability more than once. By definition this function ensures a set of active timed states is coordinated along with the set of delayed state from the previous iteration. This function does not engage in any CSP events at the semantic level, and will always behave as *trun* after ordering, the function does not cause deadlock at the semantic level if and only if *trun* does not cause deadlock at the semantic level.

■