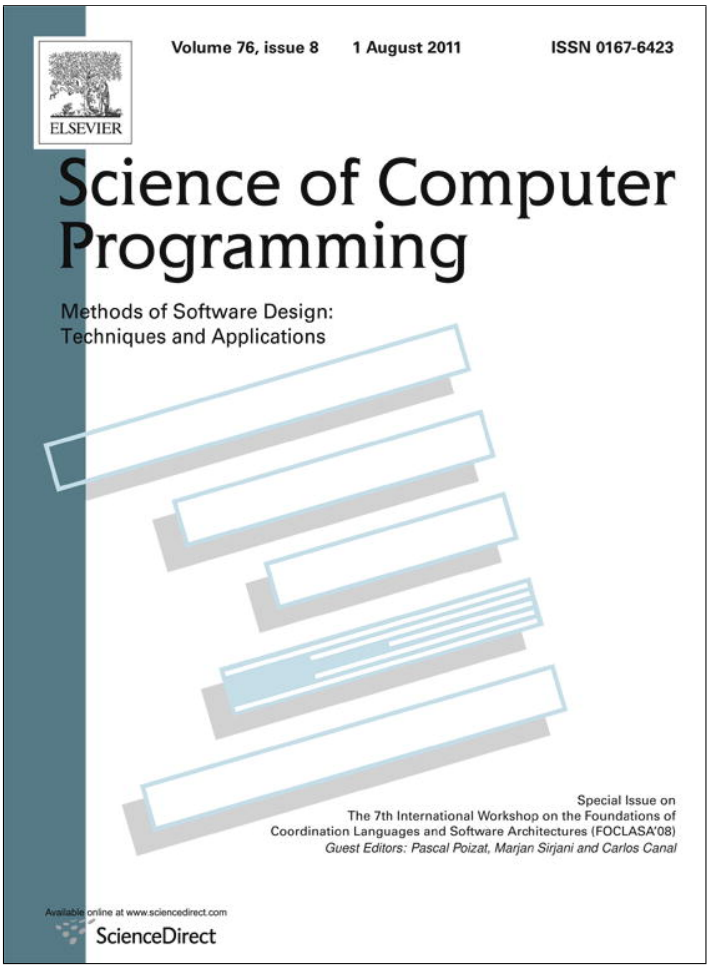

Science of Computer Programming 76 (2011) 633–650

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Formalisations and applications of BPMN

Peter Y.H. Wong *, Jeremy Gibbons

Computing Laboratory, University of Oxford, United Kingdom

ARTICLE INFO

Article history:
Received 12 January 2009
Received in revised form 13 July 2009
Accepted 21 September 2009
Available online 26 September 2009

Keywords:
Business process
CSP
Model checking
Property specifications
Refinement
Semantics
Timed behaviour
Workflow

ABSTRACT

We present two formalisations of the Business Process Modelling Notation (BPMN). In particular, we introduce a semantic model for BPMN in the process algebra CSP; we then study an augmentation of this model in which we introduce relative timing information, allowing one to specify timing constraints on concurrent activities. By exploiting CSP refinement, we are able to show some relationships between the timed and the untimed models. We then describe a novel empirical studies' model, and the transformation to BPMN, allowing one to apply our formal semantics for analysing different kinds of workflows. To provide a better facility for describing behaviour specification about a BPMN diagram, we also present a pattern-based approach using which a workflow designer could specify properties which could otherwise be difficult to express. Our approach is specifically designed to allow behavioural properties of BPMN diagrams to be mechanically verified via automatic model checking as provided by the FDR tool. We use two examples to illustrate our approach.

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, but the notation specification [19] does not contain a precise semantics and this means BPMN diagrams are ambiguous and cannot be verified for behavioural correctness. In this paper we describe two formalisations of the notation and also how to apply the notation to workflows that are normally beyond its scope.

Specifically the following results are presented in this paper:

- An untimed process semantics for BPMN in the process algebra CSP, allowing formal reasoning via CSP's process refinements and verification via model checking.
- A relative timed semantics for BPMN in CSP, giving a behavioural model for specifying timing information to BPMN diagrams. By choosing CSP as the common semantic domain, behavioural properties can be preserved between the untimed and timed semantic models.
- An empirical studies' model and the bi-directional transformation to BPMN, allowing one to apply the formal semantics and corresponding tool support for analysing workflows describing empirical studies e.g. clinical trials.
- A small specification language *PL*, giving an alternative pattern-based approach to behavioural specification for BPMN. The correctness of BPMN diagrams against a *PL* specification may be verified via model checking.

* Corresponding author.
E-mail addresses: peter.wong@comlab.ox.ac.uk (P.Y.H. Wong), jeremy.gibbons@comlab.ox.ac.uk (J. Gibbons).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.09.010

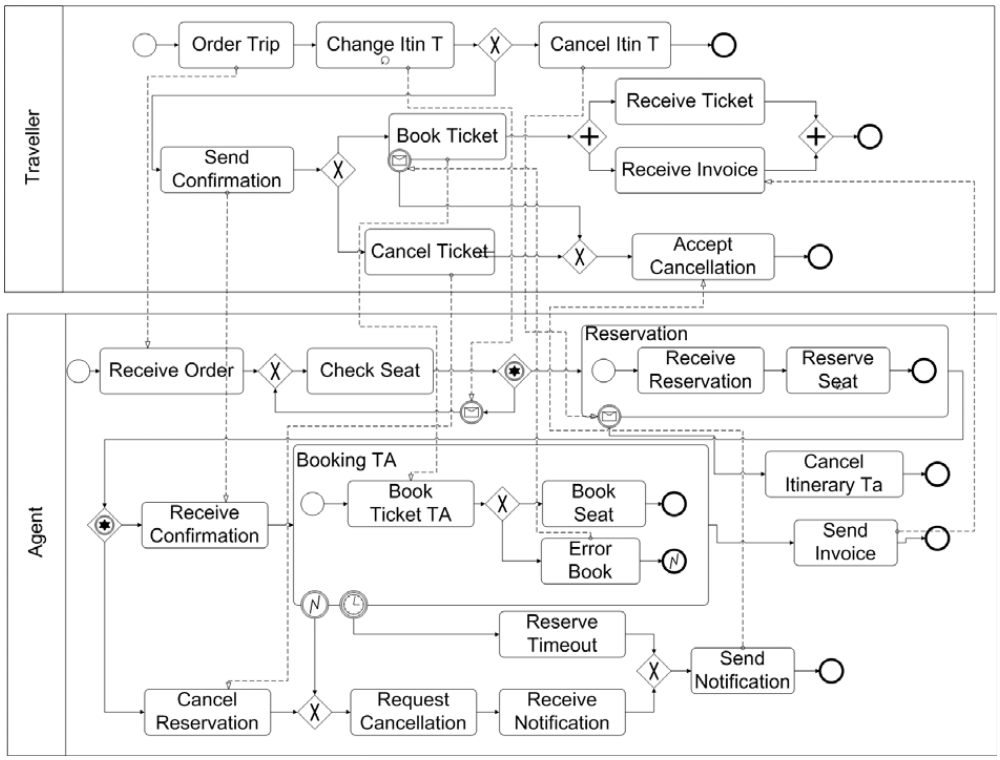

Fig. 1. Collaboration between the traveller and the travel agent.

1.1. Running example

As a running example for this paper, we consider the scenario where a traveller business process interacts with a travel agent business process. This is adapted from a well-documented example from the W3C [25]. Fig. 1 shows the BPMN diagram describing the collaboration between the traveller and the travel agent. An overview of BPMN notation is presented in Section 2, but an informal explanation of Fig. 1 follows.

1.1.1. Traveller

The traveller can order a trip by setting up an itinerary for airline ticket; thereafter she can reserve the seats and subsequently proceed with the booking, after which the travel agent and the airline will send her the statement and the ticket respectively. Specifically, after choosing her preferred travel plan (from a catalogue independently), the traveller may submit her choice to her travel agent via her local web service (e.g. web form) (*Order Trip*). The travel agent in return offers her an itinerary (not shown in Fig. 1). For various reasons this itinerary might not be satisfactory to the traveller and she may choose to change her itinerary (*Change Itin T*). The number of changes allowed is bounded and may be assumed to be determined by the particular policy of the travel agency and the airline. She may also decide not to take the trip, in which case she may cancel her order (*Cancel Itin T*).

In case she decides to accept the proposed itinerary, she may proceed to reserve this itinerary (*Send Confirmation*) and provide her credit card information to the travel agent. The travel agent then finalises the ticket reservation (carried out by the *Agent* business process in Fig. 1), after which the traveller may either confirm her ticket (*Book Ticket*) or cancel it (*Cancel Ticket*); if she chooses to cancel her ticket, she will receive a cancellation notification (*Accept Cancellation*). Also if an error occurs from the travel agent's system, the traveller would receive an error message and an exception will occur. The ticket will then be unreserved and she will receive a cancellation notification (*Accept Cancellation*).

After the traveller confirmed her ticket, she will receive the statement from the travel agent (*Receive Invoice*) and ticket from the airline (*Receive Tickets*). Note from the point of view of the traveller's workflow, it is not important from whom she receives her invoice and ticket, and in this example we do not present the airline business process.

1.1.2. Travel agent

The main purpose of the travel agent is to mediate interactions between the traveller who wants to buy airline ticket and the airline who supplies them; here for brevity we have omitted the airline system. Specifically, once the travel agent receives an initial order from the traveller (*Receive Order*), he needs to verify with the airline if the seats are available for the desired trip (*Check Seats*). In order to cater for the possibility of the traveller making changes to her itinerary, for every change of her itinerary (*Change Itin TA*), the travel agent verifies with the airline the availability of the seats (*Check Seats*). Once the traveller has agreed upon a particular itinerary (*Receive Reservation*), the travel agent reserves the seats for the traveller (*Reserve Seats*).

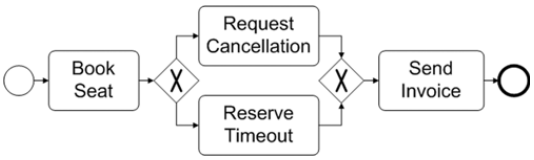

Fig. 2. A BPMN diagram capturing Requirement 4.

During the reservation period, modelled by the *Reservation* subprocess state, the traveller may cancel her itinerary, thereby releasing the reservation; this is modelled as a message exception flow, attached to the *Reservation* subprocess.

Once the reservation has been completed, the travel agent may receive a confirmation notice from the traveller (*Receive Confirmation*), in which case he receives the credit card information from the traveller (*Book Ticket TA*) and proceeds with the booking (*Book Seat*). The travel agent may also receive a cancellation of the reservation (*Cancel Reservation*), in which case he will request a cancellation from the airline (*Request Cancellation*), wait for a notification confirming the cancellation from the airline (*Receive Notification*), and send it to the traveller (*Send Notification*). During the booking phase, either an error (e.g. incorrect card information) (*Error Book*) or a time out (*Reserve Timeout*) may occur; in both cases, a corresponding notification confirming the cancellation will be sent to the traveller. Otherwise, a corresponding invoice on the booking will be sent to the traveller for billing (*Send Invoice*).

Here are some of the requirements the collaboration should meet.

1. Individual participants should be deadlock free.
2. Collaboration should be deadlock free.
3. If the traveller makes a cancellation, the travel agent must confirm that cancellation.
4. The travel agent must not allow any kind of cancellation after the traveller has booked her ticket, if an invoice is to be sent to the traveller.

1.2. Semantics

Our approach is to give two compositional semantic definitions to a subset of BPMN, informal description and the abstract syntax of this subset are discussed in Sections 2.1 and 3 respectively. The first one is a purely untimed model [26] and we have chosen the process algebra CSP [22] to be the semantic domain. This allows one to compare the behaviour of diagrams and consequently infer suitable refinements over them. The advantage of our approach is that business process developers could now specify abstract properties using the same notation as for modelling their business processes. Specifically we define a semantic function, which takes the syntax of a given BPMN diagram and returns a CSP process describing its untimed behaviour. We present an overview of this semantic model in Section 4.

The untimed model provides the facility for verifying both safety (with CSP's traces refinement) and liveness (with failures refinement) properties. However, the assumption made in the model due to abstraction is that non-interacting concurrent activities in a business process *interleave*, that is, they may occur in any order. This assumption might not be satisfactory in some cases where the timing information becomes a factor in governing behaviour. In this case our second model [27] augments the first one by introducing *relative timing* information, this allows one to model concurrent activities under temporal constraints. We have chosen CSP to be the common semantic domain for both models, and this enables us to show some properties relating these two models based on CSP refinements. The augmentation and its relationship with the untimed process semantics are presented in Sections 5.1 and 5.2 respectively. Section 5.2 also revisits the example in Fig. 1, and discusses how our models may be used to verify against the business process's requirements.

1.3. Specification

Our semantic models leverage the refinement orderings that derive from CSP's denotational semantics, allowing BPMN to be used for specification as well as modelling of business processes. However, the expressiveness of BPMN is *strictly less than* that of CSP, and consequently some behavioural properties, against which developers might be interested to verify their business processes, might not be easy or even possible at all to capture in BPMN. Consider Requirement 4 of the ticket reservation example (Fig. 1): assuming the process *Agent* denotes the process semantics of the travel agent, one might attempt to draw a BPMN diagram like the one shown in Fig. 2 to express the negation of the property, and prove the satisfiability of *Agent* by showing this diagram does not failures refine the process $Agent \setminus N$ where N is the set of CSP events that are not associated with tasks *Book Seat*, *Request Cancel*, *Request Timeout* and *Send Invoice*. However, while this behavioural property should also permit other behaviours such as task *Request Cancel* being performed before task *Book Seat*, it could be difficult to specify all these behaviours in the same BPMN diagram. Since BPMN is a modelling notation for describing the *performance* of behaviour, in general it is difficult to use it to specify liveness properties about the *refusal* of some behaviour within a context while asserting the *availability* of it outside the context. In Section 6 we present a complementary approach in which we consider a pattern-based approach to expressing behavioural properties. We describe a property specification language *PL* for capturing a generalisation of Dwyer et al.'s Property Specification Patterns [5], and present a translation from *PL* into a bounded, positive fragment of linear temporal logic, which can then be automatically

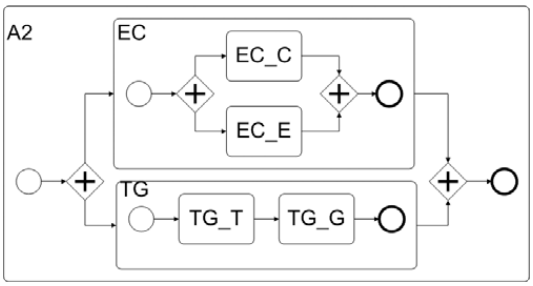

Fig. 3. A set of clinical interventions.

translated into CSP for simple refinement checking. We demonstrate its application by revisiting the running example shown in Fig. 1.

1.4. Empirical studies

While BPMN is becoming a standard for modelling business processes and we could demonstrate the application of our models with conventional business process example (as shown in Fig. 1), we consider extending its application scope by investigating an alternative class of workflows. Specifically we propose a declarative model for *empirical studies*, and a method for transforming this model into BPMN so that one may leverage BPMN's graphical expressiveness and newly-defined formal semantics [28].

Here *empirical studies* are plans consisting of a series of scientific procedures interleaved with a set of observations performed over a period of time; these observations may be manually performed or automated, and are usually recorded in a *calendar schedule*. An example of a long running empirical study is a clinical trial, where observations, specifically case report form submissions, are performed at specific points in the trial. In such examples, observations are interleaved with clinical interventions on patients; precise descriptions of these observations and interventions are then recorded in a patient study calendar.

For example, in a clinical study it is important that interventions are carried out safely and effectively, and often interventions *must* satisfy a set of oncological safety principles [8]. Fig. 3 describes a set of clinical intervention $A2$ that might be assumed to be part of a more complex clinical procedure. Its constructions will be described later on in the paper. Below we show the schedule of each drug administration; we have omitted dosage for simplicity.

- *EC_C* – Cyclophosphamide, every 14 to 20 days
- *EC_E* – Epirubicin, every 18 to 21 days
- *TG* – Paclitaxel, every 5 to 10 days followed by Gemcitabine, up to 10 days later.

The safety principle *Sequencing* ensures that an intervention "order(s) (essential) actions temporally for good effect and least harm". Here we are interested in the following particular instance of this principle for interventions $A2$:

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

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

This approach is presented in Section 7, in which we give an overview of our declarative model, for recording empirical studies such as clinical trials, and show how this model may be transformed into BPMN using a transformation function implemented in Haskell [9], we also discuss the example shown in Fig. 3 in this section.

1.5. Related work

Results described in Sections 2–4 are based on the work reported in [26,29]; Section 5 is based on [27]; Section 7 is based on [28]; and parts of Section 6 are based on [30]. The restrictive disjunctive normal form and the more complete treatment of the bounded existence pattern, both in Section 6, are previously unpublished.

2. Notations

2.1. BPMN

States in our subset of BPMN, shown in Fig. 4, 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

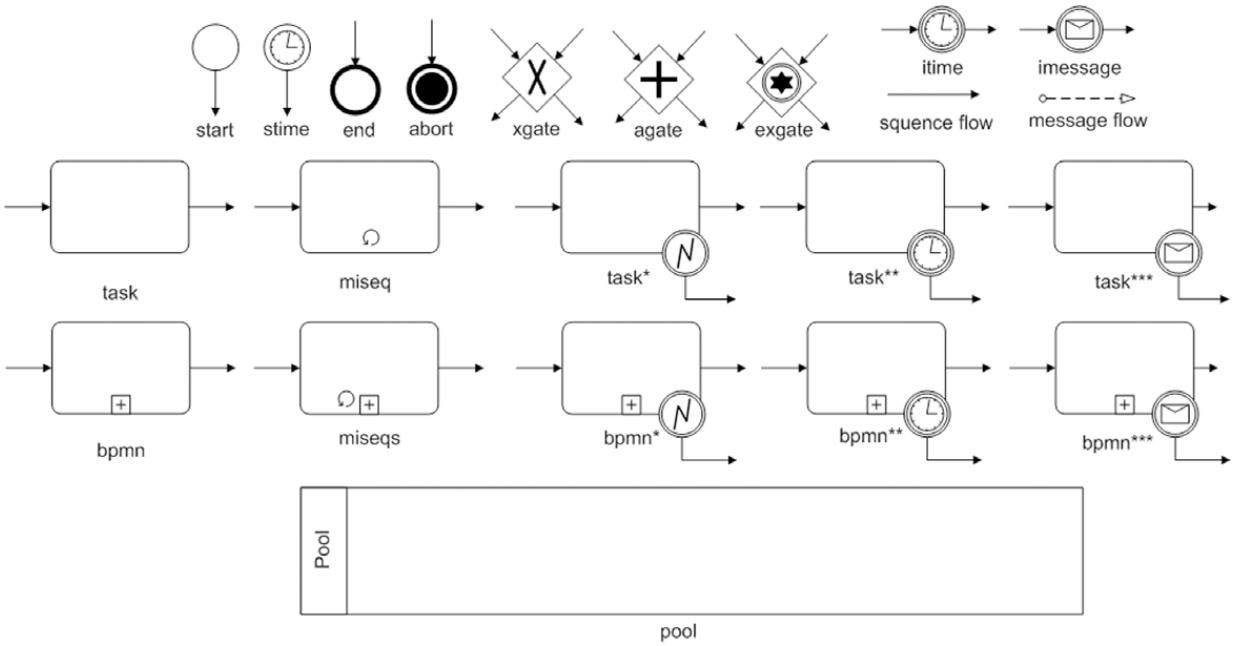

Fig. 4. States of BPMN diagrams.

be either incoming to or outgoing from a state and have associated guards; an exception sequence flow, depicted by the states labelled *task**, *bpmn**, *task*** and *bpmn***, represents an occurrence of error within the state. While sequence flows represent control flows within individual *local* diagrams, message flows represent unidirectional communication between states in different local diagrams. A *global* diagram hence is a collection of local diagrams connected via message flows. Note for brevity we only consider a subset of BPMN, which is used to model the examples in this paper.

In Fig. 4, there are two types of start state, *start* and *stime*. A *start* state models the start of the business process in the current scope by initiating its outgoing transition; it has no incoming transition and only one outgoing transition. The *stime* state is a variant start state; it initiates its outgoing transition when a specified duration has elapsed. There are also two types of intermediate state, *itime* and *imessage*. An *itime* state is a delay event; after its incoming transition is triggered, the delay event waits for the specified duration before initiating its outgoing transition. An *imessage* state is a message event; after its incoming transition is triggered, the message event waits until a specified message has arrived before initiating its outgoing transition. Both types of state have a maximum of one incoming transition and one outgoing transition.

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

Our subset of BPMN contains three types of decision state, *xgate*, *exgate* and *agate*. Each of them has one or more incoming sequence flows and one or more outgoing sequence flows. An *xgate* state is a data-based exclusive choice gateway, it accepts one of its incoming flows and takes one of its outgoing flows based on the evaluation of a boolean expression using process data [19, page 71]. An *exgate* state, on the other hand, is an event-based exclusive choice gateway, it accepts one of its incoming flows and takes one of its outgoing flows based on events, such as the receipt of a message, that occurs at that point in the process [19, page 75]. An *agate* state is a parallel gateway, which waits for all of its incoming flows before initialising all of its outgoing flows.

A *task* state describes an atomic activity, and has exactly one incoming and one outgoing transition. It takes a unique name for identifying the activity. In the environment of the timed semantic model, each atomic task must take a positive amount of time to complete. A *bpmn* state describes a subprocess state. It is a business process by itself and so it models a flow of BPMN states. In this paper, we assume all our subprocess states are expanded [19]; this means we model the internal behaviours of the subprocesses. The state labelled *bpmn* in Fig. 4 depicts a collapsed subprocess state where all internal details are hidden; this state has exactly one incoming and one outgoing transition.

Also in Fig. 4 there are graphical notations labelled *task**, *bpmn**, *task***, *bpmn***, *task**** and *bpmn****, which depict a task state and a subprocess state with an exception sequence flow. There are three types of exception associated with task and subprocess states in our subset of BPMN states. Both states *task** and *bpmn** are examples of states with an *ierror* exception flow that models an interruption due to an error within the task or subprocess state; the states *task*** and *bpmn*** are examples of states with a timed exception flow, and model an interruption due to an elapse of the specified duration; the states *task**** and *bpmn**** are examples of states with a message exception flow, and model an interruption upon receiving the specified message. Each task and subprocess state can have a maximum of one timed exception flow, although it may have multiple error and message exception flows.

Each task and subprocess may also be defined as *multiple instances*. There are two types of multiple instances in BPMN, sequential and parallel. While our semantics captures both types, in this paper we only consider the sequential type, whose task and subprocess are specified by the state types *miseq* and *miseqs* respectively. A sequential multiple instances repeats its task (subprocess) in sequence.

The graphical notation *pool* in Fig. 4 forms the outermost container for each local diagram, representing a single business process; only one execution instance is allowed at any one time. Each local diagram contained in a pool can also be a participant within a business collaboration (global diagram) involving multiple business processes.

2.2. Z

Throughout this paper we use the Z notation [31] to provide an abstract syntax for BPMN. Here we give a brief overview of the notation.

The Z notation 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:

$$\begin{array}{|l} \textit{Name} \\ \hline \textit{declaration} \\ \hline \textit{constraint} \\ \hline \end{array}$$

or equivalently

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

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.

$$\textit{Type} ::= \textit{element}_1 \mid \ldots \mid \textit{element}_n \qquad [\textit{Type}] \qquad \textit{symbol} == \textit{term}.$$

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

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

2.3. CSP

In CSP [22], 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.

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

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

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 $A \cup B$ can occur without the cooperation of P and Q respectively. We write $|||\ i : I \bullet P(i)$, $|[A]|\ i : I \bullet P(i)$ and $\|\ i : I \bullet A(i) \circ P(i)$ 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 [22]. 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 $refusals(P)$ is P's initial refusals. A failure therefore is a pair (s, X) where $s \in traces(P)$ is a trace of P leading to a stable state and $X \in refusals(P/s)$ where P/s represents process P after the trace s. We write $traces(P)$ and $failures(P)$ as the set of all P's traces and failures respectively.

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

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

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

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

While traces only carry information about *safety* conditions, refinement under the stable failures model allows one to make assertions about a system's *safety* and *availability* properties. These assertions can be automatically proved using a model checker such as FDR [6], 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

In this section we describe the abstract syntax of BPMN using Z notation [31]. For reasons of space, this section provides partial definitions of BPMN's abstract syntax; readers may refer to our longer papers [26,27] for full definitions.

We first introduce some maximal sets of values to represent constructs such as *transitions*, *task* and *message flows*, defined as Z basic types:

$$[PName, Task, Trans, Msgflow]$$

where *PName* is the set of process's names. We also derive subtypes *BName* for subprocess names and *PLName* for pool names, and they partition *PName*.

$$BName, PLName : \mathbb{P}\, PName$$
$$\langle BName, PLName \rangle \text{ partition } PName$$

Note our relative timed model defines the semantics of BPMN timed events describing only time cycles (duration) and not absolute time stamps. We define schema type *Time* to record each duration; this schema models a strictly positive subset of the six-dimensional space of the XML schema data type *duration* [32, Section 3.2.6].

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

Each type of state shown in Fig. 4 is defined using the free type *Type* where each of its constructors describes a particular type of state. For example, the type of an atomic task state is defined by *task t* where t is a unique name that identifies that task state. Below is the partial definition.

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

According to the BPMN specification [19], each state type has other associated attributes describing its properties; our syntactic definition has included only some of these attributes. For example, the number of loops of a sequence multiple instance state type is recorded by the natural number in the constructor function *miseq*. In this paper we call both sequence flows and exception flows 'transitions'; states are linked by transition lines representing flows of control. Each atomic task

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

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

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

$$State \mathrel{\widehat=} [type : Type;\ in, out : \mathbb{P}\, Trans;\ err : \mathbb{P}(Type \times Trans);\ ran : Range]$$

Note in our untimed process semantics, the schema's component *ran* is not considered.

Each BPMN diagram encapsulated by a *pool* is a local diagram and represents an individual business participant in a collaboration, built up from a well-configured finite set of well-formed states [26]. While we associate each local diagram with a unique name, a global diagram, representing a business collaboration, is built up from a finite set of names, each associated with its local diagram; we also associate each global diagram with a unique name. We define the specification environment as a set of mappings from diagram names to their states. For example the local environment is defined as the abbreviation $Local == BName \pfun \mathbb{P}\, State$.

4. Process semantics

4.1. Semantic function

Our semantic function *bsem* 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 and synchronising on its own alphabet to ensure the correct order of control flow.

$$bsem : PName \pfun Local \pfun Process$$

Here we let [*Process*, *Event*] be basic types. Specifically *bsem* returns a process in this form,

$$(\|\ i : I \bullet A(i) \circ P(i)) \setminus S \tag{1}$$

where set I indexes states in the diagram, the process $P(i)$ denotes the semantics of the state identified by i, and the set $A(i)$ is the alphabet of the process $P(i)$. Set S denotes the set of events associated to all transitions in the diagram; since they should not be affected by the environment, we internalise them via the hiding operation. The definition of these sets and processes are described throughout this section. For brevity, we omit formal definition of the semantic function. Full definition may be sought in our technical report [26].

The alphabet of a state is the set of events associated with its state type (*Type*), transitions (*Trans*) and message flows (*Msgflow*). We first define the functions α_{trans} and α_{mge}, which take a set of transitions and message flows and return their corresponding set of events. We also define the function ϵ_{task}, which takes a task name and returns a CSP event denoting the execution of the task.

Next we define the function α_{state}, which takes a local environment of type *Local* and the state we are interested in and returns a set of events of type *Event*. For example, for the *Order Trip* task state, identified by *OrderTrip*, of the traveller participant in Fig. 1, its alphabet is

$$A(OrderTrip) = \{init.s1, start.OrderTrip, init.s2, mge.m1\}$$

assuming $s1$ and $s2$ identify the incoming and outgoing transitions of the state, and $m1$ identifies its outgoing message flow. In this paper we use events $init.i$, $start.w$ and $mge.m$ to denote transition i, message flow m and the execution of task w respectively. This is convenient as we may now specify the set of all events denoting transitions $S = \{\!|init|\!\}$.

Similarly one may define the function α_{proc} to map each diagram to the set of all possible events performed by the process describing an individual *local* diagram's behaviour.

To capture the behaviour of sequence flow looping [19], we model the behaviour of each state in a diagram recursively, where each recursive call denotes an iteration of the state's execution. The behaviour of each iteration is defined by the sequential compositions of the behaviour of performing its incoming transitions and message flows, state type, and outgoing transitions and message flows. For example, the following process denotes the semantics of the *Order Trip* task state.

$$\begin{aligned} OrderTripProc \ = \ & init.s1 \rightarrow Skip \fatsemi start.OrderTrip \rightarrow Skip \fatsemi \\ & mge.m1 \rightarrow Skip \fatsemi init.s2 \rightarrow Skip \fatsemi OrderTripProc \end{aligned} \tag{2}$$

We observe that the processes corresponding to a start, an end or an abort state are non-recursive. This is because they have either no incoming or no outgoing transitions, that is they cannot be part of a sequence flow loop. For example, the following process denotes the semantics of the start state of the traveller process.

$$StartProc = init.s1 \rightarrow Skip \tag{3}$$

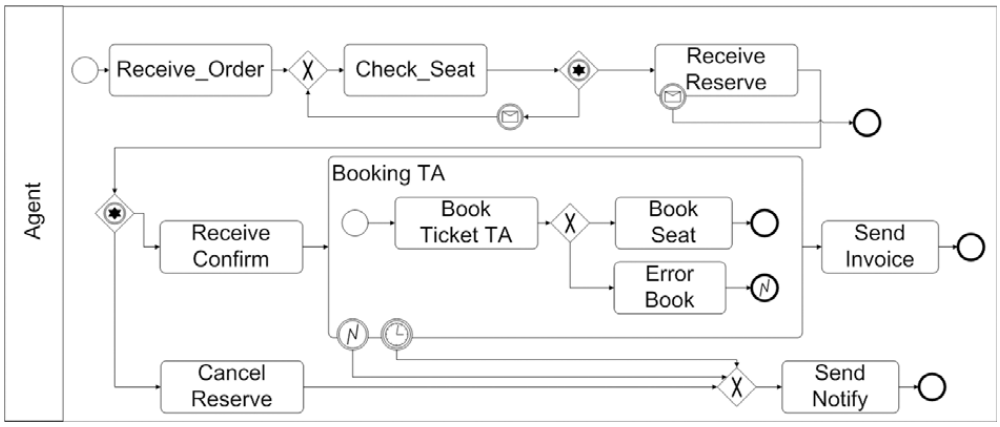

Fig. 5. An abstracted travel agent process.

However, to capture behaviour of completion and termination, denoted by end and abort states respectively, we introduce a CSP event for each *end* and *abort* state and this event will be communicated to all the other states contained in the process. For example, the following process denotes the semantics of the end state of the traveller process,

$$P(End1) = (init.s8 \rightarrow Skip \fatsemi fin.1 \rightarrow Skip) \Box fin.2 \rightarrow Skip \Box fin.3 \rightarrow Skip \tag{4}$$

where the events $fin.i$ signals the completion of the process by the end state i, and $End1$ identifies the end state. Here we use natural number to identify each end state in a process. Consequently we extend the semantics of processes (2) and (3) as follows,

$$\begin{aligned} P(OrderTrip) &= OrderTripProc \fatsemi P(OrderTrip) \Box CompleteProc \\ P(Start) &= (StartProc \fatsemi CompleteProc) \Box CompleteProc \end{aligned}$$

where $CompleteProc = (\Box\, n : \{1, 2, 3\} \bullet fin.n \rightarrow Skip)$, and $Start$ identifies the start state.

We have implemented the semantics described in this section as a prototype tool using the functional programming language Haskell. Readers may find a copy of the implementation from our web site [18]. The tool inputs an XML serialised representation of BPMN diagram from the JViews BPMN Modeler [11], and translates it into an ASCII file containing CSP processes representing its behaviours expressed in machine-readable CSP [22].

We now revisit our example in Fig. 1. We assume CSP processes *Traveller* and *Agent* denote the semantics of the traveller and travel agent BPMN processes respectively. Initially we would like to check Requirement 1 in Section 1.1, that is, that both traveller and travel agent processes are *individually* deadlock free. Semantically we verify this by checking if their process semantics refine process $DF = (\sqcap\, i : \Sigma \bullet i \rightarrow DF) \sqcap Skip$, which characterises deadlock freedom in the stable failures model. This refinement may be mechanically checked using FDR.

$$DF \sqsubseteq_{\mathcal{F}} Traveller \wedge DF \sqsubseteq_{\mathcal{F}} Agent \tag{5}$$

Requirement 2 is that the collaboration should be deadlock free. Semantically this means the parallel combination of *Traveller* and *Agent*, synchronising on their own alphabet, should refine *DF*. For efficiency reasons, one might consider abstracting the participant processes into *equivalent processes for collaboration*. For example, tasks *Cancel Itin TA*, *Reserve TimeOut*, *Request Cancel*, *Receive Notify* and *Reserve Seat* (in the *Reservation* subprocess of the travel agent in Fig. 1), do not interact with the traveller process. From the point of view of the traveller process we could define an equivalent travel agent process. This is shown in Fig. 5. Let process *AgentR* denote the semantics of the travel agent in Fig. 5, by showing the following assertion holds via model checking we prove these two business processes are equivalent from the point of view of the traveller process.

$$AgentR \sqsubseteq_{\mathcal{F}} Agent \setminus NonIA \wedge Agent \setminus NonIA \sqsubseteq_{\mathcal{F}} AgentR \tag{6}$$

Formally this equivalence may be generalised to allow one to construct a *compatible class* under our formal notion of *compatibility*, whose formal definition may be found in our technical reports and papers [29,26]. Now we may check Requirement 2 by mechanically verifying the following refinement.

$$DF \sqsubseteq_{\mathcal{F}} Traveller \,[\![\, \alpha Traveller \mid \alpha AgentR \,]\!]\, AgentR \tag{7}$$

In fact this requirement is not satisfied, as the collaboration deadlocks after the *Traveller* performs the trace $\langle start.OrderTrip, start.ChangeItinT, start.ChangeItinT\rangle$ while *AgentR* performs trace $\langle start.ReceiveOrder, start.CheckSeat, start.Reservation\rangle$. Referring back to Fig. 1, we can see the travel agent could deny the traveller's wish to change her itinerary. To correct this workflow, the traveller must notify the travel agent after her last change to her order.

4.2. Refinements of diagrams

The motivation behind this model is to define the following refinement ordering upon BPMN diagrams. We introduce two types of refinement based on CSP's stable failures model and the hierarchical composition of BPMN diagrams. We first introduce the notion of hierarchical refinement, where the specification diagram is an abstraction of the implementation diagram via collapsing subprocess states.

Definition 4.1 (*Hierarchical Refinement*)**.** Given two BPMN diagrams, described by the names n_1 and n_2, and the specification environment l_1 and l_2 respectively, diagram n_1 **hierarchically refines** diagram n_2 iff

$$bsem\ n_2\ l_2 \sqsubseteq_{\mathcal{F}} (bsem\ n_1\ l_1 \setminus S)$$

where S is the set of events corresponding to the alphabet of states that are contained in the subprocess states, which are defined in diagram n_1, and have been abstracted by collapsing them into task states in diagram n_2.

This refinement ordering semantically relates different levels of abstraction between BPMN diagrams. Now we can introduce the notion of hierarchical independence upon behavioural specification.

Definition 4.2 (*Hierarchical Independence*)**.** A diagram n_1 in the environment l_1 is a **hierarchically independent specification** of diagram n_2 in the environment l_2 iff for all names m and specification environments k, the following expression holds:

$$bsem\ m\ k \sqsubseteq_{\mathcal{F}} (bsem\ n_2\ l_2 \setminus S) \Rightarrow bsem\ n_1\ l_1 \sqsubseteq_{\mathcal{F}} bsem\ m\ k$$

where S is the set of events corresponding to the alphabet of states that are contained in the subprocess states, which have been collapsed.

Hierarchical independence allows us to reason about a BPMN diagram against a behavioural specification by verifying a more abstract version of that diagram against the specification. However, sometimes it is not only convenient to hide details of subprocess states, but it is necessary to also abstract details which are irrelevant to the behavioural property we are interested in.

Definition 4.3 (*Partial Refinement*)**.** Given two BPMN diagrams, described by the names n_1 and n_2, and the specification environments l_1 and l_2 respectively, diagram n_1 **partially refines** diagram n_2 iff

$$bsem\ n_2\ l_2 \sqsubseteq_{\mathcal{F}} (bsem\ n_1\ l_1 \setminus S)$$

where S is the set of event corresponding to the alphabet of *all* states that have been abstracted.

In our example, the process *Agent* is also a partial refinement of *AgentR*. These relationships allow a business process developer to focus on the specification of part of the diagram.

5. Relative timing

5.1. Semantic function

We now give an overview of our timed model [27] which takes a syntactic description of a global diagram, describing a collaboration, and returns the CSP process that models the timed behaviour of that diagram. That is, the function takes one or more *pool* states, each encapsulating a local diagram representing an individual participant within a business collaboration, and returns a parallel composition of processes each corresponding to the timed behaviour of one of the individual participants. For reasons of space we use the example of a clinical trial in Fig. 3 to illustrate this semantics. We revisit this example in Section 7.

For each local diagram, the relative timed semantics is the partial interleaving of two processes defined by an *enactment* and a *coordination* function. The enactment function returns the parallel composition of processes, each corresponding to the untimed aspect of a state of the local diagram; this is essentially our process semantics of local diagrams defined in the previous section. The coordination function returns a single process for coordinating that diagram's timed behaviour; it essentially implements a variant of the *two-phase functioning approach* adopted by real-time systems and timed coordination languages [12]. Our timed model permits automatic translation, requiring no user interaction. We will now give a brief overview of the coordination function; again for reasons of space we only present function types accompanied with informal descriptions. The complete formal definition of both the enactment and coordination functions may be found in our technical report [27].

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

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

$$clock : PName \pfun Local \pfun Process$$

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

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

The function *stable* is a higher order function; it takes some function f (for example, constructed from the function *timer* below) and a set of *active* states, and returns a process, which recursively enacts all *untimed* active states until the local diagram is *time-stable* [27] i.e. when all active states of a local diagram are timed. Going back to the example in Fig. 3, states *EC_C*, *EC_E*, *TG_T* and *TG_G* are timed and when the function *stable* is applied to the syntax of the diagram initially, the process it returns will enact all states according to the sequence flows until the set of active states are $\{ EC\_C, EC\_E, TG\_T \}$, that is the diagram is time-stable. After this the function behaves as defined by the function f; in the definition of *clock*, f is the function *timer* applied with its first four arguments where the third and fourth arguments are initially empty.

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

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

- creates a subset of active timed states that has the shortest delay, we denote these states as *time-ready* [27], in our example after the first time being time-stable, the only time-ready state is state *TG_T*, which has the minimum delay of 5 days;
- subtracts the shortest delay from the delay of all timed states that are not time-ready to represent that at least that amount of time has passed, in our example, as *TG_T* is the time-ready, other active timed states *EC_C* and *EC_E* will have delays 9 to 15 days and 13 to 16 days respectively.

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

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

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

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

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

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

2. if all time-ready states have either been enacted or delayed, then this completes a cycle of timed coordination, and the process then behaves as defined by *stable* and proceeds with the next cycle;
3. if there exist time-ready states that have not been enacted or delayed, *record* waits for coordination events from the execution process.

In our example when the time-ready state *TG_T* is applied to *record*, the process it returns either waits for *TG_T* to be enacted or delayed. If *TG_T* is enacted, it behaves as *stable* over a empty set of untimed states and the set of timed states $\{ EC\_C, EC\_E, TG\_G \}$ since the immediately succeeding state of *TG_T* is *TG_G*, which is a timed state (rule 2). Otherwise it will also behave as *stable* since the set of currently active states are not empty (rule 2). The coordination terminates after it enacts an *end* state of the top level diagram. A complete definition of the semantic function may be found in our longer paper [27].

5.2. Analysis

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

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

One of the consequences of using a common semantic domain for both timed and untimed models is that we can transfer certain behavioural properties from the untimed to the timed world. We achieve this by showing for any timed variation of any local diagram, the timed coordination process is a *responsive plug-in* [21] to the enactment process. Informally process Q is a responsive plug-in to P if Q is prepared to cooperate with the pattern set out by P for their shared interface. We now formally present Reed et al.'s definition of the binary relation *RespondsTo* over CSP processes using the stable failures model.

Definition 5.2. For any processes P and Q where there exists a set J of shared events, Q is a **responsive plug-in** to P, denoted as Q *RespondsTo* P iff for all traces $s \in \text{seq}(\alpha P \cup \alpha Q)$ and event sets X

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

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

Proposition 5.3 (*Responsiveness*)**.** *For any local diagram p under the relative timed model where its enactment and coordination are modelled by processes E and T respectively, T RespondsTo E.*

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

A direct consequence of Proposition 5.3 is that deadlock freedom is preserved from the untimed to the timed setting. So taking the definition of process *DF* from Section 4,

Proposition 5.4 (*Deadlock Freedom Preservation*)**.** *For any process P, modelling the behaviour of an untimed local diagram, and for any process Q modelling the behaviour of a timed variant of that diagram,*

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

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

Definition 5.5 (*Time Independence*)**.** A behavioural specification process S is **time-independent** with respect to some untimed local diagram whose behaviour is given by process P iff for any process Q modelling the behaviour of a timed variant of that diagram,

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

As a consequence of Propositions 5.3 and 5.4 and refinements over $\mathcal{T}$, we can generalise time-independent specifications by the following result.

Proposition 5.6. *A specification process S is time-independent with respect to some untimed local diagram whose behaviour is given by the process P iff*

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

where $\mathit{traces}(P)$ *is the set of possible traces of process P and* $\mathit{deadlocks}(P)$ *is the set of traces on which P can deadlock.*

As well as describing individual business processes, BPMN may also be used to specify business collaboration where more than one business process (participant) communicates via message flows; informally we say a participant is compatible with respect to a collaboration if it cooperates on the pattern of message flow communications. Similar to the notion of compatibility defined over the untimed model [26,29] and illustrated in the previous section, we formalise *time-compatibility* using CSP's responsiveness.

Definition 5.7 (*Time-Compatibility*)**.** Given some collaboration described by the CSP process,

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

where n ranges over $\mathbb{N}$ and M is the set of events corresponding to the message flows between its participants, whose **timed behaviour** are modelled by the processes T_i, participant T_i is time-compatible with respect to the collaboration C iff $\forall j : \{1 \ldots n\} \setminus \{ i \} \bullet T_i \ \mathit{RespondsTo} \ T_j$

One result of formalising compatibility under our timed semantics is that, since responsiveness is *refinement-closed* under $\mathcal{F}$ [21], time-compatibility is also refinement-closed.

Proposition 5.8. *Given that the participants P_i, where i ranges over some index set, are time-compatible in some collaboration C, their refinements under $\mathcal{F}$ are also time-compatible in C.*

However, refinement closure does not capture all possible compatible participants within a collaboration. Specifically, for each participant in a collaboration there exists a *time-compatible class* of participants of which any member may replace it and preserve time-compatibility. This class may be formalised via the stable failures equivalence. This notion augments our earlier definitions in the untimed setting [26].

Definition 5.9 (*Time-Compatible Class*)**.** Given some local diagram name p and its specification l, we define its time-compatible class of participants $cf_T(p, l)$ axiomatically as a set of pairs where each pair specifies a BPMN diagram by its environment and the name which identifies it.

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

where $\mathit{mgs}(q, m) = (\alpha_{\mathit{proc}}\ q\ m \setminus \mathit{mg}\ q\ m)$ and function *mgs* takes a description of a local diagram and returns a set of CSP events corresponding to the message flows of that diagram.

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

Definition 5.10 (*Characteristic Participant*)**.** Given the time-compatible class cp of some participant p, specified in some environment l, for some collaboration c, the characteristic participant of cp, specified by a pair of name and the environment, is given by the function $char_T$ applied to cp.

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

The following result is a direct consequence of Proposition 5.8, and Definitions 5.9 and 5.10.

Proposition 5.11. *If a characteristic participant p of a time-compatible class cp, specified in some environment l, is time-compatible with respect to some collaboration c, then all participants in cp are also time-compatible with respect to c.*

6. Property specifications

Our semantics provide a natural refinement ordering upon BPMN diagrams, allowing one to use BPMN for both specification and modelling and as a result, promoting both compositional and stepwise development of business processes. However, the expressiveness of BPMN is *strictly less than* that of CSP and as a result, some behavioural properties about business processes may not be easy to capture in BPMN. This is illustrated in Section 1.3, where we consider Requirement 4 of our case study. This section gives an overview of a complementary approach [30], in which a CSP formalisation of a *generalisation* of Dwyer et al. Property Specification Patterns (PSP) [5]; PSP are intended to describe the essential structure of commonly occurring requirements on the permissible behaviour in a finite state model of a system. We generalise PSP

to capture admissible sequences of *patterns of behaviours*, rather than individual events, within a scope of a pattern. In particular our approach may be summarised as follows: we define a small property specification language *PL*, based on the generalised patterns, for describing behavioural properties, and then provide a function that returns an expression in the *bounded, positive fragment* of linear temporal logic (*BTL*) that specifies the behaviour properties; we then translate the given *BTL* expression into its corresponding CSP process based on Lowe's characterisation [14]; using this, one may check whether a workflow system behaves according to a property specification.

BTL. The semantics of *BTL* extends the original *LTL* for capturing both *performance* and *availability* of behaviour. For example, while atom formula a denotes the event a is available to be performed initially, and no other events may be performed, the formula `available` a denotes the event a must not be refused initially, and other events may be performed.

Refusal traces. This formalisation requires a finer CSP semantics, refusal traces $\mathcal{RT}$ [17], than the standard stable failures as it has been demonstrated that $\mathcal{F}$ is not adequate for capturing this interpretation of temporal logics [14]. In $\mathcal{RT}$, each CSP process may be denoted as a set of refusal traces; each refusal trace is an alternating sequence of refusal information and events. More precisely, a refusal trace takes the form,

$$\langle X_1, a_1, X_2, a_2, \ldots, X_n, a_n, \Sigma \rangle$$

where each X_i is a refusal set, and each a_i is an event. This test represents that the process can refuse X_1, perform a_1, refuse X_2, perform a_2, etc. In this particular example the refusal trace finishes by refusing Σ (the set of all possible events), i.e. deadlocking. We write $\mathcal{RT}[\![P]\!]$ for the refusal traces of P and refinement in the $\mathcal{RT}$ is then defined as $Spec \sqsubseteq_{\mathcal{RT}} P \Leftrightarrow \mathcal{RT}[\![Spec]\!] \supseteq \mathcal{RT}[\![P]\!]$. Currently FDR [6] is being extended to include the checking of refinement in this model.

6.1. Patterns of behaviour

To capture patterns of behaviour for pattern-based specification, *PL* contains a sub-language *SPL*, which assists developers to construct BPMN-based patterns of behaviour. *SPL* contains a subset of CSP with the addition of a new *non-deterministic interleaving* operator ($\sqcap\!\sqcap$). Informally the term $P \sqcap\!\sqcap Q$ communicates events from both P and Q, but unlike CSP's interleaving, our operator chooses them non-deterministically. Here we present the step law governing the operator in the form of CSP's algebraic laws [22]: if $P = p \rightarrow P'$ and $Q = q \rightarrow Q'$ then $P \sqcap\!\sqcap Q = (p \rightarrow (P' \sqcap\!\sqcap Q)) \sqcap (q \rightarrow (P \sqcap\!\sqcap Q'))$.

SPL contains an atomic term *End*, which has empty semantics over $\mathcal{RT}$ and that is a *unit* over $\sqcap\!\sqcap$. This operator is particularly useful when reasoning about a BPMN process over the timed model, that is, when concurrent activities are constrained due to timing information. In our formalisation [30], *SPL* is translated into *BTL* so that it could be used inside a *BTL* expression of a property pattern. In particular, all expressions translated from *SPL* are characterised by atomic formulae over $\vee$, $\wedge$ and $\bigcirc$ ('next' operator in temporal logic), and as such each *BTL*-translation of *SPL* may be captured by the grammar $E ::= a\,(\wedge \bigcirc E)^* \mid (E \vee E)$, where a is some atomic formula. Moreover, we are able to show that each *BTL*-translation of *SPL* may be translated into an equivalent *BTL* expression in *restricted disjunctive normal form* (*rDNF*). While an ordinary disjunctive normal form expression is one which consists of a disjunction of conjunctions of variables and negations of variables, a *rDNF* expression consists of a disjunction of conjunctions of atomic formulae and terms defined by $\bigcirc$ operators over an atomic formula.

Definition 6.1. An *BTL* expression is in **restricted disjunctive normal form** (*rDNF*) if it has the form,

$$(a_1^1 \wedge \bigcirc a_2^1 \wedge \cdots \wedge \bigcirc^{k-1} a_k^1) \vee \cdots \vee (a_1^l \wedge \bigcirc a_2^l \wedge \cdots \wedge \bigcirc^{j-1} a_j^l)$$

where each a_i^j is a atomic formula and $\bigcirc^i a$ is defined as i $\bigcirc$ operators over some formula a.

We have shown that any *BTL* expression generated by the grammar E may be translated into *rDNF* and that the translation is valid under $\mathcal{RT}$ [30]. This normal form is proved to be useful when formalising the *bounded existence* pattern.

6.2. Bounded existence

We define the function *boundexists* to take pattern of behaviour μ, a bound b and a scope s and returns the corresponding expression in *BTL* stating μ must occur for the number of times specified by b within s and other behaviours may also occur within s. While other patterns only require the *maximum* number of states of the patterns of behaviour when specifying properties [30], it is necessary to calculate *all possible number of states* of the pattern of behaviour for specifying properties in the bounded existence pattern. This is because to express a context over a bounded number of occurrences of some pattern of behaviour μ, we need to know *exactly* the number of states all occurrences of μ span. For example the maximum number of states for the pattern of behaviour $a \wedge (\bigcirc b \vee \bigcirc(c \wedge \bigcirc d))$ is three, while it also specifies a behaviour that only spans two states, namely $a \wedge \bigcirc b$, therefore the number of states covered by two occurrences of this pattern of behaviour may either be four, five or six; we use the term *state* in the sense of a transition system of a CSP process describing a BPMN diagram: a graph showing the states it can go through and actions, each denoted by a single CSP event, that it takes to get from one to another. Algebraically this is where each transition between states is an application of a step law. To record the all possible numbers of states we provide the function *combine* that takes some *BTL* expression μ in *rDNF* recording a

pattern of behaviour and some integer n indicating the possible number of occurrences our property interested, and return a set of patterns of behaviour, each defining a disjunction of possible n occurrences of possible behaviour by μ such that each disjunct covers an equal number of states. For example, the two occurrences of the behaviour $a \wedge (\bigcirc b \vee \bigcirc(c \wedge \bigcirc d))$ would give the following set

$$\{\, a \wedge \bigcirc(b \wedge \bigcirc(a \wedge \bigcirc b)),\ a \wedge \bigcirc(c \wedge \bigcirc(d \wedge \bigcirc(a \wedge \bigcirc(c \wedge \bigcirc d)))),$$
$$(a \wedge \bigcirc(b \wedge \bigcirc(a \wedge \bigcirc(c \wedge \bigcirc d)))) \vee (a \wedge \bigcirc(c \wedge \bigcirc(d \wedge \bigcirc(a \wedge \bigcirc b)))) \,\}$$

Consequently we define function *boundexists*, which takes a pattern of behaviour μ, a bound b and a scope s and returns the corresponding *BTL* expression stating μ must occur for the number of times specified by b within s and other behaviours may also occur within s.

$$boundexists : (SPL \times Bound \times SL) \nrightarrow BTL$$

$$\forall ps : \mathbb{F}_1\, BTL;\ \mu : SPL;\ b : Bound;\ n \in \mathbb{N}_1;\ s : SL \mid$$
$$ps = combine(patternDNF(\mu), getbound(b)) \bullet$$
$$boundexists(\mu, b, s) = \bigvee \{\, p : ps \bullet boundexist(p, b, s) \,\}$$

Here the function *patternDNF normalises* the *SPL* term and the function *boundexist* considers individual partitions of possible alternative behaviour such that each partition contains a set of behaviour, each of which covers the same number of states. We write $getbound(b)$ for some bound b to denote the number part of the value. Full definitions of functions and *BTL* mappings of the property patterns may be found in our technical report [30].

For example we could use the pattern "The bounded existence of μ after ν" to describe the property that either task *A* or *C* followed by *D* has to occur followed by either one of them again throughout the whole execution of a business process. This may be expressed in *PL* as $\texttt{BEx}(a \rightarrow End \sqcap c \rightarrow End, = 2, \texttt{always})$ and the CSP specification $Spec = Spec0 \sqcap Spec1$ defines the translation corresponding of the *PL* expression, and it is defined in terms of the following processes,

$$Spec0 = start.c \rightarrow Spec2 \qquad Spec1 = start.a \rightarrow Spec3 \sqcap Spec4$$
$$Spec2 = start.d \rightarrow Spec3 \sqcap Spec4 \qquad Spec3 = start.c \rightarrow Spec5$$
$$Spec4 = start.a \rightarrow Spec6 \sqcap Spec7 \qquad Spec5 = start.d \rightarrow Spec6 \sqcap Spec7$$
$$Spec6 = Pr(\{a\}, Spec8 \sqcap Spec9) \qquad Spec7 = Pr(\{a, c\}, Spec6 \sqcap Spec7)$$
$$Spec8 = Pr(\{a, d\}, Spec8 \sqcap Spec9) \qquad Spec9 = Pr(\{a, c, d\}, Spec6 \sqcap Spec7)$$

where $Pr(X, P) = Stop \sqcap Skip \sqcap (\sqcap x : \Sigma \setminus \{t : X \bullet starts.t\} \bullet x \rightarrow P)$.

6.3. Applications

Back to our example in Fig. 1, we could now apply the absence pattern "the absence of μ between some behaviours ν and υ" [30] to specify the Requirement 4. Let us assume travel agent in Fig. 1 is deadlock free (i.e. *Agent* refines *DF*), and *start.BookSeat*, *start.RequestCancel*, *start.ReserveTimeOut* and *start.SendInvoice* denote the tasks *Book Seat*, *Request Cancel*, *Request TimeOut* and *Send Invoice* of the travel agent in Fig. 1 respectively; the following is the corresponding *PL* expression specifying this property.

$$\texttt{Abs}(Cancel, \texttt{between}\ start.BookSeat \rightarrow End\ \texttt{and}(start.Sendinvoice \rightarrow End, 2))$$

where the behaviour *Cancel* is defined as follows:

$$Cancel = start.RequestCancel \rightarrow End \sqcap start.ReserveTimeOut \rightarrow End$$

The corresponding CSP process is $Spec = Spec0 \sqcap Spec1$, which is defined in terms of the following processes.

$$Spec0 = Pr(\{\, BookSeat \,\}, Spec0 \sqcap Spec1)$$
$$Spec1 = start.BookSeat \rightarrow (Spec2 \sqcap Spec3 \sqcap Spec4 \sqcap Spec5 \sqcap Spec6)$$
$$Spec2 = Pr(\{\, BookSeat, SendInvoice \,\}, Spec7 \sqcap Spec1)$$
$$Spec3 = start.SendInvoice \rightarrow (Spec0 \sqcap Spec1)$$
$$Spec4 = start.BookSeat \rightarrow (Spec2 \sqcap Spec4 \sqcap Spec8 \sqcap Spec9)$$
$$Spec5 = Pr(\{\, BookSeat, RequestCancel, ReserveTimeOut \,\}, Spec3)$$
$$Spec6 = start.BookSeat \rightarrow (Spec3)$$
$$Spec7 = Pr(\{\, BookSeat, SendInvoice \,\}, Spec0 \sqcap Spec1)$$
$$Spec8 = Pr(\{BookSeat, RequestCancel, ReserveTimeOut, SendInvoice\}, Spec3)$$
$$Spec9 = start.BookSeat \rightarrow (Spec3)$$

Now it is possible to see if the travel agent diagram satisfies this property by checking the following refusal traces refinement assertion using the FDR tool.

$$Es = \{\, BookSeat, RequestCancel, ReserveTimeOut, SendInvoice \,\}$$
$$Spec \sqsubseteq_{\mathcal{RT}} Agent \setminus (\Sigma \setminus \{t : Es \bullet start.t\})$$

7. Modelling empirical studies

7.1. Empirical studies

This section considers the application of BPMN to analysing long running empirical studies. In particular we develop a novel declarative model, *OWorkflow*, for specifying empirical studies, and implement transformation functions between *OWorkflow* and BPMN in Haskell. Note that the mapping from BPMN to *OWorkflow* is partial, as only a subset of BPMN is required.

```
wTob :: OWorkflow -> BPMN          bTow :: BPMN -> OWorkflow
```

OWorkflow extends the original CancerGrid trial model [2], which has been developed specifically for recording clinical trials.

Formally an empirical study, represented by *OWorkflow*, is a list of *sequence rules*: a sequence rule is a 8 tuple N, P, D, G, E, A, R, W, where N is a unique identifier, typed `ActivityId`; P is a structural composition of prerequisites, identifying preceding sequence rules; D is a structural composition of dependencies, identifying succeeding sequence rules; G and E, both typed `Condition`, are the starting and terminating conditions; A is a observation group, identifying manual and automated activities to be performed when the rule is evaluated; R is a list of repeat clauses, and W is a work group, identifying the collection of work blocks to be carried out when the rule is evaluated, each block representing a set of empirical procedures such as those of administering drug treatments to a patient in a clinical trial. For reasons of space we only describe the structure of a work group via our example in Fig. 3; full definition and description of sequence rules may be found in our technical reports [28]. Each work group is defined by the data type `Wks`.

```
data Wks = ChoiceW [Wks] | ParW [Wks] | SeqW [Wks] | Wk WBlock
```

where the constructor `ChoiceW`, `ParW` and `SeqW` denotes the choice, interleaving and sequential composition of collection works `[Wks]`. Each `WBlock` records a set of empirical procedures; each procedure is identified by its name, methods, duration and iteration. For example the subprocess state *EC* in Fig. 3 is work block that contains a two work units *EC_C* and *EC_E* which are represented as BPMN task states.

7.2. Verification

We assume the process *A*2 to be the relative timed behaviour of the diagram in Fig. 3. Here we use the CSP events *starts*.*N* where *N* is a value over the data type *Node* to denote administration of the respective drug.

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

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

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

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

and here is the corresponding failures refinement assertion.

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

We have abstracted the behaviour of the diagram by hiding part of *A*2's alphabet because the property we are interested in only covers the set of events,

$$\{ start.TG\_G, start.EC\_E, start.EC\_C, fin.0 \}$$

i.e. the alphabet of the process S. When we ask FDR to check this assertion the counterexample trace $\langle start.EC\_C, start.TG\_G \rangle$ is given. This tells us that a dosage of gemcitabine can be given after a dosage cyclophosphamide; this trace is sufficient to disprove the correctness of our example against the sequence rule, since a dosage of epirubicin must be after gemcitabine according to the syntactic structure of the diagram.

A more detailed analysis reveals that while cyclophosphamide may be administered after 14 days and epirubicin may only be administered after 18 days, paclitaxel may be delayed for as long as 10 days before being administered, and since gemcitabine is allowed to be administered within the 10 days, it may be given after 5 days, that is before epirubicin and after cyclophosphamide. A possible solution to this is either to restrict the duration in which cyclophosphamide and epirubicin may be administered, or to delay the administration of gemcitabine.

8. Related work and summary

8.1. Semantics

To the best of our knowledge, the only previous attempt at defining a formal semantics for a subset of BPMN did so using Petri nets [4]. However, that semantics does not properly model multiple instances and does not allow comparisons of diagrams via refinements. A significant amount of work has been done towards the mapping between a particular class of BPMN diagrams and WS-BPEL (e.g. [20]), and the formal semantics of WS-BPEL (e.g. [10]). However, as the use of graphical notations to assist the development process of complex software systems has become increasingly important, it is necessary to define a formal semantics for BPMN to ensure precise specification and to assist developers in moving towards correct implementation of business processes. A formal semantics also encourages automated tool support for the notation.

Similarly we introduce the first relative timed model for a collaborative graphical notation like BPMN. Some attempts have been made to provide timed models for similar notations such as UML activity diagrams (e.g. [7]) and Workflow nets [13]. Guelfi et al. [7] have defined their discrete timed semantic models in the Clocked Transition System notation, where behavioural specifications are expressed as temporal logic formulae and verification is carried out via model checking; Ling et al. defined a formal semantics for a timed extension of van der Aalst's Workflow nets [24] in terms of timed Petri nets; nevertheless, their semantics do not provide the level of abstraction required to model time explicitly, in that they model discrete units of time, which we believe may not be directly applicable to the business process developers whereas our definition captures the six-dimensional space defined by W3C standards [32, Section 3.2.6]. Also unlike BPMN, their target graphical notations and hence their semantic models are not designed for analyses of collaborations where more than one diagram is under consideration. Furthermore, our semantic model has been defined in correspondence to our earlier untimed model [26] so that time-independent behavioural properties may be preserved across both models.

8.2. Compatibility

This paper also addresses the notion of compatibility of BPMN diagrams; in particular, our work documents the relationship between compatibility in the untimed and timed settings. While we are unaware of prominent work in dealing with this issue when focusing in the untimed setting, there exist 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 [3] 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.

8.3. Empirical studies

While the application of graphical workflow technology to empirical studies and calendar scheduling is new, large amounts of research have focused on the application of workflow notations and implementations to "in silico" scientific experiments. Notable is Ludäscher et al.'s Kepler System [15] and Microsoft Research's Trident Workflow Workbench [23], in which such experiments are specified as a workflow graphically and fully automated by interpreting the workflow descriptions on a runtime engine. On the other hand we employ BPMN as a graphical notation to specify and graphically visualise experiments and studies that are typically long running and in which automated tasks are often interleaved with manual ones: studies such as clinical trial would also include "in vivo" intervention. Furthermore, our approach targets studies that are usually recorded in a calendar schedule to assist administrators and managers. Similarly, research effort has been directed towards effective planning of *specific types* of long running empirical studies, namely clinical trials and guidelines. Notable is Modgil and Hammond's Design-a-Trial (DaT) [16]. DaT is a decision support tool for critiquing the data supplied specifically for randomized controlled clinical trial specification based on expert knowledge, and subsequently outputting a protocol describing the trial. DaT includes a graphical trial planner, which allows description of complex procedural contents of the trial. To ease to complexity of protocol constructions, DaT uses macros, common plan (control flow) constructs, to assist trial designers to construct trial specifications. More recently

8.4. Summary

This paper presents a formalisation of BPMN and describes some of its applications through examples. In particular, we introduce a process semantics in CSP and describe how this may be applied to reasoning as well as the refinement of BPMN diagrams. A timed model is then introduced, which augments our untimed model with relative timing, and using these two models we discuss the notion of compatibility and its relationship between the models. We also present a pattern-based approach to constructing property specifications for BPMN, which complements our formal semantics. In applications of

BPMN we use a well-documented example of ticket reservation systems to illustrate the applicability of our approach, as well as investigating the use of BPMN in modelling empirical studies.

References

[1] Mario Bravetti, Gianluigi Zavattaro, Towards a unifying theory for choreography conformance and contract compliance, in: Proc. of 6th International Symposium on Software Composition, SC'07 2007.
[2] James Brenton, Carlos Caldas, Jim Davies, Steve Harris, Peter Maccallum, CancerGrid: Developing open standards for clinical cancer informatics, in: Proceedings of the UK e-science All Hands Meeting 2005, 2005, pp. 678–681.
[3] Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, Steve Ross-Talbot, A Theoretical Basis of Communication-Centred Concurrent Programming, Technical Report, W3C, 2006.
[4] R. M. Dijkman, M. Dumas, C. Ouyang, Formal semantics and automated analysis of BPMN process models, Technical Report Preprint 5969, Queensland University of Technology 2007.
[5] Matthew B. Dwyer, George S. Avrunin, James C. Corbett, Patterns in property specifications for finite-state verification, in: Proceedings of the 21st International Conference on Software Engineering, 1999.
[6] Formal Systems (Europe) Ltd. Failures-Divergences Refinement, FDR2 User Manual 1998, www.fsel.com.
[7] Nicolas Guelfi, Amel Mammar, A formal semantics of timed activity diagrams and its PROMELA translation, in: APSEC05, 2005, pp. 283–290.
[8] Peter Hammand, Marek J. Sergot, Jeremy C. Wyatt, Formalisation of safety reasoning in protocols and hazard regulations, in: 19th Annual Symposium on Computer Applications in Medical Care, 1995.
[9] Haskell, http://www.haskell.org.
[10] H. Foster, Mapping BPEL4WS to FSP, Technical Report, Imperial College, 2003.
[11] ILOG JViews BPMN Modeler. Available at http://www.ilog.com/products/jviews/diagrammer/bpmnmodeler/.
[12] I. Linden, J.-M. Jacquet, K. De Bosschere, A. Brogi, On the expressiveness of timed coordination models, Science of Computer Programming 61 (2) (2006) 152–187.
[13] Sea Ling, H. Schmidt, Time petri nets for workflow modelling and analysis, in: Proceedings of 2000 IEEE International Conference on Systems, Man, and Cybernetics, 2000, pp. 3039–3044.
[14] Gavin Lowe, Specification of communicating processes: temporal logic versus refusals-based refinement, Formal Aspects of Computing 20 (3) (2008).
[15] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the Kepler system: Research articles, Concurrency and Computation: Practice & Experience 18 (10) (2006) 1039–1065.
[16] S. Modgil, P. Hammond, Decision support tools for clinical trial design, Artificial Intelligence in Medicine 27 (2003).
[17] Abida Mukarram, A Refusal Testing Model for CSP, D.Phil Thesis, University of Oxford, 1992.
[18] Model Checking BPMN. Available at http://www.comlab.ox.ac.uk/peter.wong/bpmn/.
[19] OMG. Business Process Modeling Notation (BPMN) Specification. February 2006. www.bpmn.org.
[20] C Ouyang, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, Translating BPMN to BPEL, Technical Report BPM-06-02, BPM Center, 2006.
[21] J. N. Reed, J. E. Sinclair, A. W. Roscoe, Responsiveness of interoperating components, Formal Aspects of Computing 16 (4) (2004) 394–411.
[22] A. W. Roscoe, The Theory and Practice of Concurrency, Prentice-Hall, 1998.
[23] Microsoft Research Trident Scientific Workflow Workbench, http://research.microsoft.com/en-us/collaboration/tools/trident.aspx.
[24] W.M.P. van der Aalst, Verification of workflow nets, in: ICATPN'97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets, 1997 pp. 407–426.
[25] W3C, Web Service Choreography Interface (WSCI) 1.0, November 2002, www.w3.org/TR/wsci.
[26] Peter Y. H. Wong, Jeremy Gibbons, A process semantics for BPMN, in: Proceedings of 10th International Conference on Formal Engineering Methods, in: LNCS, vol. 5256, 2008, Technical Report version available at http://www.comlab.ox.ac.uk/peter.wong/pub/bpmnsem.pdf.
[27] Peter Y. H. Wong, Jeremy Gibbons, A relative-timed semantics for BPMN, in: Proceedings of 7th International Workshop on the Foundations of Coordination Languages and Software Architectures, in: ENTCS, vol. 229, 2009, Technical Report version available at http://www.comlab.ox.ac.uk/peter.wong/pub/bpmntime.pdf.
[28] Peter Y.H. Wong, Jeremy Gibbons, On specifying and visualising long-running empirical studies, in: Proceedings of International Conference on Model Transformation, in: LNCS, vol. 5063, 2008, Technical Report version available at http://www.comlab.ox.ac.uk/peter.wong/pub/transext.pdf.
[29] Peter Y. H. Wong, Jeremy Gibbons, Verifying business process compatibility, in: Proceedings of 8th International Conference on Quality Software, IEEE Computer Society, 2008, pp. 126–131.
[30] Peter Y. H. Wong, Jeremy Gibbons, Property specifications for workflow modelling, in: Proceedings of 7th International Conference on Integrated Formal Methods in: LNCS vol. 5423, February 2009.
[31] J.C.P. Woodcock, J. Davies, Using Z: Specification, Proof and Refinement, Prentice Hall International Series in Computer Science, 1996.
[32] XML Schema Part 2: Datatypes Second Edition, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.