A Process Semantics for BPMN

Peter Y. H. Wong and Jeremy Gibbons

Computing Laboratory, University of Oxford, United Kingdom
{peter.wong,jeremy.gibbons}@comlab.ox.ac.uk

Abstract. Business Process Modelling Notation (BPMN), developed by the Business Process Management Initiative (BPMI), intends to bridge the gap between business process design and implementation. However, the specification of the notation does not include a formal semantics. This paper shows how a subset of the BPMN can be given a process semantics in Communicating Sequential Processes. Such a semantics allows developers to formally analyse and compare BPMN diagrams. A simple example of a business process is included to demonstrate the application of the semantics; some theoretical results about the semantics are briefly discussed.

1 Introduction

Modelling of business processes and workflows is an important area in software engineering. Business Process Modelling Notation (BPMN) [13] 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 developed by BPMI and adopted by OMG does not have a formal behavioural semantics, which we believe is crucial in behavioural specification and verification activities.

BPMN has been specified to map directly to the BPML standard, which has subsequently been superceded by WS-BPEL [2]. 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, 5]. However, their semantics does not properly model multiple instances, exception handling and message flows. A significant amount of work has been done towards the mapping between a particular class of BPMN diagrams to WS-BPEL and [14, 15], and the formal semantics of WS-BPEL [8, 10–12]. 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.

The main contribution of our work is to provide a formal process semantics for a subset of BPMN, in terms of the process algebra CSP [16]. By using the language and the behavioural semantics of CSP as the denotational model, we show how the existing refinement orderings defined upon CSP processes can be

applied to the refinement of business process diagrams, and hence demonstrate how to specify behavourial properties using BPMN. Moreover, our processes may be readily analysed using a model checker such as FDR [7]. Our semantic construction starts from syntax expressed in Z [19], following Bolton and Davies's work on UML activity graphs [1].

This paper begins with an introduction to BPMN and the mathematical notations, Z [19] and CSP [16], that are used throughout the paper. Our contribution starts in Section 3, with a Z model of BPMN syntax, and continues in Section 4 with a behavioural semantics in CSP. In Section 5 we give a simple example to show how our semantics allows *consistency* between different levels of abstraction to be verified, and discuss briefly some theoretical results. We conclude this paper with a summary.

2 Notation

2.1 BPMN

States in our subset of BPMN [13] can either be pools, tasks, subprocesses, multiple instances or control gateways; they are linked by sequence, exception or message flows; sequence flows can be either incoming to or outgoing from a state and have associated guards; an exception flow from a state represents an occurrence of error within the state. Message flows represent directional communication between states. A sequence of sequence flows hence represents a specific control flow instance of the business process.

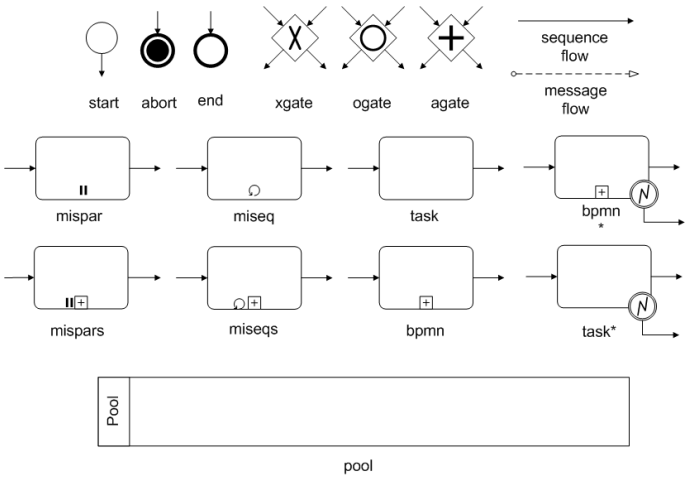

Fig. 1. States of BPMN diagram

A table showing each type of state is presented in Figure 1. In the figure, 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. 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 models an unsuccessful termination, usually an error of an instance of the business process in the current scope.

Also in the figure, each of the *xgate*, *agate* and *ogate* state types 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. An *ogate* state is an inclusive gateway, accepting one or more incoming sequence flows depending on their associated guards and initialising one or more of its outgoing flows also depending on their associated guards.

A *task* state describes an atomic activity and has exactly one incoming and one outgoing transition. A *bpmn* state describes a subprocess state; it is a business process by itself and so it models a flow of BPMN states. Figure 1 depicts a collapsed subprocess state where all internal details are hidden; this state has exactly one incoming and one outgoing transition. Also in Figure 1 there are graphical notations labelled *task** and *bpmn**, which depict a task state and a subprocess state with an exception flow. Each task and subprocess can 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 1 depicts 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. For reasons of space we have omitted the syntactic and semantic definitions of message flows, details of which are in an extended version of this paper [18].

2.2 Z

The Z notation [19] 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:

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

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.

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

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

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

2.3 CSP

In CSP [16], 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.

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

Process $P \mid\mid\mid Q$ denotes the interleaved parallel composition of processes P and Q. Process $P \,[\![A]\!]\, Q$ denotes the partial interleaving of processes P and Q sharing events in set A. Process $P \,[\![\, A \mid B \,]\!]\, Q$ 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 $|||\, i : I \bullet P(i)$, $||[A]\, i : I \bullet P(i)$ 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 [16]. For example, the processes $a \rightarrow \mathit{Skip}$ and $(a \rightarrow \mathit{Skip}) \sqcap \mathit{Stop}$ have the same traces (the traces model is prefix-closed), even though the latter one is allowed to do nothing at all no matter what we offer it. 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.

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

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

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

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 [7], 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 Syntactic Description of BPMN

In this section we describe the abstract syntax of BPMN using Z schemas and set theory, and use an example in Section 3.2 to show how the syntax can be applied on a given BPMN diagram. For reasons of space, we have omitted certain schema and function definitions and have only concentrated on the definition of a smaller subset of the BPMN states than shown in Section 2; readers may refer to our longer paper [18] for their full definitions.

3.1 Abstract Syntax

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

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

We then derive subtypes $BName$ and $PLName$ axiomatically:

$$BName, PLName : \mathbb{P}\, PName$$

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

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

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

Each type of state shown in Figure 1 is defined using the free type *Type* where each of its constructors describes a particular type of states. For example, the type of an atomic *task* state is defined by $task\, t$ where t is a unique name that identifies that task state. Below is the partial definition.

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

According to the specification [13], each BPMN 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*. We define some abbreviations as follows to assist our specification.

$$Tasks == \text{ran}\ task \cup \text{ran}\ miseq \cup \text{ran}\ mipar$$
$$Subs == \text{ran}\ bpmn \cup \text{ran}\ mipars \cup \text{ran}\ miseqs$$

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.

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

Here we show a partial definition of the schema *State* for each BPMN *state*, omitting the inclusion of schema components for message flows.

$$State \mathrel{\widehat=} [type : Type;\ in, out, error : \mathbb{P}\, Trans;\ exit : \mathbb{P}(\mathbb{N} \times Trans);\ loop : \mathbb{N}]$$

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. Each state incorporates the variable *loop* to limit the number of state instances the process instance can invoke. The state also records different types of message flows, but we have omitted their definition in this paper.

We denote a subset of *well-formed* states in BPMN by the schema type *WFS*, and we define the type $WCF : \mathbb{P}(\mathbb{P}\, State)$ to be the set of *well-configured* sets of well-formed states *WCF*. Well-formedness is defined to conform to the constraints within the official documentation [13]; for example, a *start state* must have no incoming transition and only one outgoing transition. A definition of this subset may be found in the extended version of this paper [18].

Each BPMN diagram, encapsulated by a *pool* representing an individual participant in a collaboration, is 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. The function type *Local* represents the environment of the local specification and each function of its type maps each name of a BPMN diagram to its associated diagram. Consequently a collaboration is built up from a finite set of names, and each of the names is associated with a BPMN diagram. For reasons of space both the syntactic and semantic definition of collaboration have been omitted, again, see the extended version of this paper [18].

$$BPD ::= states\langle\!\langle WCF \rangle\!\rangle$$
$$Local == PName \nrightarrow BPD$$

3.2 An Example

We present an example of a business process of an airline reservation system shown in Figure 2; this example has been taken from the WSCI specification [17]. It could be assumed to have been constructed during the development of the reservation system. We have abstracted message flows, as there is only one business participant in the example. We use this example to illustrate how a BPMN

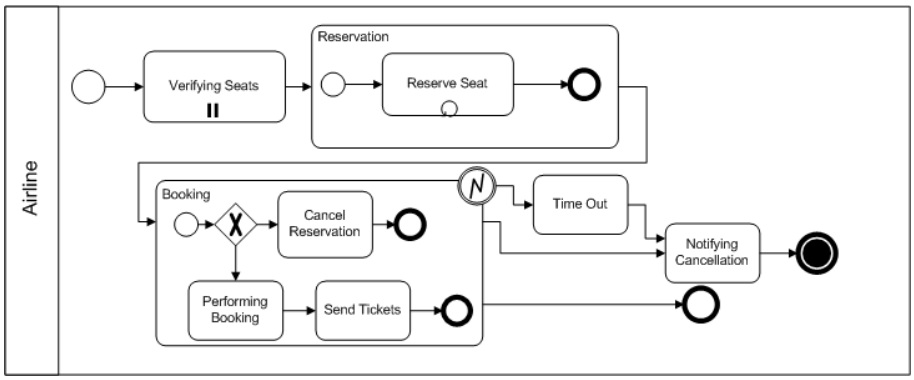

Fig. 2. A BPMN diagram describing the workflow of an airline reservation application.

diagram can be translated into a well-configured set of states describing the diagram's syntax.

We observe that the airline reservation business process is initiated by verifying seat availability, after which seats may be reserved. If the reservation period elapses, the business process will cancel the reservation automatically and notify the user. The user might decide to cancel her reservation, or proceed with the booking. Upon a successful booking, tickets will be issued.

Given the business process name *airline*, the following shows a set of well-formed states translated from the diagram describing the reservation part of business process. We have omitted details of the bindings of *Trans* and *Messageflow*. We write $a_1 \mathinner{..} a_n \rightsquigarrow \emptyset$ inside some schema binding s to specify the components $s.a_1 \mathinner{..} s.a_n$ to be empty. The syntactic details of the subprocesses *Reserve* and *Booking* are also omitted.

$$
\begin{array}{|l}
airline : PName;\ book, reserve : BName;\ verify, timeout, notify : Task \\
\hline
\exists\, local : Local;\ t1, t2, t3, t4, t5, t6, t7, t8 : Trans;\ i, j, k, l, m, n : \mathbb{N} \bullet \\
\quad states^{\sim}(local\ airline) = \\
\qquad \{ \langle type \rightsquigarrow start, out \rightsquigarrow \{ t1 \}, in, error, exit \rightsquigarrow \emptyset \rangle, \\
\qquad\ \langle type \rightsquigarrow mipar\ verify\ n, in \rightsquigarrow \{ t1 \}, out \rightsquigarrow \{ t2 \}, error, exit \rightsquigarrow \emptyset \rangle, \\
\qquad\ \langle type \rightsquigarrow bpmn\ reserve, in \rightsquigarrow \{ t2 \}, out \rightsquigarrow \{ t3 \}, error \rightsquigarrow \emptyset, exit \rightsquigarrow \{ (m, t3) \} \rangle, \\
\qquad\ \langle type \rightsquigarrow bpmn\ book, in \rightsquigarrow \{ t3 \}, out \rightsquigarrow \{ t4, t5 \}, error \rightsquigarrow \{ t6 \}, \\
\qquad\qquad exit \rightsquigarrow \{ (k, t4), (l, t5) \} \rangle, \\
\qquad\ \langle type \rightsquigarrow task\ timeout, in \rightsquigarrow \{ t6 \}, out \rightsquigarrow \{ t7 \}, error, exit \rightsquigarrow \emptyset \rangle, \\
\qquad\ \langle type \rightsquigarrow task\ notify, in \rightsquigarrow \{ t5, t7 \}, out \rightsquigarrow \{ t8 \}, error, exit \rightsquigarrow \emptyset \rangle, \\
\qquad\ \langle type \rightsquigarrow end\ i, in \rightsquigarrow \{ t4 \}, out, error, exit \rightsquigarrow \emptyset \rangle, \\
\qquad\ \langle type \rightsquigarrow abort\ j, in \rightsquigarrow \{ t8 \}, out, error, exit \rightsquigarrow \emptyset \rangle \ \}
\end{array}
$$

4 Behavioural Semantics of BPMN

In Section 3 we gave an overview of the abstracted syntax for BPMN in Z. In this section, we define a semantic function which takes the syntactic description of a BPMN diagram and returns the CSP process that models the behaviour of that diagram. That is, the function returns the parallel composition of processes corresponding to the states of the diagram, each synchronising on its own alphabet, which represents its transition events, to ensure the correct order of control flow. For reasons of space, we only consider the semantics of a BPMN diagram with a single participant (i.e. one pool), and each function associated to the semantics will be defined over a smaller subset of the BPMN states, namely the states of type *start*, *end*, *task*, *miseq*, *miseqs* (subprocess), *bpmn* (subprocess), *agate*, *xgate* and *ogate*, which have been described in Section 2; the semantics of other states in the figure may be defined similarly. The complete semantic definition of business collaboration and of other states may be found in our longer paper [18]. The rest of the section is structured as follows: in Section 4.1 we define functions to associate each transition, state and diagram with their set of events; Section 4.2 presents the overall semantic functions for mapping each BPMN diagram to its process describing its behaviour; in Section 4.3 we present the CSP processes corresponding to the behaviour of each gateway; and in Section 4.4 we define processes corresponding to the behaviour of each state type and transition, and the general functions for mapping each BPMN state to the CSP process describing its behaviour.

4.1 Alphabets

First we define the basic types *Process* and *Event* which correspond to CSP processes and events.

$$[Process, Event]$$

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.

$$\begin{array}{l} \epsilon_{line} : Line \rightarrowtail Event \\ \epsilon_{task} : Task \rightarrowtail Event \\ \epsilon_{pname} : PName \rightarrowtail Event \\ \epsilon_{trans} : Trans \rightarrowtail (Event \times Guard) \\ \hline \epsilon_{trans} = \lambda\, Trans \bullet (\epsilon_{line}\, line, guard) \end{array}$$

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 function α_{trans} which maps each set of transitions to the set of associated events. (Given a tuple of n

elements t, we use the projection notation $t.m$ to the denote the mth element of the tuple.)

$$\alpha_{trans} : \mathbb{P}\, Trans \rightarrowtail \mathbb{P}\, Event$$

$$\alpha_{trans} = \lambda\, ts : Trans \bullet \{\, t : \epsilon_{trans}(\!|\ ts\ |\!) \bullet t.1 \,\}$$

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 its incoming and outgoing transitions, type and exception flows. We define α_{state} to be a function mapping each state into its alphabet.

$$\begin{array}{l}
\alpha_{state} : Local \nrightarrow State \nrightarrow \mathbb{P}\, Event \\
\hline
\alpha_{state} = (\lambda\, l : Local \bullet (\lambda\, State \bullet \\
\textbf{if } (type \notin (Tasks \cup Subs)) \textbf{ then } \alpha_{trans}(out \cup in) \\
\textbf{else } (\textbf{if } (type \in \operatorname{ran} miseq) \textbf{ then } \alpha_{trans}\, (\mu\, t : miseqt\, s \bullet \{\, t.1, t.2 \,\}) \textbf{ else } \emptyset)) \\
\qquad \cup (\textbf{if } (type \in Subs) \textbf{ then } \bigcup((\alpha_{state}\, l)\, (\!|\ states^{\sim}(l(bpmn^{\sim} type))\ |\!)) \\
\qquad\quad \textbf{else } (\textbf{if } (type \notin Tasks) \textbf{ then } \emptyset \textbf{ else } \{\, \epsilon_{task}(task^{\sim} type) \,\})) \\
\qquad \cup\, \alpha_{trans}(out \cup in \cup error))))
\end{array}$$

The function *miseqt* maps each state of type *miseq* to a transition pair used to connect the state's task or subprocess state.

$$miseqt : State \rightarrowtail (Trans \times Trans)$$

$$miseqt = (\lambda\, State \bullet (\mu (s, t) : (Trans \times Trans) \mid s \neq t))$$

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 *local* diagram's behaviour.

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

4.2 Processes corresponding to Diagrams

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 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. 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.

$$
\begin{array}{l}
bsem : PName \pfun Local \pfun Process \\
hide : PName \pfun Local \pfun \mathbb{P}\, Event \\
\hline
\forall\, p : PName;\ l : Local \bullet \\
bsem\, p\, l = \\
\quad \textbf{let}\ A = \{ a : \epsilon_{abort}\, p\, l;\ e : \epsilon_{end}\, p\, l \bullet fin.e, abt.a \} \cup \alpha_{proc}\, p\, l \\
\quad\quad X = \Box\, i : \alpha_{proc}\, p\, l \bullet i \rightarrow X \ \Box\ (\Box\, e : \epsilon_{abort}\, p\, l \bullet abt.e \rightarrow Stop) \\
\quad\quad\quad \Box\ (\Box\, e : \epsilon_{end}\, p\, l \bullet fin.e \rightarrow Skip) \\
\quad \textbf{in}\ \ (bsm\, p\, l \,[\![\, A \,]\!]\, X) \setminus hide\, p\, l \\
\land\ hide\, p\, l = \bigcup \{\, s : states^{\sim}(l\, p) \bullet \alpha_{trans}(s.in \cup s.out \cup s.error) \,\}
\end{array}
$$

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

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* states within the diagram's syntax.

$$
\begin{array}{l}
\alpha_{start} : PName \pfun Local \pfun \mathbb{P}\, Event \\
\epsilon_{end} : PName \pfun Local \pfun \mathbb{P}\, \mathbb{N} \\
\hline
\forall\, p : PName;\ local : Local \bullet \\
\quad \alpha_{start}\, p\, local = \bigcup \{\, s : states^{\sim}(local\, p) \mid s.type = start \bullet \alpha_{trans}(s.out) \,\} \\
\quad \land\ \epsilon_{end}\, p\, local = \{\, s : states^{\sim}(local\, p) \mid s.type \in \operatorname{ran} end \bullet end^{\sim} s.type \,\}
\end{array}
$$

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

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

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

We define the function ρ_{miseq} to map each state of type *miseq* or *miseqs*. The following describes the function ρ_{miseq}.

$$
\begin{array}{l}
\rho_{miseq} : State \pfun Local \pfun Process \\
\hline
\forall\, s : State;\ local : Local \bullet \exists\, t1, t2 : Trans;\ e1, e2 : Event;\ n : \mathbb{N} \bullet \\
\quad (t1, t2) = miseqtst\ s \land (e1, e2) = ((\epsilon_{trans}\ t1).1, (\epsilon_{trans}\ t2).1) \\
\quad \land\ (\textbf{if}\ s.type \in \ran\ miseq\ \textbf{then}\ n = (miseq^{\sim} s.type).2\ \textbf{else}\ n = (miseqs^{\sim} s.type).2) \\
\quad \land\ \rho_{miseq}\ s\ local = \\
\qquad\qquad \textbf{let}\ \ SY = \alpha_{trans}(s.out \cup s.error) \cup \{\, e1, e2 \,\} \\
\qquad\qquad \textbf{in}\ \ ((Cq(n, s, e1, e2) \lpar\!\lbrack SY \rbrack\!\rpar Seq(n, s, local)) \rhd AJ(s.error)) \setminus \{\, e1, e2 \,\}
\end{array}
$$

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.

$$
\begin{array}{l}
Seq(i, s, l) = \\
\quad \textbf{let}\ \ tpe = \textbf{if}\ s.type \in \ran\ miseq\ \textbf{then}\ task\ (miseq^{\sim} s.type)\ \textbf{else}\ bpmn\ (miseqs^{\sim} s.type) \\
\qquad st = \langle\!|\, in \rightsquigarrow \{\, t1 \,\}, type \rightsquigarrow tpe, out \rightsquigarrow \{\, t2 \,\}, error \rightsquigarrow s.error, loop \rightsquigarrow 1 \,|\!\rangle \\
\quad \textbf{in}\ i > 0\ \&\ ((\rho_{state}\ st\ l) \fatsemi Seq(i-1, s, l)) \Box XS(s.out)
\end{array}
$$

The process *Cq* is triggered initially by one of the incoming transitions of the multiple instance state. Instances are triggered sequentially.

$$
\begin{array}{l}
Cq(n, s, e, f) = \\
\quad ((XJ(s.in) \Box f \then Skip) \fatsemi \\
\quad ((n > 1)\ \&\ (e \then Cq(n-1, s, e, f)) \Box n = 1\ \&\ (e \then f \then XS(s.out)) \Box XS(s.out)))
\end{array}
$$

4.3 Processes corresponding to Gateways

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

Exclusive Choice Gateway Processes $XS(tn)$ and $XJ(tn)$ model the behaviour of outgoing and incoming transitions of the state type *xgate*. Note that 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.

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

We also define the process $AJ(tn)$ to model the behaviour of incoming transitions of the state type *abort*.

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

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 transitions are required in this state type.

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

Inclusive Choice Gateway Process $OSJ(tn)$ models the behaviour of outgoing and incoming transitions of the state type *ogate*. Note that all outgoing transitions are guarded in the state type *ogate*, one or more transitions are enabled and the choice of transitions is based on the value of their guards. All its incoming transitions are also guarded; the choice of transitions is based on the value of their guards.

$$OSJ(tn) = |||\ e : \epsilon_{trans}(\!|\ tn\ |\!) \bullet \ (\text{if } e.2 \text{ then } e.1 \rightarrow \mathit{Skip} \text{ else } \mathit{Skip})$$

4.4 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.

$$\rho_{out} : \mathit{State} \nrightarrow \mathit{Process}$$
$$\rho_{in} : \mathit{State} \nrightarrow \mathit{Process}$$

$$\rho_{out} = (\lambda\, \mathit{State} \bullet \ \textbf{if } (\mathit{type} = \mathit{asplit}) \textbf{ then } ASJ(\mathit{out}) \textbf{ else if } (\mathit{type} = \mathit{osplit}) \textbf{ then } OSJ(\mathit{out}) \textbf{ else } XS(\mathit{out}))$$
$$\rho_{in} = (\lambda\, \mathit{State} \bullet \ \textbf{if } (\mathit{type} \in \operatorname{ran} \mathit{abort}) \textbf{ then } AJ(\mathit{in}) \textbf{ else if } (\mathit{type} = \mathit{ajoin}) \textbf{ then } ASJ(\mathit{in}) \textbf{ else if } (\mathit{type} = \mathit{ojoin}) \textbf{ then } OSJ(\mathit{in}) \textbf{ else } XJ(\mathit{in}))$$

The function ρ_{type} maps the type of a given state to its corresponding process. Since our semantics abstracts the internal flow of task states, we only model

the initialisation, the termination, message flows and any exception flow of each task.

$$
\begin{array}{l}
\rho_{exit} : State \nrightarrow Process \\
\rho_{type} : State \nrightarrow Local \nrightarrow Process \\
\hline
\rho_{exit} = (\lambda\, State \bullet \\
\quad \textbf{let}\ Y = \{\, (e, f) : exit \bullet (fin.e, (\epsilon_{trans}\, f).1) \,\} \\
\quad \textbf{in}\ (\Box (i, j) : Y \bullet i \rightarrow j \rightarrow Skip) \,\Box\, XJ(error)) \\
\rho_{type} = (\lambda\, State \bullet (\lambda\, l : Local \bullet \\
\ \textbf{if}\ (type \in \operatorname{ran} task) \\
\ \textbf{then if}\ (error = \emptyset)\ \textbf{then}\ \epsilon_{task}(task^{\sim} type)\ \textbf{else}\ \epsilon_{task}(task^{\sim} type) \triangle XJ(error) \\
\ \textbf{else if}\ (type \notin \operatorname{ran} task \cup \operatorname{ran} bpmn)\ \textbf{then}\ Skip \\
\quad \textbf{else}\ (\textbf{if}\ (error = \emptyset)\ \textbf{then}\ \epsilon_{pname}(bpmn^{\sim} type) \rightarrow bsem\, (bpmn^{\sim} type)\, l \\
\qquad \textbf{else}\ \epsilon_{pname}(bpmn^{\sim} type) \rightarrow (bsem\, (bpmn^{\sim} type)\, l \triangle XJ(error)))))
\end{array}
$$

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 and outgoing transitions.

$$
\begin{array}{l}
\rho_{state} : State \nrightarrow Local \nrightarrow Process \\
\hline
\rho_{state} = (\lambda\, s : State \bullet (\lambda\, l : Local \bullet \\
\textbf{if}\ (type \in \operatorname{ran} task)\ \textbf{then}\ (\rho_{in}\, s \fatsemi \rho_{type}\, s\, l \fatsemi \rho_{out}\, s) \\
\textbf{else if}\ (type \in \operatorname{ran} bpmn) \\
\quad \textbf{then}\ (\rho_{in}\, s \fatsemi ((\rho_{type}\, s\, l \,[\![\, \{\, e : exit \bullet fin.(e.1) \,\} \cup \alpha_{trans}\, error]\!] \\
\qquad \rho_{exit}\, s\, l) \,[\![\, \{\, o : out \bullet (\epsilon_{trans}\, e).1 \,\} \,]\!]\, \rho_{out}\, s)))) \\
\quad \textbf{else if}\ (type \in \operatorname{ran} miseq \cup \operatorname{ran} miseqs)\ \textbf{then}\ \rho_{miseq}\, s\, l \\
\qquad \textbf{else if}\ (type = start)\ \textbf{then}\ \rho_{out}\, s \\
\qquad\quad \textbf{else if}\ (type \in \operatorname{ran} end \cup \operatorname{ran} abort)\ \textbf{then}\ \rho_{in}\, s \\
\qquad\qquad \textbf{else}\ \rho_{in}\, s \fatsemi \rho_{out}\, s))
\end{array}
$$

We have implemented the semantics described in this paper as a prototype tool using the functional programming language Haskell. Readers may find a copy of the implementation from our web site[1]. The tool inputs a XML serialised representation of BPMN diagram from the JViews BPMN Modeler [9], and translates it into an ASCII file containing CSP processes representing its behaviours expressed in machine-readable CSP [16].

5 Revisiting the Example

5.1 Semantics of the Airline Reservation Application

We use the example of an airline reservation system in Section 3.2 to demonstrate how our semantic function may be applied to the syntactic definition described

[1] `http://www.comlab.ox.ac.uk/peter.wong/observation/`

in Section 3, and hence provide a semantics to support formal analyses. We define set J to index the processes corresponding to the states in the diagram.

$$J = \{\, start, verify, reserve, booking, notify, timeout, end, abort \,\}$$

By applying our semantic function to the diagram's syntactic description, we obtain the process corresponding to it.

$$\begin{aligned} Airline = \mathbf{let}\ & X = \Box\, i : (\alpha Y \setminus \{\, fin.1, abt.1 \,\}) \bullet \\ & \qquad (i \rightarrow X \ \Box\ abt.1 \rightarrow Stop \ \Box\ fin.1 \rightarrow Skip) \\ & Y = (\ \|\, j : J \bullet \alpha P(j) \circ P(j)) \\ \mathbf{in}\ & (Y \ [\![\, \alpha Y \,]\!]\ X) \setminus \{\!|\, init \,|\!\} \end{aligned}$$

where for each j in J, the process $P(j)$ is as defined below and $\alpha P(j)$ is the set of possible events performed by $P(j)$. We use n, ranging over $\mathbb{N}$, to denote the number of instances of the task *verify*, as specified by the second argument of constructor function *miseq*.

$$\begin{aligned} & P(verify) = \\ & \ \mathbf{let} \\ & \qquad Ts = \{\, i : \{\, 1 \,..\, n \,\} \bullet (in.i, out.i) \,\} \\ & \qquad IC(T) = \Box(i,j) : T \bullet i \rightarrow (j \rightarrow Skip \ |||\ Cn(T \setminus \{\, i, j \,\})) \\ & \qquad Cn(T) = \#T = 1 \ \&\ (\Box(i,j) : T \bullet i \rightarrow j \rightarrow init.reserve \rightarrow Skip) \\ & \qquad\quad \Box\ \#T > 1 \ \&\ IC(T) \ \Box\ init.reserve \rightarrow Skip \\ & \qquad MTask = \ \|\,[\{\, init.reserve \,\}](i,j) : Ts \bullet \\ & \qquad\quad ((i \rightarrow starts.verify \rightarrow j \rightarrow Skip \fatsemi init.reserve \rightarrow Skip) \ \Box\ init.reserve \rightarrow Skip) \\ & \ \mathbf{in}\ ((init.verify \rightarrow Skip \fatsemi \\ & \qquad (MTask \ [\![\bigcup \{\, (i,j) : Ts \{\, i, j \,\} \,\} \cup \{\, init.reserve \,\}]\!] \\ & \qquad (init.reserve \rightarrow Skip \ \Box\ Cn(Ts)))) \fatsemi P(verify)) \ \Box\ fin.1 \rightarrow Skip \end{aligned}$$

$$\begin{aligned} P(start) &= (init.verify \rightarrow Skip) \fatsemi (fin.1 \rightarrow Skip) \\ P(reserve) &= (init.reserve \rightarrow Skip \fatsemi (starts.reserve \rightarrow \\ &\quad (Reserve \ [\![\, \{\, fin.2 \,\} \,]\!]\ fin.2 \rightarrow init.booking \rightarrow Skip) \\ &\quad [\![\, \{\, init.booking \,\} \,]\!]\ init.booking \rightarrow Skip) \fatsemi P(reserve)) \\ &\quad \Box\ (fin.1 \rightarrow Skip) \\ P(booking) &= (init.booking \rightarrow Skip \fatsemi (starts.booking \rightarrow ((Booking \triangle init.timeout \rightarrow Stop) \\ &\quad [\![\, \{\, fin.3, fin.4, init.timeout \,\} \,]\!]\ (init.timeout \rightarrow Stop \\ &\quad \Box\ fin.3 \rightarrow init.notify1 \rightarrow Skip \ \Box\ fin.4 \rightarrow init.end \rightarrow Skip)) \\ &\quad [\![\, \{\, init.notify1, init.end \,\} \,]\!]\ (init.notify1 \rightarrow Skip \ \Box\ init.end \rightarrow Skip)) \fatsemi \\ &\quad P(booking)) \ \Box\ (fin.1 \rightarrow Skip) \\ P(timeout) &= (init.timeout \rightarrow Skip \fatsemi starts.timeout \rightarrow Skip \fatsemi \\ &\quad init.notify2 \rightarrow Skip \fatsemi P(notify)) \ \Box\ (fin.1 \rightarrow Skip) \\ P(notify) &= ((init.notify1 \rightarrow Skip \ \Box\ init.notify2 \rightarrow Skip) \fatsemi \end{aligned}$$

$$starts.notify \rightarrow Skip \fatsemi init.abort \rightarrow Skip \fatsemi P(notify)) \Box (fin.1 \rightarrow Skip)$$

$$P(end) = (init.end \rightarrow Skip \fatsemi fin.1 \rightarrow Skip)$$

$$P(abort) = (init.abort \rightarrow Skip \fatsemi abt.1 \rightarrow Stop) \Box (fin.1 \rightarrow Skip)$$

The process *Reserve* describes the semantics of the subprocess *Reservation* upon its syntactic description. We define set J' to index the processes corresponding to the states of the subprocess:

$$\begin{aligned}
&J' = \{\, start1, reseat, end1 \,\} \\
&Reserve = \textbf{let } X = \Box\, i : (\alpha Y \setminus \{\, fin.2 \,\}) \bullet (i \rightarrow X \,\Box\, fin.2 \rightarrow Skip) \\
&\qquad\qquad Y = (\ \|\ j : J' \bullet \alpha P(j) \circ P(j)) \\
&\qquad \textbf{in } (Y \,[\![\, \alpha Y \,]\!]\, X) \setminus \{\!|init|\!\}
\end{aligned}$$

where for each j in J', the process $P(j)$ is as defined below; we write m, ranging over $\mathbb{N}$, to denote the number of iterations in the multiple instance *Reserve Seat*:

$$\begin{aligned}
&P(start1) = (init.rseat \rightarrow Skip \fatsemi fin.2 \rightarrow Skip) \\
&P(reseat) = \\
&\ \textbf{let } X(n) = ((init.reseat \rightarrow Skip \,\Box\, init.out \rightarrow Skip) \fatsemi \\
&\qquad (n > 1 \;\&\; init.in \rightarrow X(n-1) \\
&\qquad \Box\ n = 1 \;\&\; init.in \rightarrow init.out \rightarrow init.end1 \rightarrow Skip \\
&\qquad \Box\ init.end1 \rightarrow Skip \,\Box\, n = m \;\&\; init.end1 \rightarrow Skip)) \\
&\quad A(n) = n > 0 \;\& \\
&\qquad (init.in \rightarrow Skip \fatsemi starts.reseat \rightarrow Skip \fatsemi init.out \rightarrow Skip \fatsemi A(n-1)) \\
&\qquad \Box\ init.end1 \rightarrow Skip \\
&\ \textbf{in } ((X(m) \,[\![\, \{\, init.end1, init.in, init.out \,\} \,]\!]\, A(m)) \fatsemi P(reseat)) \Box\, fin.2 \rightarrow Skip \\
&P(end1) = (init.end1 \rightarrow Skip \fatsemi fin.2 \rightarrow Skip)
\end{aligned}$$

The process *Booking* describes the semantics of the subprocess *Booking* upon its the syntactic description. It is defined as follows, where we define set J'' to index the processes corresponding to the states of the subprocess:

$$\begin{aligned}
&J'' = \{\, start2, xs3, pbooking, cancel, ticket, end3, end4 \,\} Booking = \\
&\quad \textbf{let} \quad X = \Box\, i : (\alpha Y \setminus \{\, fin.3, fin.4 \,\}) \bullet \\
&\qquad\qquad\qquad (i \rightarrow X \,\Box\, (fin.3 \rightarrow Skip \,\Box\, fin.4 \rightarrow Skip)) \\
&\qquad Y = (\ \|\ j : J'' \bullet \alpha P(j) \circ P(j)) \\
&\quad \textbf{in } (Y \,[\![\, \alpha Y \,]\!]\, X) \setminus \{\!|init|\!\}
\end{aligned}$$

where for each j in J'', the process $P(j)$ is as defined below:

$$\begin{aligned}
P(start2) &= (init.xs3 \rightarrow Skip \fatsemi P(start4)) \Box (fin.3 \rightarrow Skip \,\Box\, fin.4 \rightarrow Skip) \\
P(xs3) &= (init.xs3 \rightarrow Skip \fatsemi (init.pbooking \rightarrow Skip \,\Box\, init.cancel \rightarrow Skip) \fatsemi P(xs3)) \\
&\quad \Box\ (fin.3 \rightarrow Skip \,\Box\, fin.4 \rightarrow Skip) \\
P(pbooking) &= (init.pbooking \rightarrow Skip \fatsemi starts.pbooking \rightarrow Skip \fatsemi init.ticket \rightarrow Skip \fatsemi \\
&\quad P(pbooking)) \Box (fin.3 \rightarrow Skip \,\Box\, fin.4 \rightarrow Skip) \\
P(cancel) &= (init.cancel \rightarrow Skip \fatsemi starts.cancel \rightarrow Skip \fatsemi init.end3 \rightarrow Skip \fatsemi
\end{aligned}$$

$$
\begin{aligned}
& P(cancel)) \,\Box\, (fin.3 \rightarrow Skip \,\Box\, fin.4 \rightarrow Skip) \\
P(ticket) = {} & (init.ticket \rightarrow Skip \fatsemi starts.ticket \rightarrow Skip \fatsemi init.end4 \rightarrow Skip \fatsemi \\
& P(ticket)) \,\Box\, (fin.3 \rightarrow Skip \,\Box\, fin.4 \rightarrow Skip) \\
P(end3) = {} & (init.end3 \rightarrow Skip \fatsemi fin.3 \rightarrow Skip) \,\Box\, fin.4 \rightarrow Skip \\
P(end4) = {} & (init.end4 \rightarrow Skip \fatsemi fin.4 \rightarrow Skip) \,\Box\, fin.3 \rightarrow Skip
\end{aligned}
$$

5.2 Verifying Consistency of the Airline Reservation System

CSP's behavioural semantics admits refinement orderings under reverse containment, therefore a behavioural specification R can be expressed by constructing the most non-deterministic process satisfying it, called the characteristic process P_R. Any process Q that satisfies specification R has to refine P_R, denoted by $P_R \sqsubseteq Q$. For example, Figure 3 is a specification of the diagram in Figure 2, abstracting details of subprocesses *Reserve* and *Booking* in the original diagram in Figure 2 into a *task* state.

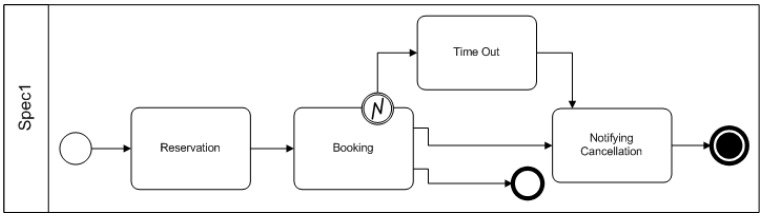

Fig. 3. A BPMN diagram describing the behavioural property defined by process *Spec*1.

Letting $K = \{\, start3, reserve2, booking2, timeout2, notify2, abort1, end1 \,\}$, the process *Spec*1 is defined as follows:

$$
\begin{aligned}
Spec1 = {} & \textbf{let} \\
& \quad X = \Box\, i : (\alpha Y \setminus \{fin.1, abt.1\}) \bullet \\
& \qquad\qquad (i \rightarrow X \,\Box\, (abt.1 \rightarrow Stop) \,\Box\, (fin.1 \rightarrow Skip)) \\
& \quad Y = \;\|\; x : K \bullet \alpha P(x) \circ P(x) \\
& \textbf{in}\ (Y \,[\![\, \alpha Y \,]\!]\, X) \setminus \{\!|init|\!\}
\end{aligned}
$$

where for each k in K, the process $P(k)$ is as defined below:

$$
\begin{aligned}
P(start3) = {} & (init.reserve2 \rightarrow Skip) \fatsemi (fin.1 \rightarrow Skip) \\
P(reserve2) = {} & ((init.reserve2 \rightarrow Skip) \fatsemi starts.reserve \rightarrow Skip \fatsemi init.booking2 \rightarrow Skip \fatsemi \\
& P(reserve2)) \,\Box\, (fin.1 \rightarrow Skip) \\
P(booking2) = {} & (init.booking2 \rightarrow Skip \fatsemi starts.booking \rightarrow (Skip \,\triangle\, init.timeout2 \rightarrow Stop) \fatsemi \\
& (init.end1 \rightarrow Skip \,\Box\, init.notify2 \rightarrow Skip) \fatsemi P(booking2)) \,\Box\, (fin.1 \rightarrow Skip)
\end{aligned}
$$

$$P(timeout2) = ((init.timeout2 \rightarrow Skip) \fatsemi starts.timeout \rightarrow Skip \fatsemi init.notify3 \rightarrow Skip \fatsemi P(timeout2)) \Box (fin.1 \rightarrow Skip)$$

$$P(notify2) = ((init.notify2 \rightarrow Skip \Box init.notify3 \rightarrow Skip) \fatsemi starts.reserve \rightarrow Skip \fatsemi init.abort1 \rightarrow Skip \fatsemi P(notify2)) \Box (fin.1 \rightarrow Skip)$$

$$P(end1) = (init.end1 \rightarrow Skip \fatsemi fin.1 \rightarrow Skip)$$

$$P(abort1) = (init.abort1 \rightarrow Skip \fatsemi abt.1 \rightarrow Stop) \Box (fin.1 \rightarrow Skip)$$

Note that CSP's traces model is insufficient to verify our models against formal specifications. If we insist on using the traces model, then under traces refinement any process P that has the trace-set $\{\langle\rangle\}$ will refine and hence satisfy process *Spec*1. Any process which corresponds to a broken or an illegal BPMN diagram might in fact have this trace-set; this demonstrates the inadequacy of the traces model. We therefore use the stable failures model to compare process *Airline* with *Spec*1.

$$Spec1 \sqsubseteq_{\mathcal{F}} Airline \setminus (\alpha Airline \setminus \alpha Spec1)$$

This refinement captures the claim that our semantic model is consistent with respect to different levels of abstraction and *Airline* is indeed a refinement of the abstraction defined by *Spec*1. Due to the specific semantic definition presented in this paper, we are able to verify refinement assertions such as this by model checking using FDR [7].

The above refinement assertion motivates the following generalisation of 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 1. ***Hierachical 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* ***hierachically 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 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:*

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

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 neccessary to also abstract details which are irrelevant to the behavioural property we are interested in.

Definition 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 diagram in Figure 2 is a partial refinement of the diagram in Figure 3. Conversely we say the diagram in Figure 3 is a partial specification of the diagram in Figure 2. Moreover, these refinement claims may be checked automatically by FDR. These relationships allow a business process developer to focus on the specification of part of the diagram.

6 Conclusion

In this paper, we have presented a process semantics in the language of CSP for a subset of BPMN. We have illustrated by examples how this semantic model may be used to verify that one BPMN diagram is consistent with another, which might be its abstract specification using the same graphical notation. Our semantic model makes it possible to formally analyse and compare BPMN diagrams, and to assert correctness conditions that can be verified using a model checker. Like any development of a complex system, the application of refinement in business process design means that development from an abstract design into an implementation becomes incremental.

The CSP process semantics of a BPMN workflow can be constructed automatically from a simple syntactic presentation of the diagram. We have used Z as a syntactic vehicle, but something like XMI would work too. We do not expect the designer to write in this syntax directly, but to generate it from the diagrammatic notation, annotated with attribute values such as guards and multiplicities.

Future work will include augmenting our semantics with a well-defined transaction and compensation handling, perhaps building on Butler's compensating CSP [3], to provide a formal semantics for the complete BPMN; formalising Property Specification Patterns [6] in CSP, specifically to allow such patterns to be employed for reasoning about behavioural properties of BPMN processes; and automating the semantic translation to facilitate automatic verification.

This work is supported by a grant from Microsoft Research.

References

1. C. Bolton and J. Davies. Activity graphs and processes. In *Proceedings of the Second International Conference on Integrated Formal Methods*, pages 77–96, 2000.
2. Business Process Execution Language for Web Services, Version 1.1., May 2003. `http://www.ibm.com/developerworks/library/ws-bpel`.
3. M. Butler, T. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *Proceedings of 25 Years of CSP*, volume 3525 of *LNCS*, 2005.
4. R. M. Dijkman. Choreography-Based Design of Business Collaborations. BETA Working Paper WP-181, Eindhoven University of Technology, 2006.
5. R. M. Dijkman, M. Dumas, and C. Ouyang. Formal semantics and automated analysis of BPMN process models. Technical Report Preprint 5969, Queensland University of Technology, 2007.
6. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-state Verification. In *2nd Workshop on Formal Methods in Software Practice*, 1998.
7. Formal Systems (Europe) Ltd. *Failures-Divergences Refinement, FDR2 User Manual*, 1998. `www.fsel.com`.
8. H. Foster. Mapping BPEL4WS to FSP. Technical report, Imperial College, London, 2003.
9. ILOG JViews BPMN Modeler. Available at `http://www.ilog.com/`.
10. J. Cámara, C. Canal, J. Cubo, and A. Vallecillo. Formalizing WSBPEL Business Processes using Process Algebra, Aug. 2005. CONCUR'2005 Workshop on the Foundations of Coordination Languages and Software Architectures.
11. M. Koshkina. Verification of business processes for web services. Master's thesis, York University, Toronto, Oct. 2003.
12. R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1), Jan. 2007.
13. OMG. *Business Process Modeling Notation (BPMN) Specification*, Feb. 2006. `www.bpmn.org`.
14. C. Ouyang, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Translating BPMN to BPEL. Technical Report BPM-06-02, BPM Center, 2006.
15. J. Recker and J. Mendling. On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In *Proceedings 18th International Conference on Advanced Information Systems Engineering*, pages 521–532, 2006.
16. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.
17. W3C. *Web Service Choreography Interface (WSCI) 1.0*, Nov. 2002. `www.w3.org/TR/wsci`.
18. P. Y. H. Wong and J. Gibbons. A Process Semantics for BPMN (extended version). `www.comlab.ox.ac.uk/peter.wong/pub/bpmnsem.pdf`, 2007.
19. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.