A Process Semantics for BPMN

Peter Y.H. Wong Jeremy Gibbons

July 2007

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. Two simple examples of business processes are included.

1 Introduction

Modelling of business processes and workflows is an important area in software engineering. 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BPMN has been specified to map directly to the BPML standard, which has subsequently been superceded by WS-BPEL [1]. To the best of our knowledge the only previous attempt at defining a formal semantics for a subset of BPMN [4, 5] did so using Petri nets. 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 WS-BPEL and BPMN [14, 18, 15], and the formal semantics of WS-BPEL [9, 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 this 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 [6]. Our semantic construction starts from syntax expressed in Z [19], following Bolton and Davies's work on UML activity graphs [2].

This paper begins with an introduction to BPMN and the mathematical notations, Z [19] and CSP [16], that are used throughout the document. 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 may be verified. In Section 6 we present another simple example to demonstrate how *compatibility* between the participants in a business collaboration may be verified. 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 flow represents a specific control flow instance of the business process.

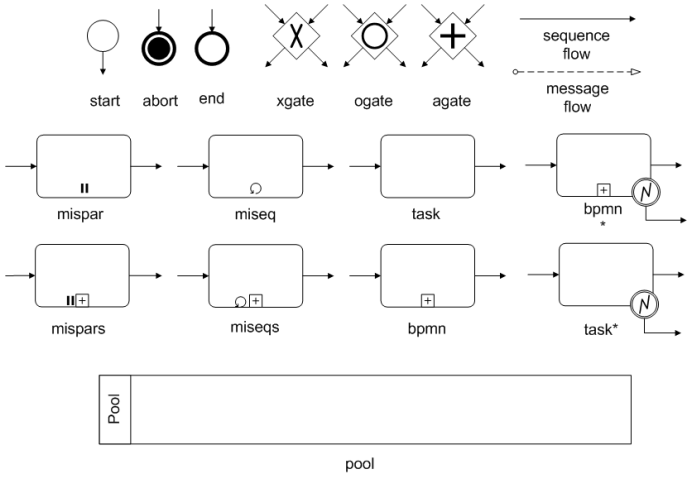

Figure 1: States of BPMN diagram

A table showing each type of state is presented in Figure 1. 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.

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. An illustration of message flow between activities across pools is shown in Figure 2. In the figure, task *A sends*

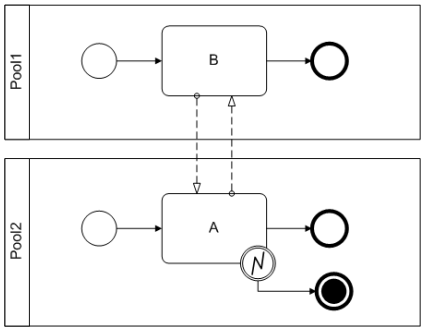

Figure 2: Interaction via message flows

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

2.2 Z

The Z notation [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:

Name
declaration
constraint

or equivalently

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

If S is a schema then θS denotes the characteristic binding of S in which each component is associated with its current value. Schemas can be used as

declarations. For example, the lambda expression $\lambda S \bullet t$ denotes a function from the schema type underlying S, a set of bindings, to the type of term expression t.

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

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

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

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

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.

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

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 $\Vert\Vert\, i : I \bullet P(i)$, $\Vert [A]\, i : I \bullet P(i)$ and $\Vert\, i : I \bullet A(i) \circ P(i)$ 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]. Notable is the semantic equivalence of processes $P \,\Box\, Q$ and $P \sqcap Q$ under the traces model. In order to distinguish these processes, it is necessary to record not only what a process can do, but also what it can refuse to do. This information is preserved in *refusal sets*, sets of events from which a process in a stable state can refuse to communicate no matter how long it is offered. The set $\mathit{refusals}(P)$ is P's initial refusals. A failure therefore is a pair (s, X) where $s \in \mathit{traces}(P)$ is a trace of P leading to a stable state and $X \in \mathit{refusals}(P/s)$ where P/s represents process P after the trace s. We write $\mathit{traces}(P)$ and $\mathit{failures}(P)$ as the set of all P's traces and failures respectively.

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

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

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

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

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 [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 Syntactic Description of BPMN

In this section we describe the abstract syntax of BPMN using Z schemas and the set theory, and use an example in Section 3.2 to show how the syntax can be applied on a given BPMN diagram.

3.1 Abstract Syntax

We intially define the following basic types:

$$[\mathit{CName}, \mathit{PName}, \mathit{Task}, \mathit{Line}, \mathit{Channel}, \mathit{Guard}, \mathit{Message}]$$

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

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

Each type of state shown in Figure 1 is defined syntactically as follows:

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

According to the specification [13], each BPMN state type has associated attributes describing its properties; our syntactic definition has included some of these attributes. For example, the number of loops of a sequence multiple instance state type is recorded by the natural number in the constructor function *miseq*. We define abbreviations *Inputs*, *NoEnds*, *Tasks*, *Subs* and *Mults* as follows to assist our specification.

$$
\begin{aligned}
Inputs &== InMsg \cup EndMsg \cup LastMsg \\
NoEnds &== InMsg \cup LastMsg \\
Tasks &== \mathrm{ran}\, task \cup \mathrm{ran}\, miseq \cup \mathrm{ran}\, mipar \\
Subs &== \mathrm{ran}\, bpmn \cup \mathrm{ran}\, miseqs \cup \mathrm{ran}\, mipars \\
Mults &== \mathrm{ran}\, miseqs \cup \mathrm{ran}\, mipars \cup \mathrm{ran}\, miseq \cup \mathrm{ran}\, mipar
\end{aligned}
$$

In this paper we call both sequence flows and exception flows 'transitions'; states are linked by transition lines representing flows of control, which may have associated guards. We give the type of a sequence flow or an exception flow by the following schema definition.

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

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

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

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

$$
\begin{array}{|l}
tt : Guard;\ empty : Message
\end{array}
$$

Each *state* records the type of its content, the sets of incoming, outgoing and error transitions and in the case of a subprocess state, a set of number-transition pairs to align the outgoing transitions of the subprocess within the outgoing transitions within the subprocess. There are also the five sets of message flows; their informal meanings are illustrated by the simple BPMN diagram in Figure 2. Each state also incorporates the variable *loopMax* to limit the number of state instances each process instance can invoke; the schema *State* records the default value 0 if there is no limit to the number of state instances. The schema

component *link* pairs each incoming message flow which initialises or interrupts the execution of the state with either an incoming transition or an exception flow; the component *depend* pairs each incoming message flow which initialise the state's execution with its corresponding outgoing message flow.

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

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

WFS

$State$

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

We introduce the notion of *well-configured* set by first defining functions *isStart* and *isEnd*, which decide respectively whether a state within a well-configured set is reachable from some start state and reaches to some end or abort state. The function *fSeq* returns a finite sequence upon some finite set, containing exactly those elements in the set.

$[X]$

$$fSeq : \mathbb{P}\ X \to (\operatorname{seq} X)$$

$$\forall\, xs : \mathbb{P}\ X \bullet \operatorname{dom}(fSeq\ xs) = 1\,..\,\#xs \land \operatorname{ran}(fSeq\ xs) = xs$$

The functions *findBState* and *findFState* recursively locate the state which connects the incoming and outgoing tranitions of the argument state from the input sequence of states respectively.

$$findBState, findFState : WFS \pfun \seq WFS \pfun \seq WFS$$

$$\begin{array}{l}
\forall\, w : WFS;\ ws : \seq WFS \bullet \\
findBState\ w\ ws = \\
\quad \textbf{if}\ (ws = \langle\rangle)\ \textbf{then}\ \langle\rangle \\
\quad \textbf{else}\ (\textbf{if}\ (((head\ ws).out \cup (head\ ws).error) \cap w.in) \neq \emptyset \\
\qquad \textbf{then}\ \langle head\ ws\rangle\ \textbf{else}\ findBState\ w\ (tail\ ws)) \\
\land\ findFState\ w\ ws = \\
\quad \textbf{if}\ (ws = \langle\rangle)\ \textbf{then}\ \langle\rangle \\
\quad \textbf{else}\ (\textbf{if}\ ((head\ ws).in \cap (w.out \cup w.error)) \neq \emptyset \\
\qquad \textbf{then}\ \langle head\ ws\rangle\ \textbf{else}\ findFState\ w\ (tail\ ws))
\end{array}$$

$$isStart, isEnd : WFS \pfun \seq WFS \pfun \nat$$

$$\begin{array}{l}
\forall\, w : WFS;\ ws : \seq WFS \bullet \\
isStart\ w\ ws = \\
\quad \textbf{if}\ (w.type = start)\ \textbf{then}\ 1 \\
\quad \textbf{else if}\ (ws = \langle\rangle)\ \textbf{then}\ 0 \\
\qquad \textbf{else if}\ (findBState\ w\ ws = \langle\rangle)\ \textbf{then}\ 0 \\
\qquad\quad \textbf{else}\ isStartm\ (findBState\ w\ ws)\ (squash\ (ws \rhd \{\ w\ \})) \\
\land\ isEnd\ w\ ws = \\
\quad \textbf{if}\ (w.type \in \ran end \cup \ran abort)\ \textbf{then}\ 1 \\
\quad \textbf{else if}\ (ws = \emptyset)\ \textbf{then}\ 0 \\
\qquad \textbf{else if}\ (findFState\ w\ (fSeq\ ws) = \langle\rangle)\ \textbf{then}\ 0 \\
\qquad\quad \textbf{else}\ isEndm\ (findFState\ w\ ws)\ (squash\ (ws \rhd \{\ w\ \}))
\end{array}$$

$$isStartm, isEndm : \seq WFS \pfun \seq WFS \pfun \nat$$

$$\begin{array}{l}
\forall\, ws, xs : \seq WFS \bullet \\
\quad isStartm\ ws\ xs = \\
\qquad \textbf{if}\ (ws = \langle\rangle)\ \textbf{then}\ 0 \\
\qquad \textbf{else if}\ ((isStart\ (head\ ws)\ xs) = 1)\ \textbf{then}\ 1 \\
\qquad\quad \textbf{else}\ isStartm\ (tail\ ws)\ xs \\
\quad \land\ isEndm\ ws\ xs = \\
\qquad \textbf{if}\ (ws = \langle\rangle)\ \textbf{then}\ 0 \\
\qquad \textbf{else if}\ ((isEnd\ (head\ ws)\ xs) = 1)\ \textbf{then}\ 1 \\
\qquad\quad \textbf{else}\ isEndm\ (tail\ ws)\ xs
\end{array}$$

The set of *well-configured* sets of well-formed states WCF is hence defined axiomatically as follows:

$$WCF : \power(\power State)$$

$$\begin{array}{l}
WCF = \{\ C : \power State \mid (C \in \finset_1 WFS \\
\quad \land\ \{\ s : State \mid s.type \in \ran pool\ \} \cap C = \emptyset \\
\quad \land\ \{\ s : State \mid s.type = start\ \} \cap C \neq \emptyset \\
\quad \land\ \{\ s : State \mid s.type \in (\ran end \cup \ran abort)\ \} \cap C \neq \emptyset \\
\quad \land\ (\forall\, a : State \bullet a \in C \land (a.type \notin (\ran end \cup \ran abort) \Leftrightarrow isEnd\ a\ (fSeq\ (C \setminus \{\ a\\})) = 1)) \\
\quad \land\ (\forall\, a : State \bullet a \in C \land (a.type \neq start \Leftrightarrow isStart\ a\ (fSeq\ (C \setminus \{\ a\\})) = 1)) \\
\quad \land\ (\forall\, s, t : State \bullet \{\ s.type, t.type\ \} \subseteq \ran end \Rightarrow end^\sim s.type \neq end^\sim t.type) \\
\quad \land\ (\forall\, s, t : State \bullet \{\ s.type, t.type\ \} \subseteq \ran abort \Rightarrow abort^\sim s.type \neq abort^\sim t.type))\ \} \\
\quad \land\ (\forall\, s, t : State \bullet s \neq t \Rightarrow (s.in \cap t.in = s.out \cap t.out = \emptyset))
\end{array}$$

Each BPMN diagram encapsulated by a *pool* represents an individual participant in a collaboration, built up from a well-configured finite set of well-formed states. We do not allow local states to have type *pool*, since this represents a boundary of a business domain. The function *Local* represents the environment of the local specification and maps each name of a BPMN diagram to its associated diagram. However, a collaboration is built up from a finite set of names, each of the names is associated with a BPMN diagram and as such the function *Global* represents the environment of a global specification and maps each collaboration name to its associated diagram's name.

$$BPD ::= states\langle\langle WCF \rangle\rangle \qquad Local == PName \pfun BPD$$
$$Chor ::= bpmns\langle\langle \mathbb{F}\, PLName \rangle\rangle \qquad Global == CName \pfun Chor$$

3.2 An Example

We present an example of a business process of an airline reservation system shown in Figure 3, this example has been taken from the WSCI specification [17]. It could be assumed to have been constructed during the development of the business process. We have abstracted message flows, as there is only one business participant in the example. We use this example to illustrate how a BPMN diagram can be translated into a well-configured set of states describing the diagram's syntax.

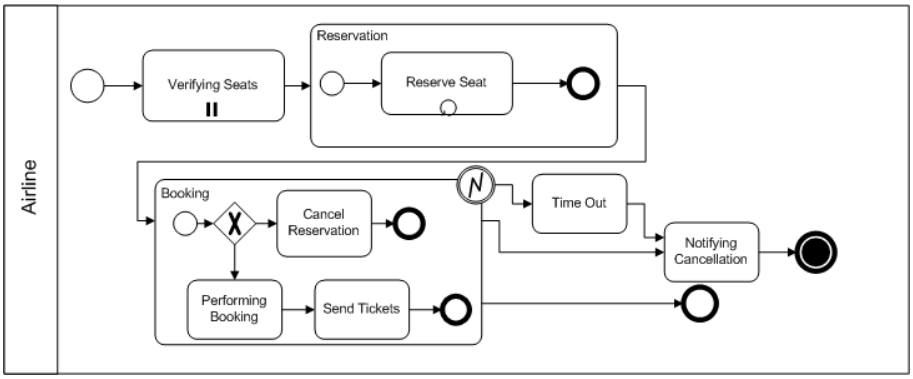

Figure 3: A BPMN diagram describing the workflow of a airline reservation application.

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 *Transition* and *Messageflow*. We write $a_1 \,..\, a_n \rightsquigarrow \emptyset$ inside some schema binding s to specify the components $s.a_1 \,..\, s.a_n$ to be empty. The syntactic detail of the subprocesses *Reserve* and *Booking* are also omitted.

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

4 Behavioural Semantics of BPMN

Section 3 presented an 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. In Section 4.1 we define functions to associate each transition, messageflow, 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 and each set of BPMN diagrams to its process describing the diagrams' behaviour and interconnection; in Section 4.3 we define the functions for mapping each type of multiple instance states to its process describing its behaviourn; in Section 4.4 we present the CSP processes corresponding to the behaviour of each gateways and message flows; and in Section 4.5 we define processes corresponding to the behaviour of each state types and transitions, and the general functions for mapping each BPMN states to its CSP process describing its behaviour.

4.1 Alphabets

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

$$[Process, Event, Data]$$

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

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

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

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

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

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

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

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

4.2 Processes corresponding to Local and Global 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.

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

$$
\begin{array}{l}
bsm : PName \pfun Local \pfun Process \\
\hline
\forall\, p : PName;\ local : Local;\ c : CName;\ global : Global \bullet \\
bsm\ p\ local = (\ \|\ s : \{\, s : (states^{\sim}(local\ p)) \mid s.type \neq start \,\} \bullet \\
\quad (\alpha_{state}\ s\ local \cup \{\, i : \epsilon_{end}\ p\ local \bullet fin.i \,\} \cup \\
\quad (\mathbf{if}\ (s.type \in \operatorname{ran} abort)\ \mathbf{then}\ \{\, abt.(abort^{\sim} s.type) \,\}\ \mathbf{else}\ \emptyset)) \circ \\
\quad \mathbf{if}\ (s.type \in \operatorname{ran} end) \\
\quad \mathbf{then}\ ((\rho_{state}\ s \fatsemi fin.(end^{\sim} s.type) \rightarrow Skip) \\
\qquad \Box\ (\Box\, e : \epsilon_{end}\ p\ local \setminus \{\, end^{\sim} s.type \,\} \bullet fin.e \rightarrow Skip)) \\
\quad \mathbf{else\ if}\ (s.type \in \operatorname{ran} abort) \\
\qquad \mathbf{then}\ ((\rho_{state}\ s \fatsemi abt.(abort^{\sim} s.type) \rightarrow Stop) \ \Box\ \rho_{end}\ p\ local) \\
\qquad \mathbf{else}\ \mathsf{let}\ X = ((\rho_{state}\ s \ \Box\ \rho_{extmsg}\ s.receive\ EndMsg \fatsemi X) \ \Box\ \rho_{end}\ p\ local) \\
\qquad\quad \mathsf{within}\ (\mathbf{if}\ (s.loopMax = 0)\ \mathbf{then}\ X \\
\qquad\qquad\quad \mathbf{else}\ (X \,[\![\, \alpha_{msgtype}\ s.receive\ NoEnds \\
\qquad\qquad\qquad\quad \cup\, \alpha_{trans}\ s.in \cup \{\, i : \epsilon_{end}\ p\ local \bullet fin.i \,\}]\!] \\
\qquad\qquad\quad \rho_{loop}\ p\ s\ local)))) \\
\quad [\![\, \alpha_{start}\ p\ local \cup \{\, i : \epsilon_{end}\ p\ local \bullet fin.i \,\}]\!] \\
\quad \Box\, s : \{\, s : states^{\sim}(local\ p) \mid s.type = start \,\} \bullet (\rho_{state}\ s \fatsemi \rho_{end}\ p\ local))
\end{array}
$$

We define the function *csem*, which takes a syntactic description of one or more states of type *pool*, each encapsulating a separate BPMN diagram representing an individual participant within a business collaboration, and returns a parallel composition of processes, each corresponding to an individual participant.

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

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.

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

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

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

4.3 Processes corresponding to Multiple Instances

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

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

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

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

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

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

The process *Cn* takes the set of event-pairs, each corresponding to the incoming and outgoing transitions of an instance defined in *MTask*, and returns the process that controls the multiple instances in *MTask*. If the multiple instance state's *receive* component is empty, then it internally controls the number of instances to trigger, otherwise it controls the number of instances according to the message received through the set of message flows specified by the component *receive*. The control process keeps a counter of the number of instances triggered.

The process *CL* takes a set of transition-pairs and a set of message flows specified in *send* and recursively sends messages of type either *InMsg* or *LastMsg* along the channels specified by the component *send*. If all of the messages are

of type *LastMsg*, then *CL* triggers one of the outgoing transitions and the whole multiple instance state terminates, otherwise it behaves as the process *Cn*.

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

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

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

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

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

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

$$\rho_{intermsg}, \rho_{extmsg} : \mathbb{P}\, Messageflow \nrightarrow \mathbb{P}\, Message \nrightarrow Process$$
$$\alpha_{msgtype} : \mathbb{P}\, Messageflow \nrightarrow \mathbb{P}\, Message \nrightarrow \mathbb{P}\, Event$$

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

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

$$\rho_{mierror} : State \nrightarrow Process$$

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

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

$$mipartst : State \rightarrowtail \mathbb{P}(Transition \times Transition)$$
$$miseqtst : State \rightarrowtail (Transition \times Transition)$$

$$\forall\, s : State \bullet \exists\, n : \mathbb{N};\ t1, t2 : Transition;\ ts : \mathbb{P}(Transition \times Transition) \bullet$$
$$((s.type \in \operatorname{ran} mipar \Rightarrow n = (mipar^{\sim} s.type).2$$
$$\land\ s.type \in \operatorname{ran} mipars \Rightarrow n = (mipars^{\sim} s.type).2)$$
$$\land\ ((\#ts = \#\{\, (i,j) : ts \bullet i\,\} = \#\{\, (i,j) : ts \bullet j\,\} = n)$$
$$\land\ (\{\, (i,j) : ts \bullet i\,\} \cap \{\, (i,j) : ts \bullet j\,\} = \emptyset))$$
$$\land\ mipartst\ s = ts) \land miseqtst\ s = (t1, t2)$$

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

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

$$\forall\, s : State;\ local : Local \bullet \exists\, t1, t2 : Transition;\ e1, e2 : Event;\ n : \mathbb{N} \bullet$$
$$(t1, t2) = miseqtst\ s \land (e1, e2) = ((\epsilon_{trans}\ t1).1, (\epsilon_{trans}\ t2).1)$$
$$\land\ (\textbf{if}\ s.type \in \operatorname{ran} miseq\ \textbf{then}\ n = (miseq^{\sim} s.type).2\ \textbf{else}\ n = (miseqs^{\sim} s.type).2)$$
$$\land\ \rho_{miseq}\ s\ local =$$
$$\textsf{let}$$
$$SY = \bigcup\{\, \alpha_{trans}(s.out \cup s.error), \{\, e1, e2\,\}, \{\, i : \alpha_{chn}\ s.receive;\ j : NoEnds \bullet i.j\,\},$$
$$\{\, i : \alpha_{chn}(s.send \cup s.break);\ j : InMsg \bullet i.j\,\}\,\}$$
$$\textsf{within}$$
$$(Cq(n, n, s, e1, e2) \triangle (\textbf{if}\ s.break = \emptyset\ \textbf{then}\ AJ(s.error)\ \textbf{else}\ \rho_{mierror}\ s))$$
$$[\![\, SY\,]\!]\ Seq(n, s, local)$$

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

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

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

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

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

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

4.4 Processes corresponding to Gateways and Message flows

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

4.4.1 Exclusive Choice Gateway

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

by the behaviour of the preceding states.

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

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

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

4.4.2 Parallel Gateway

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

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

4.4.3 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.

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

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

4.4.4 Message Flow Interaction

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

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

4.5 Processes corresponding to Transitions, Types and States

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

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

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

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

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

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

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

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

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

5 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 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.

$$
\begin{array}{rl}
Airline = \mathsf{let}\ X = \Box\, i : (\alpha\, Y \setminus \{\, fin.1, abt.1 \,\}) \bullet & \\
\qquad\qquad\qquad\qquad (i \rightarrow X \ \Box\ abt.1 \rightarrow Stop \ \Box\ fin.1 \rightarrow Skip) & \\
\qquad\qquad Y = (\ \|\, j : J \bullet \alpha P(j) \circ P(j)) & \\
\qquad\quad \mathsf{within}\, (Y \lpar\!| \alpha\, Y |\!\rpar X) \setminus \{\!| init |\!\} &
\end{array}
$$

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*.

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

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

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:

$$J' = \{\, start1, reseat, end1 \,\}$$

$$
\begin{array}{l}
Reserve = \mathsf{let}\; X = \Box\, i : (\alpha Y \setminus \{\, fin.2 \,\}) \bullet (i \rightarrow X \;\Box\; fin.2 \rightarrow Skip) \\
\qquad\qquad Y = (\; \|\, j : J' \bullet \alpha P(j) \circ P(j)) \\
\qquad\quad \mathsf{within}\; (Y \;[\![\, \alpha Y \,]\!]\; X) \setminus \{\!|\, init \,|\!\}
\end{array}
$$

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*:

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

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:

$$J'' = \{\, start2, xs3, pbooking, cancel, ticket, end3, end4 \,\}$$

$$
\begin{array}{l}
Booking = \\
\quad \mathsf{let} \\
\qquad X = \Box\, i : (\alpha Y \setminus \{\, fin.3, fin.4 \,\}) \bullet \\
\qquad\qquad\quad (i \rightarrow X \;\Box\; (fin.3 \rightarrow Skip \;\Box\; fin.4 \rightarrow Skip)) \\
\qquad Y = (\; \|\, j : J'' \bullet \alpha P(j) \circ P(j)) \\
\quad \mathsf{within} \\
\qquad (Y \;[\![\, \alpha Y \,]\!]\; X) \setminus \{\!|init|\!\}
\end{array}
$$

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

$$
\begin{array}{rcl}
P(start2) & = & (init.xs3 \rightarrow Skip \fatsemi P(start4)) \;\Box\; (fin.3 \rightarrow Skip \;\Box\; fin.4 \rightarrow Skip) \\
P(xs3) & = & (init.xs3 \rightarrow Skip \fatsemi (init.pbooking \rightarrow Skip \;\Box\; init.cancel \rightarrow Skip) \fatsemi P(xs3)) \\
 & & \Box\; (fin.3 \rightarrow Skip \;\Box\; fin.4 \rightarrow Skip) \\
P(pbooking) & = & (init.pbooking \rightarrow Skip \fatsemi starts.pbooking \rightarrow Skip \fatsemi init.ticket \rightarrow Skip \fatsemi \\
 & & P(pbooking)) \;\Box\; (fin.3 \rightarrow Skip \;\Box\; fin.4 \rightarrow Skip) \\
P(cancel) & = & (init.cancel \rightarrow Skip \fatsemi starts.cancel \rightarrow Skip \fatsemi init.end3 \rightarrow Skip \fatsemi \\
 & & P(cancel)) \;\Box\; (fin.3 \rightarrow Skip \;\Box\; fin.4 \rightarrow Skip) \\
P(ticket) & = & (init.ticket \rightarrow Skip \fatsemi starts.ticket \rightarrow Skip \fatsemi init.end4 \rightarrow Skip \fatsemi \\
 & & P(ticket)) \;\Box\; (fin.3 \rightarrow Skip \;\Box\; fin.4 \rightarrow Skip) \\
P(end3) & = & (init.end3 \rightarrow Skip \fatsemi fin.3 \rightarrow Skip) \;\Box\; fin.4 \rightarrow Skip \\
P(end4) & = & (init.end4 \rightarrow Skip \fatsemi fin.4 \rightarrow Skip) \;\Box\; fin.3 \rightarrow Skip
\end{array}
$$

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 4 is a specification of the diagram in Figure 3, abstracting details of subprocesses *Reserve* and *Booking* in the original diagram in Figure 3 into a *task* state.

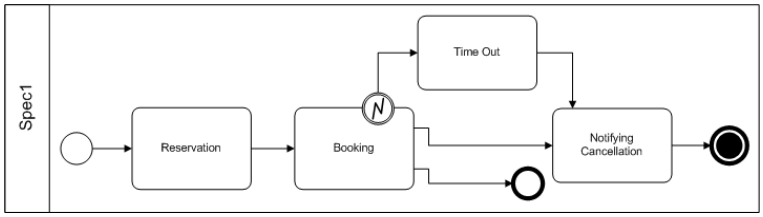

Figure 4: A BPMN diagram describing behavioural property defined by process *Spec*1.

Letting $K = \{\, start3, reserve2, booking2, timeout2, notify2, abort1, end1 \,\}$, the process *Spec*2 is defined as follows:

$$
\begin{array}{l}
Spec2 = \mathsf{let} \\
\qquad X = \Box\, i : (\alpha Y \setminus \{fin.1, abt.1\}) \bullet \\
\qquad\qquad (i \rightarrow X \;\Box\; (abt.1 \rightarrow Stop) \;\Box\; (fin.1 \rightarrow Skip)) \\
\qquad Y = \;\|\; x : K \bullet \alpha P(x) \circ P(x) \\
\mathsf{within}\; (Y \;[\![\, \alpha Y \,]\!]\; X) \setminus \{\!|init|\!\}
\end{array}
$$

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

$$
\begin{array}{rcl}
P(start3) & = & (init.reserve2 \rightarrow Skip) \fatsemi (fin.1 \rightarrow Skip) \\
P(reserve2) & = & ((init.reserve2 \rightarrow Skip) \fatsemi starts.reserve \rightarrow Skip \fatsemi init.booking2 \rightarrow Skip \fatsemi \\
 & & P(reserve2)) \;\Box\; (fin.1 \rightarrow Skip) \\
P(booking2) & = & (init.booking2 \rightarrow Skip \fatsemi starts.booking \rightarrow (Skip \;\triangle\; init.timeout2 \rightarrow Stop) \fatsemi \\
 & & (init.end1 \rightarrow Skip \;\Box\; init.notify2 \rightarrow Skip) \fatsemi P(booking2)) \;\Box\; (fin.1 \rightarrow Skip) \\
P(timeout2) & = & ((init.timeout2 \rightarrow Skip) \fatsemi starts.timeout \rightarrow Skip \fatsemi init.notify3 \rightarrow Skip \fatsemi \\
 & & P(timeout2)) \;\Box\; (fin.1 \rightarrow Skip) \\
P(notify2) & = & ((init.notify2 \rightarrow Skip \;\Box\; init.notify3 \rightarrow Skip) \fatsemi \\
 & & starts.reserve \rightarrow Skip \fatsemi init.abort1 \rightarrow Skip \fatsemi P(notify2)) \;\Box\; (fin.1 \rightarrow Skip) \\
P(end1) & = & (init.end1 \rightarrow Skip \fatsemi fin.1 \rightarrow Skip) \\
P(abort1) & = & (init.abort1 \rightarrow Skip \fatsemi abt.1 \rightarrow Stop) \;\Box\; (fin.1 \rightarrow Skip)
\end{array}
$$

As mentioned in Section 2.3 the 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 following trace-set will refine and hence satisfy process *Spec*1.

$$traces(P) = \{\, \langle\rangle \,\}$$

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 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 check tells us 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. We have specifically defined our semantics to allow refinement checks such as this one to be readily performed by a model checker like FDR [6].

6 Business Collaboration

In previous sections we have demonstrated how consistency between different levels of abstraction may be verified. In this section we show how our semantics also allows formal reasoning about business-to-business collaboration, where there are multiple business processes under consideration.

6.1 An Example: Airline Ticket Reservation

Consider, for instance, the simple example of an airline ticket reservation shown in Figure 5.

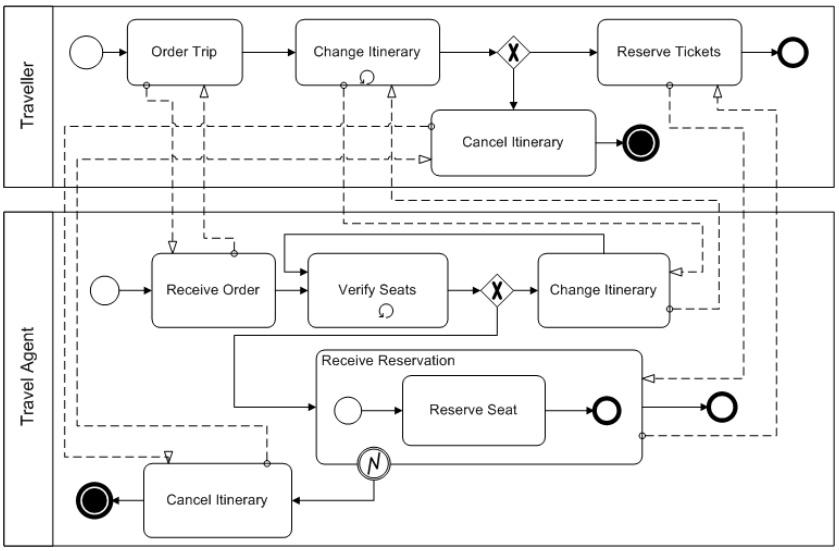

Figure 5: A business collaboration of an airline ticket reservation.

The figure depicts the message flows between two participants, the traveller and the travel agent, which are independent business processes and may be assumed to have been constructed separately during the development process. Clearly a necessary behavioural property for a successful collaboration is *compatibility* between the participants: the mutually consistency of the assumptions each makes about their interaction. For example, from the traveller participant's perspective, the behaviour of interest is the ability to cancel an

itinerary by sending a message to the travel agent participant prior to making her ticket reservation, while from the travel agent participant's perspective such a sequence of tasks might not be allowed.

Given the business process name *traveller*, the following shows a set of well-formed states translated from the diagram describing the traveller participant. We have omitted details of the bindings of *Transition* and *Messageflow*. Similar to Section 3.2, we write $a_1 \,..\, a_n \rightsquigarrow \emptyset$ inside some schema binding s to specify the components $s.a_1 \,..\, s.a_n$ to be empty.

$$
\begin{array}{|l}
traveller : PName;\ order, change, cancel, reserve : Task \\
\hline
\exists\, local : Local;\ t1, t2, t3, t4, t5, t6, t7 : Transition;\ i, j, k : \mathbb{N}; \\
\quad m1, m2, m3, m4, m5, m6, m7, m8 : Messageflow \bullet \\
\quad states^{\sim}(local\ traveller) = \\
\qquad \{ \langle type \rightsquigarrow start, out \rightsquigarrow \{ t1 \}, loopMax \rightsquigarrow 0, \\
\qquad\quad break, send, receive, accept, reply, in, link, depend, error, exit \rightsquigarrow \emptyset \rangle, \\
\qquad\quad \langle type \rightsquigarrow task\ order, in \rightsquigarrow \{ t1 \}, out \rightsquigarrow \{ t2 \}, send \rightsquigarrow \{ m1 \}, \\
\qquad\quad\quad accept \rightsquigarrow \{ m2 \}, break, receive, reply, error, exit, link, depend \rightsquigarrow \emptyset, \\
\qquad\quad\quad loopMax \rightsquigarrow 0 \rangle, \\
\qquad\quad \langle type \rightsquigarrow miseq\ change\ k, in \rightsquigarrow \{ t2 \}, out \rightsquigarrow \{ t3 \}, send \rightsquigarrow \{ m3 \}, \\
\qquad\quad\quad accept \rightsquigarrow \{ m4 \}, break, receive, reply, error, exit, link, depend \rightsquigarrow \emptyset, \\
\qquad\quad\quad loopMax \rightsquigarrow 0 \rangle, \\
\qquad\quad \langle type \rightsquigarrow xs, in \rightsquigarrow \{ t3 \}, out \rightsquigarrow \{ t4, t5 \}, loopMax \rightsquigarrow 0, \\
\qquad\quad\quad accept, send, break, receive, reply, error, exit, link, depend \rightsquigarrow \emptyset \rangle, \\
\qquad\quad \langle type \rightsquigarrow task\ cancel, in \rightsquigarrow \{ t4 \}, out \rightsquigarrow \{ t6 \}, send \rightsquigarrow \{ m5 \}, \\
\qquad\quad\quad accept \rightsquigarrow \{ m6 \}, break, receive, reply, error, exit, link, depend \rightsquigarrow \emptyset, \\
\qquad\quad\quad loopMax \rightsquigarrow 0 \rangle, \\
\qquad\quad \langle type \rightsquigarrow task\ reserve, in \rightsquigarrow \{ t5 \}, out \rightsquigarrow \{ t7 \}, send \rightsquigarrow \{ m7 \}, \\
\qquad\quad\quad accept \rightsquigarrow \{ m8 \}, break, receive, reply, error, exit, link, depend \rightsquigarrow \emptyset, \\
\qquad\quad\quad loopMax \rightsquigarrow 0 \rangle, \\
\qquad\quad \langle type \rightsquigarrow end\ i, in \rightsquigarrow \{ t7 \}, loopMax \rightsquigarrow 0, \\
\qquad\quad\quad break, send, receive, accept, reply, out, link, depend, error, exit \rightsquigarrow \emptyset \rangle, \\
\qquad\quad \langle type \rightsquigarrow abort\ j, in \rightsquigarrow \{ t6 \}, loopMax \rightsquigarrow 0, \\
\qquad\quad\quad break, send, receive, accept, reply, out, link, depend, error, exit \rightsquigarrow \emptyset \rangle \ \} \\
\end{array}
$$

The syntactic description may be assumed to have been derived mechanically from the diagram; a similar translation may be applied to the travel agent part of the diagram. We may apply our semantic function to the syntax of the diagram and mechanically obtain a parallel composition of processes, each corresponding to a business participant. We denote the processes corresponding to the traveller and the travel agent participants by the names *Tr* and *Ag* respectively. We define set I to index the processes corresponding to the states of the traveller participant.

$$I = \{ start, order, change, xs, cancel, reserve, end, abort \}$$

We use channels $init.a$ to denote transitions to states of participant a, and $starts.a$ to denote initiation of its tasks or subprocesses. We write $msg.t.x$ to denote communication of message x during task or subprocess t. The process *Tr* mechanically obtained by the translation we have described above is as follows:

$$
\begin{array}{l}
Tr = \mathsf{let}\ X = \Box\, i : (\alpha Y \setminus \{fin.1, abt.1\}) \bullet \\
\qquad\qquad\qquad (i \rightarrow X \ \Box\ fin.1 \rightarrow Skip \ \Box\ abt.1 \rightarrow Stop) \\
\qquad\quad Y = (\ \|\ i : I \bullet \alpha P(i) \circ P(i)) \\
\qquad \mathsf{within}\ (Y \ [\![\, \alpha Y \,]\!]\ X) \setminus \{\!|init.tr|\!\}
\end{array}
$$

where for each i in I, the process $P(i)$ is as defined below. We use p, ranging over $\mathbb{N}$, to denote the number of instances of the task *change*, as specified by the second argument to constructor function *miseq*. We write αQ to denote the set of possible events performed by process Q.

$$
\begin{array}{l}
P(change) = \\
\ \mathsf{let}\ \ A(i) = i > 0\ \& \\
\qquad\quad (init.tr.change \rightarrow Skip \fatsemi \\
\qquad\quad starts.tr.change \rightarrow Skip \fatsemi msg.change!x : \{\, in, last \,\} \rightarrow Skip \fatsemi \\
\qquad\quad msg.change.out \rightarrow Skip \fatsemi \\
\qquad\quad init.tr.xs1 \rightarrow Skip \fatsemi A(i-1)) \ \Box\ init.tr.xs \rightarrow Skip \\
\qquad X(n) = (init.tr.mchange \rightarrow Skip \ \Box\ init.tr.xs1 \rightarrow Skip) \fatsemi \\
\qquad\quad (n > 1\ \& \ (init.tr.change \rightarrow (msg.change.in \rightarrow X(n-1) \\
\qquad\qquad\qquad \Box\ msg.change.last \rightarrow init.tr.xs1 \rightarrow init.tr.xs \rightarrow Skip)) \\
\qquad\quad \Box\ n = 1\ \& \ (init.tr.change \rightarrow msg.change.last \rightarrow \\
\qquad\qquad\qquad init.tr.xs1 \rightarrow init.tr.xs \rightarrow Skip) \\
\qquad\quad \Box\ n = N\ \& \ msg.change.end \rightarrow init.tr.xs \rightarrow Skip) \\
\qquad SynSet = \{\, msg.change.in, msg.change.last, init.tr.change, init.tr.xs1, init.tr.xs \,\} \\
\ \mathsf{within} \\
\ ((A(p) \ [\![\, SynSet \,]\!]\ X(p)) \fatsemi P(change)) \ \Box\ fin.1 \rightarrow Skip
\end{array}
$$

$$
\begin{array}{rcl}
P(start) & = & init.tr.order \rightarrow Skip \fatsemi fin.1 \rightarrow Skip \\
P(order) & = & (init.tr.order \rightarrow Skip \fatsemi starts.tr.order \rightarrow Skip \fatsemi \\
 & & msg.order!x : \{\, in, last \,\} \rightarrow Skip \fatsemi msg.order.out \rightarrow Skip \fatsemi \\
 & & init.tr.mchange \rightarrow Skip \fatsemi P(order)) \ \Box\ fin.1 \rightarrow Skip \\
P(xs) & = & (init.tr.xs \rightarrow Skip \fatsemi \\
 & & (init.tr.cancel \rightarrow Skip \ \Box\ init.tr.reserve \rightarrow Skip) \fatsemi P(xs.3)) \\
 & & \Box\ fin.1 \rightarrow Skip \\
P(cancel) & = & (init.tr.cancel \rightarrow Skip \fatsemi \\
 & & starts.tr.cancel \rightarrow msg.cancel!x : \{\, in, last \,\} \rightarrow Skip \fatsemi \\
 & & msg.cancel.out \rightarrow Skip \fatsemi init.tr.abort \rightarrow Skip \fatsemi \\
 & & P(cancelit)) \ \Box\ fin.1 \rightarrow Skip \\
P(abort) & = & (init.tr.abort \rightarrow Skip \fatsemi abt.tr.1 \rightarrow Stop) \ \Box\ fin.1 \rightarrow Skip \\
P(reserve) & = & (init.tr.reserve \rightarrow Skip \fatsemi starts.tr.reserve \rightarrow Skip \fatsemi \\
 & & msg.reserve!x : \{\, in, last \,\} \rightarrow Skip \fatsemi msg.reserve.out \rightarrow Skip \fatsemi \\
 & & init.tr.end \rightarrow Skip \fatsemi P(reserve)) \ \Box\ fin.1 \rightarrow Skip \\
P(end) & = & init.tr.end \rightarrow Skip \fatsemi fin.1 \rightarrow Skip
\end{array}
$$

The process Ag can similarly be obtained mechanically using the semantic function. Their collaboration hence is the parallel composition of processes Tr and Ag.

$$Collab = (Tr \ [\![\, \alpha Tr \ ||\ \alpha Ag \,]\!]\ Ag) \setminus \{\!|msg|\!\}$$

6.2 Verifying Compatibility of Business Collaboration

The admission of refinement means that a CSP process can be a specification as well as a model of an implementation; hence it is possible to design and compare specifications using BPMN. For example, to check whether both the traveller and the travel agent participants in the reservation system, shown in Figure 5, are compatible, we first construct CSP process *Spec*, a high-level specification corresponding to the traveller participant without message flows, which can also be derived mechanically:

$$
\begin{array}{l}
I' = \{\, st, or, ch, x1, ca, re, en, ab \,\} \\
Spec = \mathsf{let}\ X = \Box\, i : (\alpha Y \setminus \{fin.1, abt.1\}) \bullet \\
\qquad\qquad\qquad (i \rightarrow X \ \Box\ fin.1 \rightarrow Skip \ \Box\ abt.1 \rightarrow Stop) \\
\qquad\qquad Y = (\ \|\ i : I' \bullet \alpha P(i) \circ P(i)) \\
\qquad \mathsf{within}\ (Y \ [\![\, \alpha Y \,]\!]\ X) \setminus \{\!|init.tr|\!\}
\end{array}
$$

For each i in I', the process $P(i)$ is as defined below

$$
\begin{array}{l}
P(ch) = \\
\ \mathsf{let}\ \ A(i) = i > 0\ \&\ (init.tr.change \rightarrow Skip \fatsemi starts.tr.change \rightarrow Skip \fatsemi \\
\qquad\qquad\qquad\qquad init.tr.xs1 \rightarrow Skip \fatsemi A(i-1)) \ \Box\ init.tr.xs \rightarrow Skip \\
\qquad X(n) = (init.tr.mchange \rightarrow Skip \ \Box\ init.tr.xs1 \rightarrow Skip) \fatsemi \\
\qquad\qquad (n > 1\ \&\ (init.tr.change \rightarrow X(n-1)) \\
\qquad\qquad\ \Box\ n = 1\ \&\ (init.tr.change \rightarrow init.tr.xs1 \rightarrow init.tr.xs \rightarrow Skip) \\
\qquad\qquad\ \Box\ init.tr.xs \rightarrow Skip \\
\ \mathsf{within} \\
\qquad ((A(N) \ [\![\, \{\, init.tr.change, init.tr.xs1, init.tr.xs \,\} \,]\!]\ X(N)) \fatsemi \\
\qquad\ P(ch)) \ \Box\ fin.1 \rightarrow Skip
\end{array}
$$

$$
\begin{array}{rcl}
P(st) & = & init.tr.order \rightarrow Skip \fatsemi fin.1 \rightarrow Skip \\
P(or) & = & (init.tr.order \rightarrow Skip \fatsemi \\
 & & starts.tr.order \rightarrow Skip \fatsemi \\
 & & init.tr.mchange \rightarrow Skip \fatsemi P(order)) \ \Box\ fin.1 \rightarrow Skip \\
P(x1) & = & (init.tr.xs \rightarrow Skip \fatsemi \\
 & & (init.tr.cancel \rightarrow Skip \ \Box\ init.tr.reserve \rightarrow Skip) \fatsemi P(xs.3)) \\
 & & \ \Box\ fin.1 \rightarrow Skip \\
P(ca) & = & (init.tr.cancel \rightarrow Skip \fatsemi \\
 & & starts.tr.cancel \rightarrow Skip \fatsemi init.tr.abort \rightarrow Skip \fatsemi \\
 & & P(cancelit)) \ \Box\ fin.1 \rightarrow Skip \\
P(ab) & = & (init.tr.abort \rightarrow Skip \fatsemi abt.tr.1 \rightarrow Stop) \ \Box\ fin.1 \rightarrow Skip \\
P(re) & = & (init.tr.reserve \rightarrow Skip \fatsemi starts.tr.reserve \rightarrow Skip \fatsemi \\
 & & init.tr.end \rightarrow Skip \fatsemi P(reserve)) \ \Box\ fin.1 \rightarrow Skip \\
P(en) & = & init.tr.end \rightarrow Skip \fatsemi fin.1 \rightarrow Skip
\end{array}
$$

We use stable failures refinement to compare *Spec* with *Collab*.

$$Spec \sqsubseteq_{\mathcal{F}} (Collab \setminus (\alpha Collab \setminus \alpha Spec))$$

This expression asserts that the collaboration behaves as specified by the traveller participant; in order for this to happen, both participants must be *compatible* with respect to their collaboration. We have specifically defined our semantics to allow refinement assertions such as this one to be automatically checked by a model checker such as FDR [6]. In this particular example, we find that the refinement assertion above does not hold; this means that the participants in the collaboration described in Figure 5 are *incompatible*. When we ask FDR to verify the assertion above, the following counterexample in the form of a failure is given, where Σ denotes the set of all event names.

$$(\langle starts.tr.order, starts.tr.cancel \rangle, \Sigma)$$

This counterexample tells us that a deadlock has occurred while the traveller is cancelling her itinerary: after the *order* and *cancel* events, the collaboration may refuse to engage in any further activity. A more detailed analysis of the counterexample may be carried out by looking at the failures of processes *Tr* and *Ag* separately:

$$(\langle starts.tr.order, msg.order.in, msg.order.out, msg.change.end, starts.tr.cancel \rangle, ref1)$$
$$(\langle msg.order.in, starts.ag.order, msg.order.out, msg.change.end \rangle, ref2)$$

where $msg.cancel.in \notin ref1$ and $msg.cancel.in \in ref2$. The failures inform us that while the process *Traveller* is willing to only perform the event $msg.cancel.in$, the process *Agent* is not, and this leads to a deadlock. This means that while the traveller may cancel her itinerary before deciding to reserve her ticket, and hence send a message to the travel agent about the cancellation, the travel agent may only carry out her cancellation after entering the reservation phase, and hence may not send a reply message back to the traveller. This discrepancy might have been deliberate due to the internal policies of different business domain, or it might just be a human error. There are two ways to correct this collaboration, either by changing the traveller's or the travel agent's internal process description. We have chosen the latter; a compatible collaboration with a modified travel agent participant is shown in Figure 6. Note the change in the travel agent participant, allowing the task state *Cancel Itinerary* to be triggered before the subprocess state *Receive Reservation*.

By applying our semantic function to the syntax of this diagram we obtain the following parallel composition of processes, each corresponding to a participant.

$$Collab2 = (Tr \,[\![\, \alpha Tr \,||\, \alpha Ag2 \,]\!]\, Ag2) \setminus \{\!|msg|\!\}$$

To check for compatibility, we ask FDR to verify the following refinement assertion. This time, the verification is successful.

$$Spec \sqsubseteq_{\mathcal{F}} (Collab2 \setminus (\alpha Collab2 \setminus \alpha Spec))$$

To generalise the notion of compatibility we first formalise *incompatibility* with respect to a collaboration as follows:

Definition 1 *Given some collaboration* $C = (\, \|\, i : \{\, 1 \,..\, n \,\} \bullet \alpha P(i) \circ P(i)) \setminus M$ *where* n *ranges over* $\mathbb{N}$ *and* M *is the set of events corresponding to the*

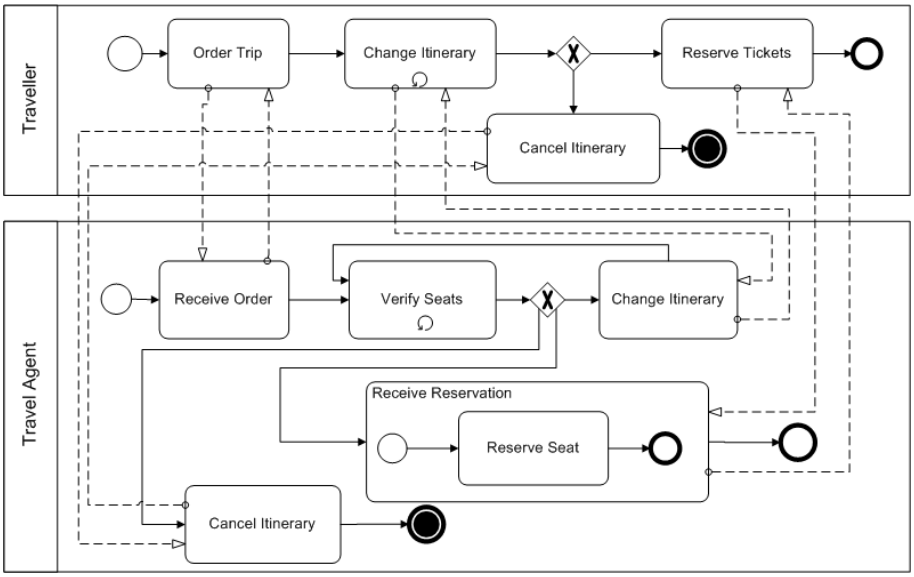

Figure 6: A BPMN diagram describing a compatible business collaboration for airline ticket reservation.

message flows between participants $P(i)$, particpants $P(i)$ are incompatible *with respect to C iff the following statement holds. Note in CSP the event ✓ denotes successful termination.*

$$
\begin{aligned}
(\exists\, i, j : \mathbb{N};\ tr, t, u : \operatorname{seq}\Sigma;\ m, r, s : \mathbb{P}\,\Sigma \bullet \\
\quad (tr, \Sigma) \in \mathit{failures}(C) \land \mathit{last}\ tr \notin (\{\!|\, abt \,|\!\} \cup \{\, \checkmark \,\}) \\
\quad \land \{\, i, j \,\} \subseteq \{\, 1 \mathinner{..} n \,\} \land (t, r) \in \mathit{failures}(P(i)) \land (u, s) \in \mathit{failures}(P(j)) \\
\quad \land\ t \upharpoonright M = tr \upharpoonright \alpha P(i) \land u \upharpoonright M = tr \upharpoonright \alpha P(j) \land m \in (M \cap (r \setminus s \cup s \setminus r)))
\end{aligned}
$$

On the other hand, participants $P(i)$ are compatible *with respect to the collaboration C if they are not incompatible.*

7 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 consistency and compatibility between business processes specified in BPMN. 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.

Our semantic model also allows compatibility of collaboration between multiple business processes to be verified. To the best of our knowledge, no work has been done towards the compatibility verification of business collaborations described in a graphical modelling notation like BPMN; other approaches have mainly focused on the compatibility problem of web services choreographies

described in XML-based languages such as WS-BPEL [1, 7], WSCI [17, 3] and WS-CDL [11, 8]. While the language WSCI [3] has been succeeded by WS-CDL, WS-CDL has been implemented as an interaction model and BPMN models only the interconnection between participants. Furthermore whereas they focus on compatibility at the implementation level, we have moved it forward to the design level, allowing collaboration to be verified and agreed upon before implementation.

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.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services version 1.1.* BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG and Siebel Systems, May 2003. Available at `ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf`.

[2] C. Bolton and J. Davies. Activity graphs and processes. In *IFM '00: Proceedings of the Second International Conference on Integrated Formal Methods*, pages 77–96, 2000.

[3] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing Web Services Choreographies. In *Electronic Notes in Theoretical Computer Science 105*, pages 73–94, 2004.

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

[7] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility verification for web service choreography. In *IEEE International Conference on Web Services*, 2004.

[8] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-Based Analysis of Obligations in Web Service Choreography. In *IEEE International Conference on Internet and Web Applications and Services*, 2006.

[9] H. Foster. Mapping BPEL4WS to FSP. Technical report, Imperial College, London, 2003.

[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] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. *Web Services Choreography Description Language 1.0*, 2005. W3C Candidate Recommendation.

[12] M. Koshkina. Verification of business processes for web services. Master's thesis, York University, Toronto, Oct. 2003.

[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] S. White. Using BPMN to Model a BPEL Process. BPTrends, 2005. Available at `www.bptrends.com`.

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