# Compositional Development of BPMN

Peter H. Wong

# Compositional Development of BPMN *

Peter Y. H. Wong

SDL Fredhopper, Amsterdam, The Netherlands, peter.wong@fredhopper.com

**Abstract.** Business Process Modelling Notation (BPMN) intends to bridge the gap between business process design and implementation. Previously we provided a process semantics to a subset of BPMN in the language of Communicating Sequential Processes (CSP). This semantics allows developers to formally analyse and compare BPMN diagrams using CSP's traces and failures refinements. In this paper we introduce a comprehensive set of operations for constructing BPMN diagrams, provide them a CSP semantics, and characterise the conditions under which the operations are monotonic with respect to CSP refinements, thereby allowing compositional development of business processes.

## 1 Introduction

Modelling of business processes and workflows is an important area in software engineering. Business Process Modelling Notation (BPMN) [8] allows developers to take a process-oriented approach to modelling of systems. There are currently over seventy implementations of the notation, but the notation specification does not have a formal behavioural semantics, which we believe to be crucial in behavioural specification and verification activities. Previously a process semantics [12] has been given for a large subset of BPMN in the language of Communicating Sequential Processes (CSP) [10]. This semantics maps BPMN diagrams to finite state CSP processes. Using this semantics, the behaviour expressed in BPMN can be formally verified and BPMN diagrams can be behaviourally compared. Verification and comparison are expressed as CSP traces and failures refinements. Refinements of finite state CSP processes can be automatically verified using a model checker like FDR [4].

### 1.1 Monotonicity and Refinement

One major problem of verifying concurrent systems by model checking is potential state explosion. To alleviate this one can exploit compositionality. Our constribution is to introduce a comprehensive set of operations to incrementally construct BPMN diagrams. We provide these operations with a CSP semantics, and characterise the conditions under which the operations are monotonic with respect to CSP refinements. These operations are partitioned into the following

* Partly funded by Microsoft Research.

categories: sequential composition, split, join, iteration, exception and collaboration.

The combination of monotonicity and refinement allows one to construct and verify behavioural correctness of large complex systems compositionally. Informally, for any two BPMN diagrams $C$ and $D$, $D \sqsubseteq C$ denotes $C$ is a refinement of $D$. We let $f(.)$ be an operation over BPMN diagrams. If $f(.)$ is monotonic with respect to the refinement, then by construction $f(D) \sqsubseteq f(C) \Leftrightarrow D \sqsubseteq C$ for all BPMN diagrams $C$ and $D$. As a result we can incrementally increase the complexity of $C$ and $D$ using the proposed operations without needing further verification.

Moreover, monotonic operations preserve refinement-closed properties. A relation $\oplus$ is refinement-closed if and only if for all BPMN diagrams $P$ and $Q$ such that if $P \oplus Q$, then it is the case that $P' \oplus Q'$ for all $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$. One important refinement-closed property is the behavioural compatibility, *compatible*, between BPMN pools [11, 13]. Given two BPMN pools $P$ and $Q$, $\mathit{compatible}(P, Q)$ if and only if $P$ and $Q$ are deadlock free and that their collaboration is deadlock free. If we let $f(.)$ be a monotonic operation over BPMN diagrams and $\mathit{compatible}(f(C), f(D))$, then by construction $\mathit{compatible}(f(C'), f(D'))$ for all $C'$ and $D'$ such that $C \sqsubseteq C'$ and $D \sqsubseteq D'$. As a result we may use the proposed operations to perform independent development while maintaining *any* refinement-closed properties.

### 1.2 Structure

This paper begins with a brief introduction to Z and CSP in Section 2, and then an overview of the abstract syntax of BPMN in Z and its behavioural semantics in Section 3. Our contribution starts in Section 4: in this section we introduce a set of operations to construct BPMN diagrams and provide them with a CSP semantics. Using this semantics, we characterise the conditions under which these operations are monotonic with respect to CSP refinements. We conclude this paper with a discussion on related work and a summary.

## 2 Preliminaries

**Z** The Z notation [14] is a language for state-based specification. It is based on typed set theory with a structuring mechanism: the schema. We write some schema $N$ to make declaration $d$ that satisfies constraint $p$ as $N \mathrel{\widehat=} [d \mid p]$. 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 $[\mathit{Type}]$, or be free types, introduced by identifying each of the distinct members, introducing each element by name $\mathit{Type} ::= E_0 \mid ... \mid E_n$. By using an axiomatic definition we can introduce a new symbol $x$, an element of $S$, satisfying constraint $p$.

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

**CSP** In CSP [10], a process is a pattern of behaviour, denoted by events. Below is the syntax of the subset of CSP used in this paper.

$$P, Q ::= P \mid\mid\mid Q \mid P \,[\![ A ]\!]\, Q \mid P \,[\![ A \mid B ]\!]\, Q \mid P \Box Q \mid P \sqcap Q \mid P \fatsemi Q \mid e \rightarrow P \mid \mathit{Skip} \mid \mathit{Stop}$$

Given processes $P$ and $Q$, $P \mid\mid\mid Q$ denotes the interleaving of $P$ and $Q$; $P \,[\![ A ]\!]\, Q$ denotes the partial interleaving of $P$ and $Q$ synchronising events in set $A$; $P \,[\![ A \mid B ]\!]\, Q$ denotes parallel composition, in which $P$ and $Q$ can evolve independently performing events from $A$ and $B$ respectively but synchronise on events in their intersection $A \cap B$; we write $\Big\|\, i : I \bullet A(i) \circ P(i)$ to denote an indexed parallel combination of processes $P(i)$ for $i$ ranging over $I$. $P \Box Q$ denotes the external choice between $P$ and $Q$; $P \sqcap Q$ denotes the internal choice between $P$ or $Q$; $P \fatsemi Q$ denotes the sequential composition of $P$ and $Q$; $e \rightarrow P$ denotes a process that is capable of performing event $e$, and then behaves as $P$. The process *Stop* is a deadlocked process and the process *Skip* is a successful termination. One of CSP's semantic models is the stable failures ($\mathcal{F}$) The stable failures model records the failures of processes. We write $\mathit{traces}(P)$ and $\mathit{failures}(P)$ to denote the traces and the failures of process $P$. A failure is a pair $(s, X) \in \mathit{failures}(P)$ where $s \in \mathit{traces}(P)$ is the trace of $P$ and $X$ is the set of events of $P$ refuses to do after $s$. CSP semantics admit refinement orderings such as the stable failures refinement $P \sqsubseteq_{\mathcal{F}} Q \Leftrightarrow \mathit{traces}(P) \supseteq \mathit{traces}(Q) \wedge \mathit{failures}(P) \supseteq \mathit{failures}(Q)$. Refinement under the stable failures model allows assertions about a system's *safety* and *availability* properties.

## 3 Formalising BPMN

A BPMN diagram is made up of a collection of BPMN elements. Elements can either be events, tasks, subprocesses or control gateways. Elements are grouped in pools. A pool represents a participant in a business process; a participant may be an entity such as a company. Elements in a pool are connected by sequence flows to depict their flow of control in the pool, A BPMN diagram is a collection of pools in which elements between pools may be connected by message flows to depict the flow of messages between pools.

As a running example we consider the BPMN diagram shown in Fig. 1. The diagram describes the business process of an online shop promoting a sale. Specifically, it is a business collaboration between an online shop and a customer, depicted by two pools. The online shop business process begins by sending a message to the customer about a sale offer. This is modelled as a message flow from the task named Send Offer to the message event. The business process then waits until it receives either a confirmation or a decline from the customer. This decision waiting is modelled using the event-based exclusive-or gateway from the Send Offer task to the tasks Receive Confirmation and Receive Decline. If a decline is received, the online shop business process ends. If a confirmation is received, the online shop receives payment from the customer, sends the invoice and dispatches the goods to her. The customer's business process begins by receiving a message from the online shop about a certain promotion item. She

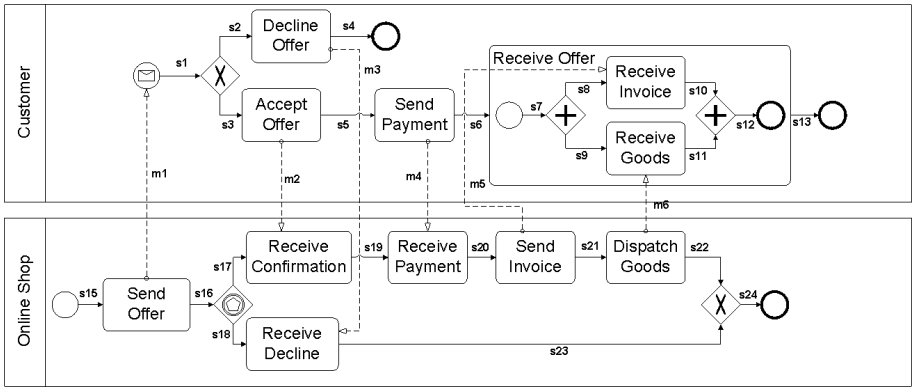


**Fig. 1.** A running example of a BPMN diagram

may either accept or decline the offer. The decision is modelled using the data-based exclusive-or gateway from the message event to the tasks Decline Offer and Accept Offer. If she decides to accept the offer, she sends payment to the shop, modelled by task Send Payment, then waits for her offer. This is modelled as a subprocess Receive Offer consisting of two tasks: The task Receive Goods models the receiving of goods, while Receive Invoice models the receiving of the invoice. Note these activities may happen in either order and this is modelled using a parallel gateway to both tasks. If she declines the offer, the business process ends.

We now give an overview of BPMN's abstract syntax in Z and its semantics in CSP. For brevity, we highlight only the most important aspects of the syntax and semantics. The complete formalisation of the syntax and semantics can be found in the author's thesis [11, Chapters 4, 5].

In the abstract syntax we let basic types *Sflow* and *Mflow* be types of sequence flows and message flows, and *TaskName* be the type of task names; we use the free type $Type ::= start \mid end \mid ...$ to record the type of an element. The free type $Element ::= atom\langle\langle Atom\rangle\rangle \mid comp\langle\langle Atom \times \mathbb{F}_1\, Element\rangle\rangle$ then records individual BPMN elements, where schema $Atom \mathrel{\widehat=} [t : Type;\ in, ou : \mathbb{F}\, Sflow;\ sn, re : \mathbb{F}\, Mflow \mid in \cap ou = \emptyset \land sn \cap re = \emptyset]$ records the type, the sequence flows and the message flows of an element.

We define schema $Pool \mathrel{\widehat=} [proc : \mathbb{F}_1\, Element]$ to record individual pools, it defines component *proc* to record a non-empty finite set of BPMN elements contained in a pool; note that for brevity we have omitted the specification of predicate constraints on that component. We define $InitPool \mathrel{\widehat=} [proc' : \mathbb{F}_1\, Element \mid proc' = \{se, ee\}]$ to be the initial state of *Pool*, where *se* is a start event and *ee* is an end event such that $(atom^{\sim} se).ou = (atom^{\sim} ee).in$. We also specify the structure of a BPMN diagram using the schema $Diagram \mathrel{\widehat=} [pool : PoolId \rightarrowtail Pool \mid pool \neq \emptyset]$. It states that a diagram consists of one or more BPMN pools, with each pool being uniquely identified by some *PoolId* value $i$ such that $pool(i)$ gives that pool (where *PoolId* is a basic type). Similar to *Pool*, we define $InitDiagram \mathrel{\widehat=} [pool' : PoolId \rightarrowtail Pool \mid pool' = \{p1 \mapsto \langle proc \rightsquigarrow \{se, ee\}\rangle\}]$ to be the initial state of *Diagram*, where $p1$ is a *PoolId*.

We define the semantic function $bToc$ that takes a BPMN diagram and returns a CSP process that models communications between elements in that diagram. We present the semantic definition in a bottom up manner, starting with individual BPMN elements.

We first define function $aToc$ to model atomic elements, that is, events, tasks and gateways. The function defines the behaviour of the elements by sequentially composing the processes that model the element's incoming sequence flows and message flows, the element's type and the element's outgoing sequence flows and message flows. Note that other than start and end events, $aToc$ models other atomic elements as recursive processes.

$$aToc : Element \pfun CSP$$

Our semantics models the communications between elements as a parallel combination of processes, each modelling the behaviour of an element. Our semantics ensures that upon an end event being triggered, the containing BPMN process may terminate if its contained elements can also terminate. This is achieved by insisting that each end event performs a completion event, and that all other elements may synchronise on one of these completion events to terminate.

$$cp : (\mathbb{F}\, Element \times Element) \to CSP$$

The function $cp$ defines the execution of a BPMN element as follows: for an atomic element, $cp$ applies function $aToc$, and for a compound element, $cp$ applies function $mToc$; function $mToc$ defines the sequential composition of these processes: the CSP process that models the incoming sequence flows of the compound element; the parallel composition of processes, each modelling an element contained in the compound element, and the CSP process that models the outgoing sequence flows of the element.

$$mToc : CSP \times \mathbb{F}\, Element \times CSP \to CSP$$

A BPMN diagram is modelled by semantic function $bToc$, which defines a parallel composition of processes, each modelling a BPMN pool in the diagram. A pool is modelled by function $pToc$, which defines a parallel composition of processes, each modelling elements in the pool.

$$pToc : Pool \to CSP$$
$$bToc : Diagram \to CSP$$

This semantic function induces an equivalence relationship on the behaviour of BPMN diagrams. Two BPMN diagrams are equivalent when each failures-refines the other. The notion of equivalence is formally defined as follows:

**Definition 1.** ***Equivalence.*** *Two BPMN diagrams $P$ and $Q$ are equivalent, denoted as $P \equiv_{BPMN} Q$ if and only if $bToc(Q) \sqsubseteq_{\mathcal{F}} bToc(P) \land bToc(P) \sqsubseteq_{\mathcal{F}} bToc(Q)$.*

For example, we consider our online shop running example shown in Fig. 1. We define CSP process $CP$ in Equation 1 to model the behaviour of the *Customer*

BPMN pool, where $\alpha P$ denotes the alphabet of process $P$. The start event is identified by $sCP$ while all other BPMN elements in the pool are identified by their incoming sequence flows.

$$
\begin{aligned}
EP &= c.s13 \rightarrow \mathit{Skip} \,\Box\, c.s4 \rightarrow \mathit{Skip} \\
P(sCP) &= (m.m1 \rightarrow s.s1 \rightarrow EP) \,\Box\, EP \\
P(s1) &= (s.s1 \rightarrow (s.s2 \rightarrow \mathit{Skip} \sqcap s.s3 \rightarrow \mathit{Skip}) \fatsemi P(s1)) \,\Box\, EP \\
P(s2) &= (s.s2 \rightarrow w.DO \rightarrow m.m3 \rightarrow s.s4 \rightarrow P(s2)) \,\Box\, EP \\
P(s3) &= (s.s3 \rightarrow w.AO \rightarrow m.m2 \rightarrow s.s5 \rightarrow P(s3)) \,\Box\, EP \\
P(s4) &= (s.s4 \rightarrow c.s4 \rightarrow \mathit{Skip}) \,\Box\, c.s13 \rightarrow \mathit{Skip} \\
P(s5) &= (s.s5 \rightarrow w.SP \rightarrow m.m4 \rightarrow s.s6 \rightarrow P(s4)) \,\Box\, EP \\
P(s6) &= (s.s6 \rightarrow RO \fatsemi s.s13 \rightarrow P(s6)) \,\Box\, EP \\
P(s13) &= (s.s13 \rightarrow c.13 \rightarrow \mathit{Skip}) \,\Box\, c.s4 \rightarrow \mathit{Skip} \\
CP &= \Big\| i : \{sCP, s1, s2, s3, s4, s5, s6, s13\} \bullet \alpha P(i) \circ P(i)
\end{aligned} \tag{1}
$$

For presentation purpose, we model sequence flows by prefixing '$s.$', message flows by '$m.$' and completion events by '$c.$'. We abbreviate each task name using the first letter of each word in its name. For example, the CSP event $w.DO$ represents the work done of task Decline Offer, the process $P(s6)$ models the behaviour of the Receive Offer subprocess; the definition of $RO$ is defined in Equation 2.

$$
\begin{aligned}
FP &= c.s12 \rightarrow \mathit{Skip} \\
P(sRO) &= (s.s7 \rightarrow FP) \,\Box\, FP \\
P(s7) &= (s.s7 \rightarrow (s.s8 \rightarrow \mathit{Skip} \,|||\, s.s9 \rightarrow \mathit{Skip}) \fatsemi P(s7)) \,\Box\, FP \\
P(s8) &= ((s.s8 \rightarrow \mathit{Skip} \,|||\, m.m5 \rightarrow \mathit{Skip}) \fatsemi w.RI \rightarrow s.s4 \rightarrow P(s8)) \,\Box\, FP \\
P(s9) &= ((s.s9 \rightarrow \mathit{Skip} \,|||\, m.m6 \rightarrow \mathit{Skip}) \fatsemi w.RG \rightarrow s.s11 \rightarrow P(s9)) \,\Box\, FP \\
P(s10) &= ((s.s10 \rightarrow \mathit{Skip} \,|||\, s.s11 \rightarrow \mathit{Skip}) \fatsemi s.s12 \rightarrow P(s10)) \,\Box\, FP \\
P(s12) &= s.s12 \rightarrow c.12 \rightarrow \mathit{Skip} \\
RO &= \Big\| i : \{sRO, s7, s8, s9, s10, 12\} \bullet \alpha P(i) \circ P(i)
\end{aligned} \tag{2}
$$

We may similarly define CSP process $OS$ to model the behaviour of the *OnlineShop* BPMN pool. The CSP process $CP \,[\![\, \alpha CP \mid \alpha OS \,]\!]\, OS$ then models the interaction between the customer and the online shop business processes.

## 4 Constructing BPMN

Using Z we provide a comprehensive set of *operations* for constructing BPMN processes. Specifically these are operation schemas on the state schemas *Pool* and *Diagram*. These operations are partitioned into the following categories: sequential composition, split, join, iteration, exception and collaboration. Informally, sequential composition adds to a BPMN process an activity (a task or a subprocess); split adds a choice to two or more activities; join joins two or more

activities via a join gateway; iteration adds a loop using a exclusive-or split and join gateway; exception adds exception flows to a activity, and collaboration connects two BPMN pools with a message flow.

While these operations are not part of BPMN, we did not need to extend the existing syntax of BPMN for defining these operations. These operations are designed to provide the following benefits:

1. To provide operations to construct business processes. We have chosen a comprehensive set of operations for constructing business processes similar to those in structured programming [1];
2. To ensure the syntactic consistency of business processes by calculating the preconditions of the operations. Precondition calculations can be found in the author's thesis [11, Appendix B];
3. To allow the compositional development of business processes. We provide a CSP semantics to these operations, and characterise the conditions under which these operations are monotonic with respect to CSP traces and failures refinements, and
4. To encourage formal tool support for constructing business processes. The behavioural of BPMN diagrams constructed using the provided operations admits verification such as automatic refinement checking. These operations can be implemented in a BPMN development environment that supports the semantic translation of BPMN to CSP[1] and the automated refinement checking via model checkers such as the FDR.

### 4.1 Syntax and Semantics

We describe four operations using the combination of Z and CSP; when describing operations, we refer to Fig. 2 for illustration purposes. Diagrams labelled with a number depict an operation's before state and diagrams with a letter depict an operation's after state. Operations whose after states are one of Digrams A and B assume Diagram 1 to be the before state; the operation whose after state is Diagram C assumes Diagram 2 to be the before state, and the operation whose after state is Diagram E assumes Diagram 3 to be the before state. For reason of space, we mainly focus on semantic definitions of these operations and present the syntactic definition of sequential composition (*SeqComp*), while for the other operations, we present their type declaration. We also provide an informal description of these operations.

We provide functions *pf*, *ext*, *int* and *par* to model prefixing, external choice, internal choice and interleaving. We provide functions *type*, *in*, *ou*, *re* and *sd* on BPMN elements to obtain their type, incoming and outgoing sequence flows, and incoming and outgoing message flows respectively. We provide functions *sf*, *mf* and *fn* to model sequence flow, message flow and completion as CSP events. Furthermore, we provide functions *sflow*, *cnt*, *ends*, *eles* and *modify* as follow:

[1] A prototypical implementation of the semantic translation can be found at `http://sites.google.com/site/peteryhwong/bpmn`

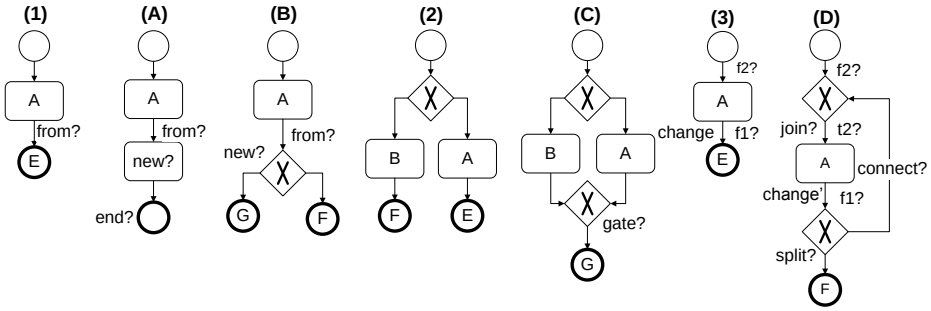


**Fig. 2.** Operations

*sflow*($es$) takes a set of elements $es$ and returns all sequence flows of the elements in $es$; *cnt*($e$) takes element $e$ and recursively returns the set of elements contained in element $e$; *ends*($ps$) takes a set of elements $ps$ and returns a partial function that maps each incoming sequence flow of an end event to that event; *nonsend*($ps$) takes a set of elements $ps$ and returns a partial function that maps each incoming sequence flow of an element that is not a task or an intermediate event to that element, and *modify*($ps, ns, os$) takes three sets of elements $ps$, $ns$ and $os$ and returns $ps$ with elements of $os$ contained in $ps$ replaced with elements in $ns$.

*SeqComp* The schema *SeqComp* adds either a task, a subprocess, or an intermediate message event to a BPMN pool. It takes two elements *new*? and *end*?, and sequence flow *from*? as inputs and replaces the end event in the component *proc* that has incoming sequence flow *from*? with elements *new*? and *end*?.

$$
\begin{aligned}
&\mathit{SeqComp} \mathrel{\widehat=} [\Delta \mathit{Pool};\ \mathit{Commons};\ \mathit{end}? : \mathit{Element} \mid \\
&\quad \#(\mathit{in}\ \mathit{new}?) = 1 \land \#(\mathit{ou}\ \mathit{new}?) = 1 \land \mathit{in}(\mathit{end}?) = \mathit{ou}(\mathit{new}?) \land \\
&\quad \mathit{type}(\mathit{end}?) \in \{\mathit{end}\} \cup \operatorname{ran} \mathit{emsg} \land \#\mathit{in}(\mathit{end}?) = 1 \land \#\mathit{ou}(\mathit{end}?) = 0 \land \\
&\quad \mathit{proc}' = \mathit{modify}(\mathit{proc}, \{\mathit{new}?, \mathit{end}?\}, \{\mathit{ends}(\mathit{proc})\,\mathit{from}?\})]
\end{aligned}
$$

Here *SeqComp* also declares the following schema *Commons*:

$$
\begin{aligned}
&\mathit{Commons} \mathrel{\widehat=} [\mathit{Pool};\ \mathit{new}? : \mathit{Element};\ \mathit{from}? : \mathit{Seqflow} \mid \\
&\quad (\mathit{ou}(\mathit{new}?) \cup \mathit{sflow}(\mathit{cnt}(\mathit{new}?))) \cap \bigcup\{e : \mathit{proc} \bullet \mathit{sflow}(\mathit{cnt}(e))\} = \emptyset \land \\
&\quad \mathit{from}? \in \mathit{in}(\mathit{new}?) \land \mathit{in}(\mathit{new}?) \subseteq \operatorname{dom}(\mathit{ends}\ \mathit{proc})]
\end{aligned}
$$

Schema *SeqComp* is illustrated by Diagrams 1 and A in Fig. 2, where the end event labelled $E$ is specified by the expression ((*ends proc*) *from*?). The illustration shows how this operation replaces element $E$ with element *new*? and *end*?. Specifically, *new*? is either an intermediate message event (with no message flow) or an activity. That is, *new*? has exactly one incoming and one outgoing sequence flow, and *end*? is an end event. Furthermore, *SeqComp* includes *Commons* to ensure the following constraints: 1) no outgoing sequence flow of *news* as well as of elements contained in *new*? must also be a sequence flow of any element contained the before state component *proc*; 2) *from*? is an incoming sequence flow of *new*?, and 3) *from*? is also an incoming sequence flow of an end event contained in *proc*.

We let $e$ denote the end event $end(proc)\,from?$. The semantics of $new?$ and $end?$ are provided by the processes $P(new?) = cp((proc \setminus \{e\}) \cup \{end?\}, new?)$ and $P(end?) = cp(proc \setminus \{e\}, end?)$ respectively. In general, unless otherwise specified, given a state $Pool$ with component $proc$, we write $P(i)$ to denote the process $cp(proc, i)$, and define $as(es) = \bigcup\{i : es \bullet \alpha P(i)\}$ to return the alphabet of the process semantics of elements in $es$. The semantics of *SeqComp* is given by the following process,

$$OB \,[\![\, AB \mid as(\{new?, end?\}) \,]\!]\, (P(new?) \,[\![\, \alpha P(new?) \mid \alpha P(end?) \,]\!]\, P(end?))$$

where $OB = \Big\|\, i : proc \setminus \{e\} \bullet ((\alpha P(i) \setminus \{fn(e)\}) \cup fn(end?)) \circ cp(proc \cup \{end?\}, i)$ and $AB = as(proc \setminus \{e\}) \cup \{fn(end?)\}$.

*Split* The schema *Split* adds either an exclusive or a parallel split gateway to a BPMN pool.

$$Split \mathrel{\widehat=} [\Delta Pool;\ Commons;\ outs? : \mathbb{F}_1\, Element]$$

The operation takes as inputs gateway $new?$, a set of end events $outs?$, and sequence flow $from?$. Schema *Split* is illustrated by Diagrams 1 and B in Fig. 2, where the end event labelled $E$ is specified by the expression $((ends\, proc)\, from?)$. The illustration shows how the operation replaces $E$ with element $new?$ and the set of elements $outs?$, which contains elements labelled $F$ and $G$. We now consider the constraints specified by this operation in detail. *Split* includes constraints specified by *Commons* about $new?$, $from?$ and the before state $Pool$. It also specifies the following constraints on all input components: 1) $new?$ must be either an exclusive or a parallel split gateway; 2) $outs?$ must be a non-empty set of end events, in which elements do not share incoming sequence flows; 3) incoming sequence flows of elements in $outs?$ are not sequence flows of elements contained in $proc$ of before state $Pool$, and 4) incoming sequence flows of elements in $outs?$ are exactly the outgoing sequence flows of $new?$.

We let $e$ denote the end event $end(proc)\,from?$. The semantics of $new?$ is then defined as $P(new?) = cp((proc \setminus \{e\}) \cup outs?, new?)$, and the semantics of the set of elements $outs?$ can be modelled by the parallel composition of processes $QS = \Big\|\, o : outs? \bullet \alpha Q(o) \circ Q(o)$, where each process $Q(o)$ is defined as $cp((proc \setminus \{e\}) \cup outs?, o)$. The semantics of *Splits* is then given by the following process,

$$OB \,[\![\, AB \mid as(outs? \cup \{new?\}) \,]\!]\, (P(new?) \,[\![\, \alpha P(new?) \mid as(outs?) \,]\!]\, QS)$$

where $OB = \Big\|\, i : proc \setminus \{e\} \bullet ((\alpha P(i) \setminus \{fn(e)\}) \cup fn(\!|\ outs?\ |\!)) \circ cp(proc \cup outs?, i)$ and $AB = as(proc \setminus \{e\}) \cup fn(\!|\ outs?\ |\!)$.

*Join* The schema *Join* adds an exclusive join gateway to a BPMN pool.

$$Join \mathrel{\widehat=} [\Delta Pool;\ gate?, end? : Element]$$

The operation takes input components gateway $gate?$ and end event $end?$. Schema *Join* is illustrated by Diagrams 2 and C in Fig. 2, where the incoming sequence

flows of $E$ and $F$ are the incoming sequence flows of $gate?$. The illustration shows how the operation replaces elements $E$ and $F$ with elements $gate?$ and $end?$. Here $end?$ is labelled $G$ in the illustration. Specifically, *Join* defines the following constraints: 1) $gate?$ is either an exclusive or a parallel join gateway; 2) $end?$ is an end event; 3) $gate?$'s incoming sequence flows are incoming sequence flows of some end events contained in either $proc$ of the before state $Pool$, or one of the subprocesses contained in $proc$ of the before state $Pool$; 4) outgoing sequence flows of $gate?$ are exactly the incoming sequence flows of $end?$, and 5) incoming sequence flows of $end?$ are not sequence flows of elements contained in $proc$ component of the before state $Pool$.

We let $es = end(proc)(\!|\ in(gate?)\ |\!)$, and the semantics of $gate?$ and $end?$ are provided by $P(gate?) = cp((proc \setminus es) \cup \{end?\}, gate?)$ and $P(end?) = cp(proc \setminus es, end?)$ respectively. The semantics of *Join* is then given by the following process,

$$OB \,[\![\, AB \mid as(\{gate?, end?\}) \,]\!]\, (P(gate?) \,[\![\, \alpha P(gate?) \mid \alpha P(end?) \,]\!]\, P(end?))$$

where $OB = \|\ i : proc \setminus es \bullet ((\alpha P(i) \setminus fn(\!|\ es\ |\!)) \cup \{fn(end?)\}) \circ cp(proc \cup \{end?\}, i)$ and $AB = (as(proc \setminus es) \setminus fn(\!|\ es\ |\!)) \cup \{fn(end?)\}$.

*Loop* The schema *Loop* adds an exclusive split gateway and an exclusive join gateway to a BPMN pool to construct a loop in the pool. The operation is defined as the conjunction of schemas *ConnectSplit*, *ConnectJoin* and *Connect*,

$$Loop \mathrel{\widehat=} (ConnectSplit \wedge ConnectJoin \wedge Connect) \setminus (change, change')$$

where the declaration of these schemas are shown as follow.

$$ConnectSplit \mathrel{\widehat=} [Commons[split?/new?];\ connect? : Seqflow;\ end? : Element]$$
$$ConnectJoin \mathrel{\widehat=} [Pool;\ ch, ch', join? : Element;\ connect?, f2?, t2? : Sflow]$$
$$Connect \mathrel{\widehat=} [\Delta Pool;\ from?, f2?, t2? : Sflow;\ ch, ch', split?, join?, end? : Element]$$

Briefly, *ConnectSplit* specifies the constraints on the input exclusive split gateway, *ConnectJoin* specifies the constraints on the input exclusive join gateway, and *Connect* specifies the interdependent constraints on the two gateways.

Schema *Loop* is illustrated by Diagrams 3 and D in Fig. 2. Specificially, *Loop* performs a two-step operation: 1) replace the end event in the component $proc$ that has incoming sequence flow $from?$ with gateway $split?$ and the end event $end?$. The constraints of *Loop* ensure $connect?$ is one of $split?$'s outgoing sequence flows, and 2) add a join gateway $join?$ to the component $proc$. The constraints of *Loop* ensure that $connect?$ and $f2?$ are incoming sequence flows of $join?$ and $t2?$ is the outgoing sequence flow of $join?$. The constraints also ensure that there is an element contained in the before state of $proc$ that has $f2?$ as one of its incoming sequence flows and replaces this element's $f2?$ incoming sequence flow with $t2?$. This element is defined by the expression $(\mu\, p : proc \mid f2? \in in(p) \bullet p)$ and we let $m$ denote this element.

We let $e = end(proc)\, from?$, and $ps = (proc \setminus \{e\}) \cup \{end?\}$. The semantics of $split?$, $join?$ and $end?$ are provided by the processes $P(split?) = cp(ps, split?)$,

$P(join?) = cp(ps, join?)$ and $P(end?) = cp(ps, end?)$ respectively. We also let $P(m') = P(m)[fn(f2?) \leftarrow fn(t2?)]$, which renames all occurrences of $sf(f2?)$ to $sf(t2?)$ in $P(m)$. The semantics of *Loop* is then given by the following process,

$$
\begin{array}{l}
OB \,[\![\, as(proc \setminus \{e, m\}) \cup \{fn(end?)\} \mid as(\{split?, join?, end?, m'\})]\!] \\
\quad (P(m') \,[\![\, \alpha P(m') \mid as(\{split?, join?, end?\})]\!] \\
\quad\quad (P(split?) \,[\![\, \alpha P(split?) \mid as(\{join?, end?\})]\!] \\
\quad\quad\quad (P(join?) \,[\![\, \alpha P(join?) \mid \alpha P(end?) \,]\!]\, P(end?))))
\end{array}
$$

where $OB = \Big\|\, i : proc \setminus \{e, m\} \bullet ((\alpha P(i) \setminus \{fn(e)\}) \cup \{fn(end?)\}) \circ cp(proc \cup \{end?\}, i)$.

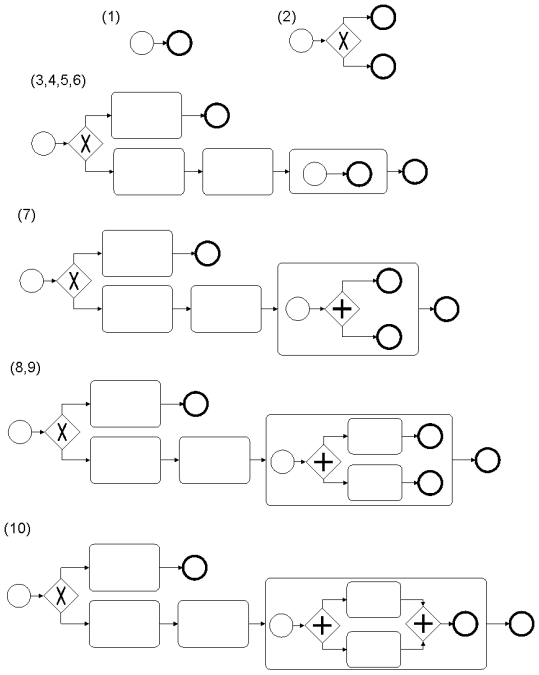


**Fig. 3.** Construction of the customer business process

**Example** Here we describe how to construct the customer business process of our online shop example in Fig. 1. A step-by-step illustration of the business process construction is shown in Figure 3. The following describes the steps shown in the figure.

1. Start with a subprocess' initial state, containing a start and an end events (Step 1);
2. Add an exclusive split gateway using *Split* (Step 2);

3. Apply *SeqComp* four times to add three task elements and one subprocess element. The subprocess element is in an initial state, again containing a start and an end events (Steps 3, 4, 5, 6);
4. Apply *Split* to add an exclusive split gateway inside the subprocess element that was added in the previous step (Step 7);
5. Add two task elements inside the subprocess element using *SeqComp* two times (Steps 8, 9) and,
6. Join two end elements inside the subprocess element using *Join* (Step 10).

### 4.2 Analysis

The combination of monotonicity and refinements allows one to verify behavioural correctness of large complex systems incrementally. We would like to show that the operations considered in the previous section to be monotonic with respect to failures refinement ($\sqsubseteq_{\mathcal{F}}$). However, we observe that *in general* these operations are *not* monotonic.

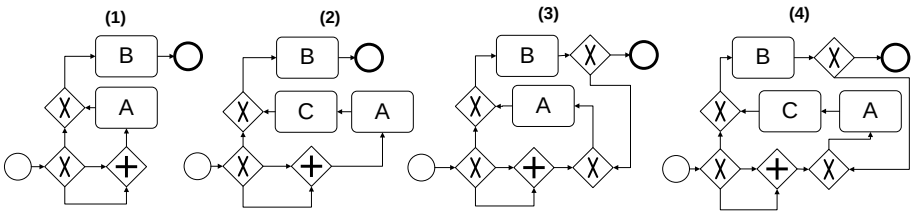


**Fig. 4.** A non-monotonic scenario

Consider the BPMN processes in Fig. 4. They are constructed by a combination of operations *SeqComp*, *Split*, *Join* and *Loop*. We let $P1$, $P2$, $P3$ and $P4$ denote the BPMN processes shown in Figs. 4(1), (2), (3) and (4) respectively We observe that both $P1$ and $P2$ deadlock, because in both cases not all of the parallel join gateway's incoming sequence flows can be triggered. Moreover, we observe that $P1$ and $P2$ admit the same behaviour, that is, $P1 \equiv_{BPMN} P2$. We now consider $P3$ and $P4$ that are constructed by applying the operation *Loop* to $P1$ and $P2$ respectively. We observe that unlike in $P1$ and $P2$, $A$ can be triggered in both $P3$ and $P4$. However, after performing $A$, $P4$ can trigger element $C$, while $P3$ cannot. As a result we have $P3 \not\equiv_{BPMN} P4$. This shows that in general not only the operations are non-monotonic, but more importantly the equivalence $\equiv_{BPMN}$ is not congruent with respect to these operations. To ensure that the operations can be applied monotonically, we assume the following conditions about BPMN diagrams and processes and state that the operations are monotonic with respect to the failures refinement; complete proofs of the following theorems can be found in the author's thesis [11, Appendix C].

(a) Given any two BPMN pools $X, Y : Pool$, if we have $pToc(X) \sqsubseteq_{\mathcal{F}} pToc(Y)$, then the set of elements $X.proc \setminus Y.proc$ can be partitioned into two sets $A$ and $B$:

(i) Each element $e \in A$ is either an exclusive split gateway or a subprocess such that there exists an element $e' \in Y.proc \setminus X.proc$ where $P(e) \sqsubseteq_{\mathcal{F}} P(e')$.

(ii) For each element $f \in B$, there exists some exclusive split gateway $e \in A$, such that there exists some element $e' \in Y.proc \setminus X.proc$ where $P(e) \sqsubseteq_{\mathcal{F}} P(e')$. Moreover, $ou(e') \subset ou(e)$ and either $in(f) \subset ou(e) \setminus ou(e')$ or there exists an element $g \in B$ such that $in(g) \subset ou(e) \setminus ou(e')$ and there exists a sequence of sequence flows connecting $g$ to $f$.

Furthermore, $Y.proc \setminus X.proc$ can be partitioned into two sets $M$ and $N$:

(iii) For each element $m \in M$, there exists exactly one element $e \in A$ such that $P(e) \sqsubseteq_{\mathcal{F}} P(m)$,

(iv) Each element $e \in N$ is an exclusive join gateway such that there exists an exclusive join gateway $f \in B$ with the same outgoing sequence flow and whose set of incoming sequence flows is a superset of $e$'s.

(b) Given any two BPMN diagrams $X, Y : Diagram$. If we have $bToc(X) \sqsubseteq_{\mathcal{F}} bToc(Y)$, we have $\mathrm{dom}\, X.pool = \mathrm{dom}\, Y.pool$ and $\forall\, i : \mathrm{dom}\, X.pool \bullet pToc(X.pool(i)) \sqsubseteq_{\mathcal{F}} pToc(Y.pool(i))$.

**Theorem 1.** ***Monotonicity.*** *Assuming BPMN diagrams satisfy Conditions (a) and (b), the composition operations are monotonic with respect to $\mathcal{F}$.*

Condition (a) is appropriate: according to our process semantics, only exclusive split gateways and subprocesses have nondeterministic behaviour. This condition ensures that when comparing the behaviour of two BPMN processes, every BPMN element from one BPMN process either is related to an element in the other BPMN process via refinement or is only reachable due to nondeterministic behaviour that is removed due to the refinement in the other BPMN process. This condition ensures that there are no hidden behaviour such as element $C$ in Fig. 4 (2). Condition (b) is also appropriate: it is reasonable to compare behaviour of business collaborations if they have the same participants during compositional development. Participants in a business collaboration are typically decided a priori before designing and refining individual processes. These conditions do not reduce the practical expressiveness of BPMN. In particular we have validated the expressiveness via case studies in which we have constructed and verified complex business processes compositionally [11, Chapter 8].

In general, for any subprocess $s$, the CSP process that models $s$'s behaviour can be generalised as $C[S]$ where $S$ is the CSP process that models elements directly contained in $s$. Here $C[.]$ is a CSP process context that models the incoming, outgoing sequence flows and message flows of $s$. The following result shows that refinements are preserved from $S$ to $C[S]$.

**Theorem 2.** *Given any subprocess $s$ satisfying Conditions (a) and (b), and that its behaviour is modelled by CSP process $C[S]$, where $S$ is the CSP process that models elements contained in $s$ and $C[.]$ is a CSP process context that models $s$'s sequence flows and message flows. Let $t$ be any subprocess whose behaviour is modelled by CSP process $C[T]$ and $T$ is the CSP process that models elements contained in $t$. If we have both $S \sqsubseteq_{\mathcal{F}} T$ and $C[S] \sqsubseteq_{\mathcal{F}} C[T]$, then we have $C[S'] \sqsubseteq_{\mathcal{F}}$*

*$C[T']$ where $S'$ and $T'$ are the results of applying any one of the composition operations on $S$ and $T$ respectively.*

A consequence of Theorems 1 and 2 is that refinement is preserved between a subprocess and the BPMN subprocess or pool that contains it. The following result lifts the composition operations to BPMN diagrams, and follows immediately from the fact that the CSP parallel operator $\|$ is monotonic with respect to refinements.

**Corollary 1.** *Given a BPMN process $X$ that directly contains some subprocess $s$, such that $X$'s behaviour is modelled by the CSP process $P(s) \,[\![\, \alpha P(s) \mid \alpha D \,]\!]\, D$, where $D$ models the behaviour of other elements directly contained in $X$. For any $s'$ such that $P(s) \sqsubseteq_{\mathcal{F}} P(s')$ we have $P(s) \,[\![\alpha P(s) \mid \alpha D]\!]\, D \sqsubseteq_{\mathcal{F}} P(s') \,[\![\alpha P(s') \mid \alpha D]\!]\, D$*

Monotonic operations preserve refinement-closed properties. For example, we previously formalized the notion of behavioural compatibility on BPMN pools [11, 13] as a binary relation *compatible*$(P, Q)$ on BPMN pools $P$ and $Q$, such that *compatible*$(P, Q)$ if and only if $P$ and $Q$ are deadlock free and their collaboration is also deadlock free. We further showed that *compatible* is a refinement-closed property. As a result, if we let $G$ be one of the composition operations on *Pools*, if *compatible*$(G(P), G(Q))$, then for all $pToc(P) \sqsubseteq_{\mathcal{F}} pToc(P')$ and $pToc(Q) \sqsubseteq_{\mathcal{F}} pToc(Q')$, *compatible*$(G(P'), G(Q'))$. Furthermore, due to monotonicity, the equivalence $\equiv_{BPMN}$ defined in Definition 1 is a congruence with respect to the composition operations.

**Corollary 2.** *Assuming BPMN processes satisfy Conditions (a) and (b), the equivalence $\equiv_{BPMN}$ is a congruence with respect to composition operations.*

A congruence relationship allows one to substitute one part of a BPMN process with another that is semantically equivalent and obtain the same BPMN process.

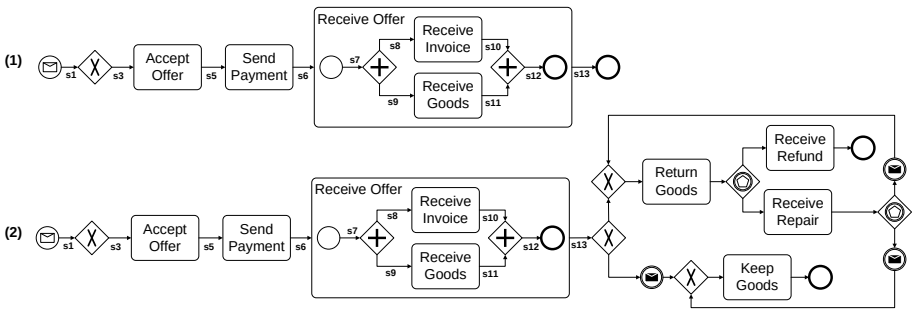


**Fig. 5.** Variants of the customer business process

Back to the running example. Fig. 5(1) shows an optimistic version of the customer business process. It is modelled by BPMN pool *OpCustomer*. After receiving an offer from the online shop, the customer eventually always accepts

the offer. Equation 3 defines process $OCP$ that models pool $OpCustomer$, where for all $i \in \{sCP, s3, s5, s6, s13\}$ process $P(i)$ is defined in Equation 1.

$$P(s1) = (s.s1 \rightarrow Skip \fatsemi (s.s2 \rightarrow Skip \sqcap s.s3 \rightarrow Skip) \fatsemi P(s1)) \,\Box\, c.s14 \rightarrow Skip$$
$$OCP = \Big\| i : \{sCP, s1, s3, s5, s6, s13\} \bullet \alpha P(i) \setminus \{c.s4\} \circ P(i) \tag{3}$$

In fact the optimistic customer process is a refinement of the customer process; we verify this by checking the refinement $CP \sqsubseteq_{\mathcal{F}} OCP$ using FDR. Suppose we extend the customer's business process with the following return policy: "After receiving the goods and the invoice, the customer may decide to either keep the goods or return them for repair. Depending on the policy of the online shop, if the customer chooses to return her goods for repair, the shop may either provide a full refund, or repair the goods and deliver them back to the customer. After every repair, the customer has the choice to send the goods back again if further repairs are required.".

Fig. 5(2) shows the result of extending the optimistic customer business process with the above return policy. Here we observe that this extension can be constructed by a combination of operations *SeqComp*, *Split* and *EventSplit*, *Join* and *EventLoop*. Note that the same combination of operations can be applied to the original customer business process to model this return policy. If we let $CP'$ be the resulting CSP process modelling the extended version of $CP$ and $OCP'$ be that of $OCP$, by Theorem 1, we have $CP' \sqsubseteq_{\mathcal{F}} OCP'$.

## 5 Related Work

Beside the work described in this paper and our earlier work [12], CSP has been applied to formalize other business process modelling languages. For example Yeung [15] mapped the Web Service Business Process Execution Language (WS-BPEL) and the Web Service Choreography Description Language (WS-CDL) to CSP to verify the interaction of BPEL processes against the WS-CDL description; his approach considers traces refinement and hence only safety properties. There have been attempts to formalize BPMN behaviour using existing formalisms (for example, Dijkman et al. [2]), which focus on the semantic definition of BPMN diagrams rather than their construction. Morale et al. [6, 7] designed the Formal Composition Verification Approach (FVCA) framework based on an extended timed version of CSP to specify and verify BPMN diagrams. Similar to us, they achieve compositionality by considering the parallel combination of individual business process participants. However, their approach does not consider the semantics of constructing individual BPMN processes. To the best of our knowledge, Istoan's work [5] is the first attempt at defining composition operations on BPMN. He provided the composition operations to construct a BPMN process by composing two BPMN processes. He also provided these operations with a semantics in terms of Petri Nets [9]. His notion of refinement is that of functional extension while our work considered the reduction of nondeterminism [3]. Moreover, Istoan did not consider the semantic property of the composition operations to allow compositional development.

## 6 Summary

In this paper we introduced a set of composition operations to construct BPMN diagrams. We provided these operations a CSP semantics, and characterised the conditions under which these operations guarantee monotonicity with respect to $\mathcal{F}$. Refinement-closed properties, such as behavioural compatibility, are preserved by monotonic operations and these operations enable compositional development of business processes.

*Acknowledgment* We wish to thank Jeremy Gibbons for suggesting improvements to this paper and Stephan Schroevers for proof reading it. We would like to thank anonymous referees for useful suggestions and comments.

## References

1. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press Ltd., 1972.
2. R. M. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 2008.
3. R. Eshuis and M. M. Fokkinga. Comparing refinements for failure and bisimulation semantics. *Fundamenta Informaticae*, 52(4), 2002.
4. Formal Systems (Europe) Ltd. *Failures-Divergences Refinement, FDR2 User Manual*, 1998. `www.fsel.com`.
5. P. Istoan. Defining Composition Operators for BPMN. In *Proceedings of 11th International Conference on Software Composition*, volume 7306 of *LNCS*, 2012.
6. L. E. Mendoza and M. I. Capel. Automatic Compositional Verification of Business Processes. In *Enterprise Information Systems*, volume 24 of *LNBIP*. Springer, 2009.
7. L. E. Mendoza Morales, M. I. C. Tuñón, and M. Pérez. A Formalization Proposal of Timed BPMN for Compositional Verification of Business Processes. In *Enterprise Information Systems*, volume 73 of *LNBIP*. Springer, 2011.
8. OMG. *Business Process Modeling Notation, V1.1*, Feb. 2008. Available Specification, `www.bpmn.org`.
9. C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut fur instrumentelle Mathematik, Bonn, 1962.
10. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.
11. P. Y. H. Wong. *Formalisations and Applications of Business Process Modelling Notation*. DPhil thesis, University of Oxford, 2011. Available at `http://ora.ox.ac.uk/objects/uuid:51f0aabc-d27a-4b56-b653-b0b23d75959c`.
12. P. Y. H. Wong and J. Gibbons. A Process Semantics for BPMN. In *Proceedings of 10th International Conference on Formal Engineering Methods*, volume 5256 of *LNCS*. Springer, Oct. 2008.
13. P. Y. H. Wong and J. Gibbons. Property Specifications for Workflow Modelling. *Science of Computer Programming*, 76(10), Oct. 2011.
14. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
15. W. L. Yeung. Mapping WS-CDL and BPEL into CSP for Behavioural Specification and Verification of Web Services. In *Proceedings of 4th European Conference on Web Services*, 2006.