Verifying Business Process Compatibility

Peter Y. H. Wong, Jeremy Gibbons
Computing Laboratory, University of Oxford, United Kingdom
{peter.wong,jeremy.gibbons}@comlab.ox.ac.uk

Abstract

We describe a process-algebraic approach to verifying process interactions for business collaboration described in Business Process Modelling Notation. We first overview our process semantics for BPMN in the language of Communicating Sequential Processes; we then use a simple example of business collaboration to demonstrate how our semantic model may be used to verify compatibility between business participants in a collaboration.

1. Introduction

This paper describes a process-algebraic approach to verifying process interactions for business collaboration described in Business Process Modelling Notation (BPMN) [2]. BPMN is a graphical modelling notation adopted by the Object Management Group (OMG) [7].

In our previous work [10] we have given a process semantics to a subset of BPMN in the language of CSP [9], of which a prototype has subsequently been implemented in the functional programming language Haskell, and can be found in our web site [1]. In this paper we show how this semantics allows formal reasoning about business-to-business collaboration where there are multiple business processes under consideration. Consider, for instance, the simple example of an airline ticket reservation shown in Figure 1.

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 mutual 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 making her ticket reservation, while from the travel agent participant's perspective such a sequence of tasks might not be allowed. By applying our semantic model, this property can be verified or disproved automatically with a model-checker.

The rest of this paper is structured as follows. Section 2 gives an introduction to BPMN. Section 3 gives an overview of our syntactic description of BPMN and its behavioural semantics. Our semantic construction starts from syntax expressed in Z [11], following Bolton and Davies' work on UML activity graphs [1]. We assume readers have basic knowledge of the mathematical notations Z [11] and CSP [9]. The details of our semantic model may be found in our earlier work [10]; considerations of space preclude anything more than a brief summary here. We revisit the example in Sections 4 and 5 and show how our semantics may be used to verify compatibility between collaboration participants and how such a property can also be specified in BPMN. We conclude with a summary and comparing with related work.

2. BPMN

States in our subset of BPMN [2] 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.

A table showing each type of state is presented in Figure 2. In the figure, a *start* state models the start of the business process in the current scope by initiating its outgoing transition. It has no incoming transition and only one outgoing transition. There are two types of end states, *end* and *abort*. An *end* state models the successful termination of an instance of the business process in the current scope by initialisation of its incoming transition. It has only one incoming transition with no outgoing transition. The *abort* state is a variant end state modelling an unsuccessful termi-

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

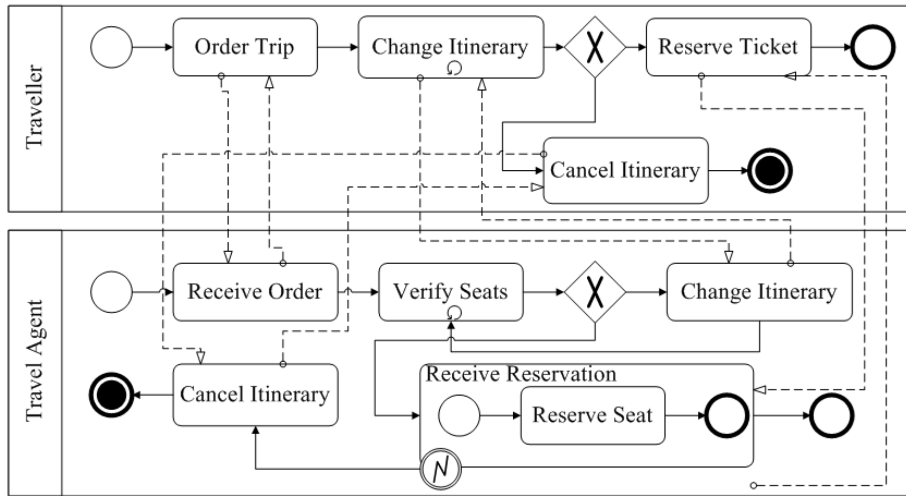

Figure 1. A business collaboration of an airline ticket reservation.

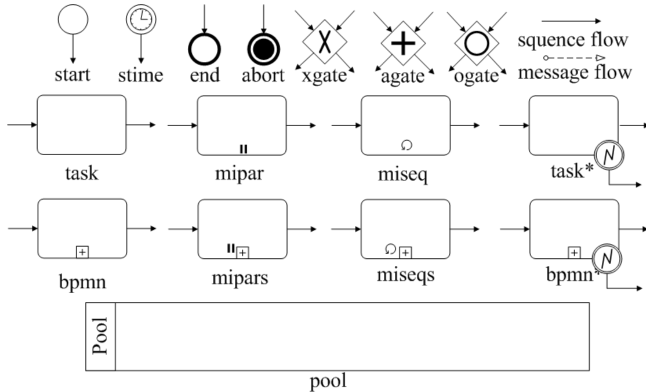

Figure 2. States of BPMN diagram

nation, usually an error of an instance of the business process in the current scope.

Also in the figure, each of the *xgate*, *agate* and *ogate* state types has one or more incoming sequence flows and one or more outgoing sequence flows. An *xgate* state is an exclusive gateway, accepting one of its incoming flows and taking one of its outgoing flows; the semantics of this gateway type can be described as an exclusive choice and a simple merge. An *agate* state is a parallel gateway, which waits for all of its incoming flows before initialising all of its outgoing flows. An *ogate* state is an inclusive gateway, accepting one or more incoming sequence flows depending on their associated guards and initialising one or more of its outgoing flows also depending on their associated guards.

A *task* state describes an atomic activity with exactly one incoming and one outgoing transitions A *bpmn* state describes a subprocess state; it is a business process by itself and so it models a flow of BPMN states. The *bpmn* state shown in Figure 2 is *collapsed* subprocess state where all internal details are hidden. This state has exactly one incoming and one outgoing transition. Also in Figure 2 there are graphical notations labelled *task** 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 instance in BPMN: The *miseq* state type represents sequential multiple instance, where the specified task is repeated sequentially; in the *mipar* state type the specified task is repeated in parallel. The types *miseqs* and *mipars* are their subprocess counterparts.

The graphical notation *pool* in Figure 2 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.

3. Syntax and Semantics of BPMN

In this section we summarise our previous syntactic description and semantic model of BPMN [10]. For reasons of space, when referring to a function defined in this previous work, we provide only a type declaration and a brief description. Given the basic types *CName*, *PName*, *Task*, *Line*, *Channel*, *Guard* and *Message*:

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

where the following defines subtypes *BName* and *PLName* for subprocess names and BPMN diagram names, and message types *InMsg*, *OutMsg*, *EndMsg* and *LastMsg* axiomatically,

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

for each type of state shown in Figure 2 we provide the corresponding syntax in Z [11], here we show only some of that definition

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

The argument of the constructor functions *end* and *abort* represents a unique identifier for each state of type *end* or *abort*; the second argument of the constructor functions *mipar*, *mipars*, *miseq* and *miseqs* represent the maximum number of instances of a task or a subprocess the multiple instance states can trigger.

We define the type of a sequence flow or an exception flow by the schema *Tns* and the type of a message flow by the schema *Msgflow*:

$$\begin{aligned} & Tns \mathrel{\widehat=} [guard : Guard;\ line : Line] \\ & Msgflow \mathrel{\widehat=} [message : Message;\ channel : Channel] \end{aligned}$$

Each state in a BPMN process has a set of incoming transitions; depending on the state's type, one or more of them might be needed to trigger the state's execution. Each state also has a set of outgoing transitions; depending on the state's type, one or more of them might be triggered after the state's execution. For example, an *agate* state waits for all of its incoming transitions before initialising all of its outgoing transitions. Each task and subprocess state might also have a set of message flows for receiving messages and a set for sending messages across participant pools within a BPMN process. We define states by the schema *State*.

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

Each *State* records the type of its content, the sets of incoming, outgoing and error transitions of schema type *Tns*, and, in the case of a subprocess state, a set of number-transition pairs to align the outgoing transitions of the subprocess with the exit states in the subprocess. There are also the five sets of message flows.

We define $WCF : \mathbb{P}(\mathbb{P}\, State)$ to be the set of *well-configured* sets of well-formed states. The type *State* allows all possible states, including those which are not permissible within a BPMN diagram (for example, a *start* state with a non-empty set of incoming transitions, or a task state with message flows that allows it to send and receive messages to and from another state within the same participant pool); the full definition can be found in our earlier paper [10]. Each BPMN diagram encapsulated by a *pool* represents an individual business 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 the boundary of a business domain. The function *Local* represents the environment of the local specification, and maps each BPMN diagram name of type *PLName*, a subtype of *PName*, to its associated diagram. A business collaboration is built up from a finite set of names, each associated with its BPMN diagram; the function *Global* represents the environment of a global specification and maps each collaboration name of type *CNAME* to its associated diagram.

$$\begin{aligned} & BPD ::= states\langle\!\langle WCF \rangle\!\rangle \\ & Local == PName \rightarrow\!\!\!\!\!+ BPD \end{aligned}$$

We define the semantic function *csem* which takes a syntactic description of a BPMN collaboration diagram and returns the CSP process of type *Process* that models the behaviour of that diagram. That is, the function takes one or more *pool* states, each encapsulating a separate BPMN diagram representing an individual participant within a business collaboration identified by its collaboration name of type *CName*, and returns a parallel composition of processes, each corresponding to an individual participant.

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

The function α_{msg} maps each set of message flows to the set of associated events. The function $\alpha_{process}$ maps each BPMN diagram representing a local business process to its alphabet of type $\mathbb{P}\, Event$, where *Event* is the basic type for CSP events.

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

The alphabet of a local business process is the union of the alphabets of each of its states, while the function α_{state} maps each state to its alphabet. The alphabet of each state is the

set of events associated with a state with which it must synchronise, hence a state's alphabet is the union of the events mapped from all incoming, outgoing and error transitions, type and message flows.

$$\alpha_{state} : State \rightarrow\!\!\!\!\!+ \; Local \rightarrow\!\!\!\!\!+ \; \mathbb{P}\, Event$$

The function *bsem* takes a syntactic description of a BPMN diagram encapsulated by a state of type *pool* or a BPMN subprocess, identified by the diagram's name of type *PName*, and returns a parallel composition of processes, each corresponding to one of the diagram's or process's states.

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

The parallel composition of processes, defined by the function *bsm*, is conjoined via partial interleaving with process M defined above 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, respectively. The functions ϵ_{end} and ϵ_{abort} return the set of numbers defined by each of the *end* states and the *abort* states within the diagram's syntax respectively, while the function α_{trans} maps each set of transitions to the set of associated events. The semantics of a BPMN state is the composition of the processes, each corresponding to the state's incoming and outgoing transitions, type, exception and message flows. Note we allow explicit ordering of message flows, for example, in Figure 1, the order of message flow is either left to right or up to down. Also we allow external choices over the semantics of exclusive gateways, this makes the modelling of process interaction possible. Readers may refer to our earlier paper [10] for the complete semantic definition and its more detailed explanation.

4. Revisiting the Example

In this section we revisit the example shown in Figure 1. Given the business process name *traveller*, Given we have the syntactic definition of the business collaboration, as described in the previous section, we may apply our semantic function to it 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:

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

where for each i in I, the process $P(i)$ is as defined below. We write αQ to denote the set of possible events performed by process Q. Here we omit the semantic definition of task *change* for reasons of space.

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

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

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

5. Verifying Compatibility

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. To check whether both the traveller and the travel agent participants are compatible, we first construct CSP process *Spec*, corresponding to the traveller participant without message flows, which can also be derived mechanically:

$$I' = \{ st, or, ch, x1, ca, re, en, ab \}$$

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

Here for reasons of space, we show below a subset of processes $P(i)$ where i ranges over I'.

$$\begin{aligned} P(st) &= init.tr.order \rightarrow Skip \,\text{;}\, fin.1 \rightarrow Skip \\ P(or) &= (init.tr.order \rightarrow Skip \,\text{;}\, starts.tr.order \rightarrow Skip \,\text{;} \\ &\quad init.tr.mchange \rightarrow Skip \,\text{;}\, P(order)) \\ &\quad \Box\, fin.1 \rightarrow Skip \\ P(x1) &= (init.tr.xs \rightarrow Skip \,\text{;}\, (init.tr.cancel \rightarrow Skip \\ &\quad \Box\, init.tr.reserve \rightarrow Skip) \,\text{;} \\ &\quad P(xs.3)) \Box fin.1 \rightarrow Skip \\ P(ab) &= (init.tr.abort \rightarrow Skip \,\text{;}\, abt.tr.1 \rightarrow Stop) \\ &\quad \Box\, fin.1 \rightarrow Skip \\ P(en) &= init.tr.end \rightarrow Skip \,\text{;}\, fin.1 \rightarrow Skip \end{aligned}$$

We use CSP's stable failures refinement [9] to compare the process *Spec* with the process *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 [5]; we have carried out such experiments. 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 1 are *incompatible*. When we ran FDR on the assertion above; the following counterexample in the form of a failure was 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:

$$\begin{aligned} &(\langle starts.tr.order, msg.order.in, msg.order.out, \\ &\quad msg.change.end, starts.tr.cancel \rangle, ref1) \\ &(\langle msg.order.in, starts.ag.order, msg.order.out, \\ &\quad msg.change.end \rangle, ref2) \end{aligned}$$

where $msg.cancel.in \notin ref1$ and $msg.cancel.in \in ref2$. The failures inform us that while the process *Traveller* is ready to 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; Figure 3 shows a compatible travel agent participant for the collaboration of an airline ticket reservation. 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.

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

Informally, participants are incompatible with respect to a collaboration if the collaboration deadlocks while individually its participants are deadlock free. Specifically we require each participants' behaviour to represent a *responsive plug-in* [8] to all other participants. Informally process Q is a responsive plug-in to P, denoted as Q *RespondsTo* P, if Q is prepared to cooperate with the pattern set out by P for their shared interface. We now generalise *compatibility* using CSP's responsiveness.

Definition 1 ***Compatibility.*** *Given some collaboration described by the CSP process,* $C = (\, \| \, i : \{1 \, .. \, n\} \bullet$

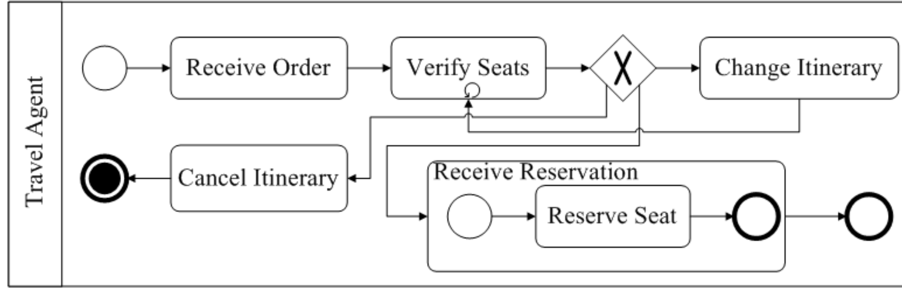

Figure 3. A BPMN diagram describing a compatible travel agent for the collaboration of an airline ticket reservation.

*$\alpha P_i \circ P_i) \setminus M$ where n ranges over $\mathbb{N}$ and M is the set of events corresponding to the message flows between its participants, whose **behaviour** are modelled by the processes P_i, participant P_i is compatible with respect to the collaboration C iff $\forall j : \{ 1 \mathinner{\ldotp\ldotp} n \} \setminus \{ i \} \bullet P_i$ RespondsTo P_j*

6. Related Work and Conclusion

In this paper we described briefly our earlier process semantics for BPMN in the language of CSP to model message flow and reason about business collaborations described in BPMN. We have illustrated by an example how this semantic model may be used to verify compatibility between participants within a business collaboration. We have subsequently implemented a prototype of the semantic function in Haskell. While our earlier work [10] described a process semantics and its application via process refinement in verifying consistency between BPMN diagrams each with a different level of abstraction, our modelling approach in compatibility verification described in this paper utilises this semantics to verify collaboration compatibility between BPMN diagrams at the same level of abstraction participating in a business collaboration.

To the best of our knowledge, the only previous attempt at implementing a formal semantics for a subset of BPMN and using it for the compatibility verification of business collaborations [4] did so using Petri nets. However, that semantics does not properly model multiple instances, exception handling and message flows. Some other approaches in the areas of business process management and services oriented computing have focused on the compatibility problem of web services choreographies described in XML-based languages such as WSCI [3] and WS-CDL [6]. While some have applied existing process calculi to model choreographies, which are equipped with model checking facilities, their models do not induce a refinement ordering between choreographies and hence specifications are often given in separate languages such as the Hennessy-Milner logic or Message Sequence Charts. Moreover their compatibility verifications focus on the implementation level and require non-standard diagram notation to construct choreographies; we, on the other hand, have moved it forward to the design level, and provide verification upon an agreed standard diagram notation. This allows collaboration to be verified and agreed upon before implementation.

This work is supported by a grant from Microsoft Research. The authors would like to thank referees for useful suggestions and comments.

References

[1] C. Bolton and J. Davies. Activity Graphs and Processes. In *IFM '00*, volume 1945 of *LNCS*, pages 77–96, 2000.
[2] *Business Process Modeling Notation (BPMN) Specification*, Feb. 2006. www.bpmn.org.
[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] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement, FDR2 User Manual*, 1998. www.fsel.com.
[6] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-Based Analysis of Obligations in Web Service Choreography. In *AICT-ICIW'06*, 2006.
[7] Object Management Group. http://www.omg.org.
[8] J. N. Reed, J. E. Sinclair, and A. W. Roscoe. Responsiveness of interoperating components. *Form. Asp. Comput.*, 16(4):394–411, 2004.
[9] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
[10] P. Y. H. Wong and J. Gibbons. A Process Semantics for BPMN, 2007. Submitted for publication. Extended version available at http://www.comlab.ox.ac.uk/peter.wong/pub/bpmnsem.pdf.
[11] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.