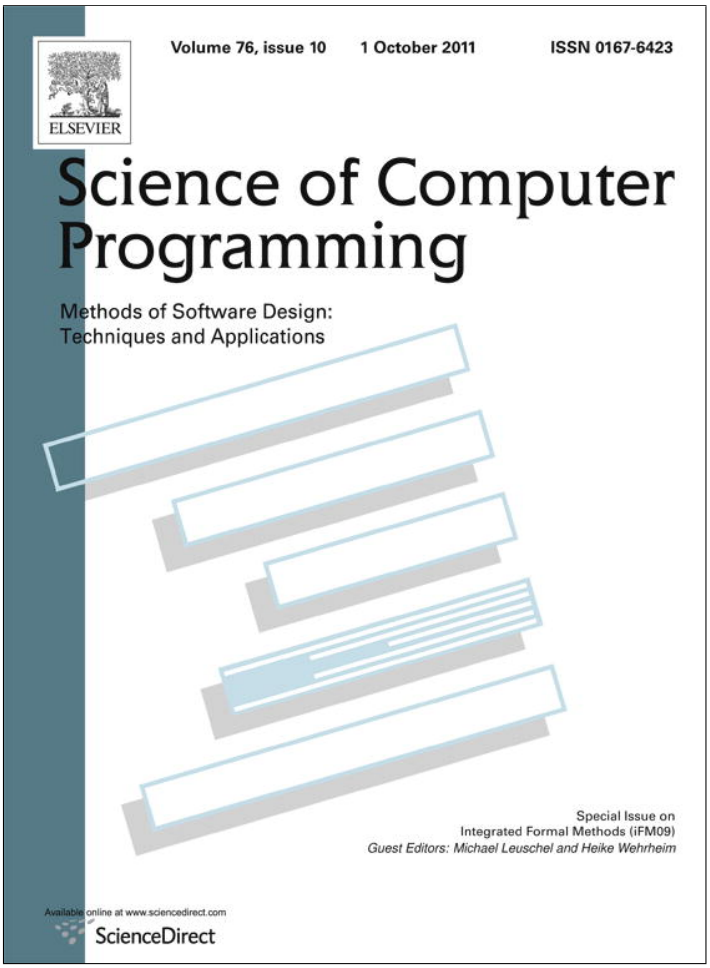

Science of Computer Programming 76 (2011) 942–967

Contents lists available at ScienceDirect

Science of Computer Programming

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

Property specifications for workflow modelling

Peter Y.H. Wong *, Jeremy Gibbons

Computing Laboratory, University of Oxford, United Kingdom

ARTICLE INFO

Article history:
Received 14 May 2009
Received in revised form 16 September 2010
Accepted 28 September 2010
Available online 7 October 2010

Keywords:
Compatibility
CSP
Workflow specification
Linear temporal logic
Property specification patterns
Workflow verification

ABSTRACT

Previously we provided two formal behavioural semantics for the Business Process Modelling Notation (BPMN) in the process algebra CSP. By exploiting CSP's refinement orderings, developers may formally compare their BPMN models. However, BPMN is not a specification language, and it is difficult and sometimes impossible to use it to construct behavioural properties against which other BPMN models may be verified. This paper considers a pattern-based approach to expressing behavioural properties. We describe a property specification language *PL* for capturing a generalisation of Dwyer et al.'s Property Specification Patterns, and present a translation from *PL* into a bounded, positive fragment of linear temporal logic, which can then be automatically translated into CSP for simple refinement checking. We present a detailed example studying the behavioural properties of an airline ticket reservation business process. Using the same example we also describe some recent results on expressing behavioural compatibility within our semantic models. These results lead to a compositional approach for ensuring deadlock freedom of interacting business processes.

1. Introduction

Formal developments in workflow languages allow developers to describe their workflow systems precisely, and permit the application of model checking to automatically verify models of their systems against formal specifications. One of these workflow languages is the Business Process Modelling Notation (BPMN) [22], for which we previously provided two formal semantic models [33,34] in the process algebra CSP [26]. Both models leverage the refinement orderings that underlie CSP's denotational semantics, allowing BPMN to be used for specification as well as modelling of workflow processes. However, some behavioural properties, against which developers might be interested to verify their workflow processes, might not be easy or even possible at all to capture in BPMN. We illustrate this via a motivating example in the next section.

1.1. Motivating example

As a motivating example, we consider a BPMN diagram describing an airline ticket reservation business process [30]. The diagram is shown in Fig. 2. In particular, this example focuses on the workflow of the travel agent, described by the BPMN pool labelled *Agent*. We investigate properties of the complete business process in Section 5. The main purpose of the travel agent is to mediate interactions between the traveller who wants to buy an airline ticket and the airline who supplies it. This section provides an informal description of travel agent's workflow. Here, we assume the readers to be familiar with BPMN and CSP. An overview of BPMN is given in Section 2.1 and an overview of CSP's syntax and semantics is given in Section 2.2.

Note that it is not possible in (machine-readable) CSP to declare events containing space characters, we therefore replace space characters in each activity name with underscores throughout this paper.

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

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.09.007

Fig. 1. A BPMN diagram capturing requirement.

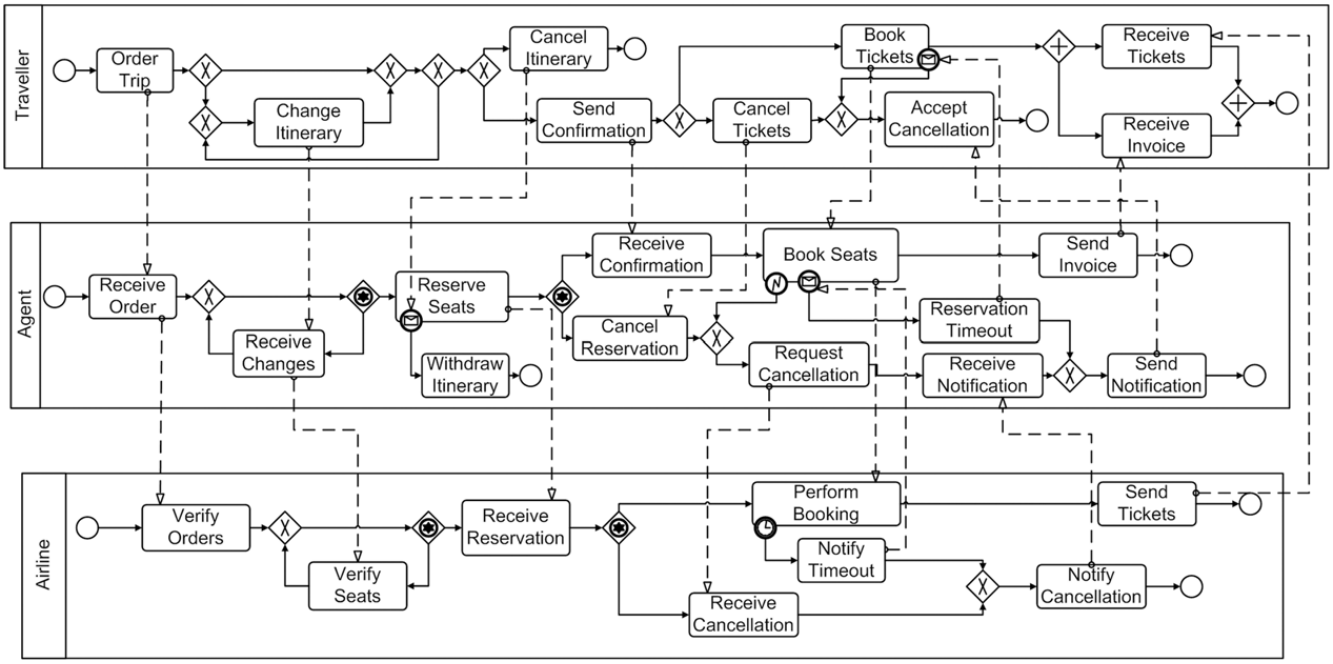

Fig. 2. Airline ticket reservation.

Once the travel agent receives an initial order from the traveller (*Receive_Order*); the agent verifies seating availability with the airline, this is denoted by the message flow (dashed line) connecting task *Receive_Order* from the travel agent to task *Verify_Order* from the airline reservation system. In order to cater for the possibility of the traveller making changes to her itinerary, the travel agent verifies with the airline the availability of the seats (*Receive_Changes*) every time there is a change to the itinerary. Once the traveller has agreed upon a particular itinerary, the travel agent may reserve the seats for the traveller (*Reserve_Seats*). During the reservation period the traveller may cancel her itinerary, thereby releasing her hold on the seats; this is modelled as a message exception flow of task *Reserve_Seats*.

Once the reservation has been completed, the travel agent may receive a confirmation notice from the traveller (*Receive_Confirmation*), in which case he receives the credit card information from the traveller and proceeds with the booking (*Book_Seats*). The travel agent may also receive cancellation of the reservation (*Cancel_Reservation*), in which case he will request a cancellation from the airline (*Request_Cancellation*), wait for a notification confirming the cancellation from the airline (*Receive_Notification*), and send it to the traveller (*Send_Notification*). Also during the booking phase, either an error (e.g. incorrect card information), modelled as an error exception flow, or a time out (*Reservation_Timeout*) may occur; in both cases, a corresponding notification confirming the cancellation will be sent to the traveller. Otherwise, an invoice for the booking will be sent to the traveller for billing (*Send_Invoice*). Note that from the point of view of the travel agent, time restriction on booking should be determined by the particular airline, therefore the time out is modelled as a message exception flow attached to task *Book_Seats*.

Here is one of the requirements this travel agent description should meet:

> The travel agent must not allow any kind of cancellation after the traveller has booked her tickets, if an invoice is to be sent to the traveller.

Assuming process *Agent* models the semantics of the travel agent diagram, one might attempt to draw a BPMN diagram like the one shown in Fig. 1 to express the negation of the property, and prove the satisfiability of *Agent* by showing this diagram does not failures-refine the process $Agent \setminus N$ where N is the set of CSP events that are not associated with tasks *Book_Seat*, *Request_Cancellation*, *Reservation_Timeout* and *Send_Invoice*. However, while this requirement should also permit other behaviours such as task *Request_Cancellation* being performed before task *Book_Seat*, experience shows it to be difficult to specify all these behaviours in the same BPMN diagram. This is because BPMN is a modelling notation for describing the *performance* of behaviour, in general it is difficult to use it to specify liveness properties about the *refusal* of some behaviour within a context while asserting the *availability* of it outside the context. We therefore need a different approach that will allow domain specialists to express property specifications for verification of workflow processes.

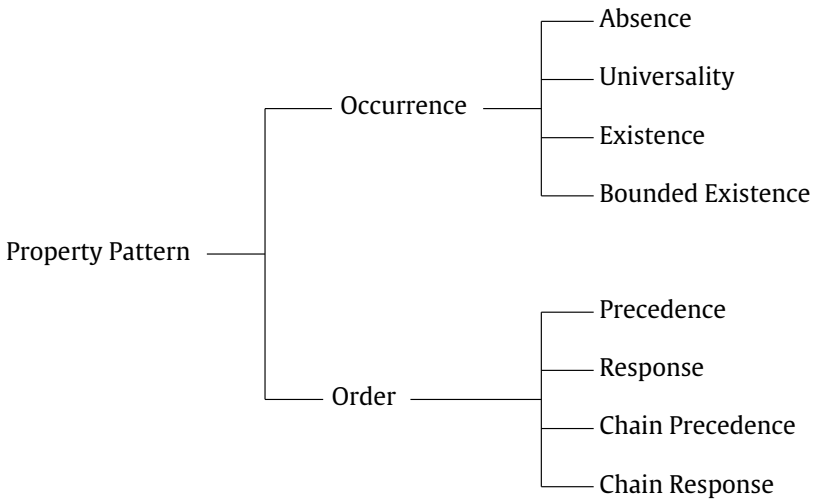

Fig. 3. Pattern hierarchy.

1.2. Property specification patterns

This paper proposes the application of Dwyer et al.'s *Property Specification Patterns* [7] to assist domain specialists in specifying behavioural properties for BPMN processes. Specification patterns are generalised specifications of properties for finite-state verification. They are intended to describe the essential structure of commonly occurring requirements on the permissible patterns of behaviours in a finite-state model of a system; the subset of BPMN considered in this paper only models finite-state systems. This is the subset of BPMN that we have provided a process semantics in CSP and does not contain BPMN constructs for modelling data flow and transactional behaviour [33,34]. Fig. 3 illustrates the hierarchy of the property patterns [28]. There exist two major groups—*order* and *occurrence*. Each pattern has a scope, the context in which the property must hold. For example, the property "task *A* cannot happen after task *B* and before task *C*" will fall into the *absence* pattern, which states that a given state/event does not occur within a scope. In this case, the property may be expressed as the absence of task *A* in the scope *after task B until task C*. The different types of scope are *Global*, *Before Q*, *After Q*, *Between Q and R* and *After Q until R*, where *Q* and *R* are states.

Currently, property patterns have been expressed in a range of formalisms such as linear temporal logic [18] and computation tree logic; however, behavioural verifications of CSP processes are carried out by proving a refinement between the specification and the implementation processes. This means CSP is also a *specification language*, and suitable for formalisation of property patterns.

1.3. Nondeterministic Interleaving

While the property patterns cover a comprehensive set of behavioural requirements, it is possible to generalise patterns in a process-algebraic setting by considering *patterns of behaviour* rather than an individual state or event within a scope. For example, we may like to express the property "the parallel execution of task *A* and either task *D* or task *E* cannot happen after task *B* and before task *C*". Here the pattern of behaviours is "the parallel execution of task *A* and either task *D* or task *E*". Consider again the motivating example in Section 1.1, the pattern of behaviours in the requirement is "any kind of cancellation". This is because a cancellation may be triggered by a request from the travel agent (*Request_Cancellation*) or a time out (*Reservation_Timeout*).

Although CSP is equipped with nondeterministic (internal) choice as one of its standard operators, there is no nondeterministic version of parallel composition; this means that while assertion (1) holds under failures refinement, assertion (2) does not.

$$a \rightarrow Skip \sqcap b \rightarrow Skip \sqsubseteq_{\mathcal{F}} a \rightarrow Skip \tag{1}$$

$$a \rightarrow Skip \,|||\, b \rightarrow Skip \not\sqsubseteq_{\mathcal{F}} a \rightarrow b \rightarrow Skip \tag{2}$$

This is because the parallel operators in CSP may be defined using the deterministic choice $\Box$ operator; here we show the semantic equivalence of interleaving with its sequential counterpart.

$$a \rightarrow Skip \,|||\, b \rightarrow Skip \equiv a \rightarrow b \rightarrow Skip \,\Box\, b \rightarrow a \rightarrow Skip$$

A nondeterministic version of the parallel operators, particularly interleaving, will be very useful for specifying behavioural properties for workflow processes. Here we show the semantic equivalence of nondeterministic interleaving $\sqcap\!\sqcap$ with its sequential counterpart. The $\sqcap\!\sqcap$ operator is formally defined in Section 3.

$$a \rightarrow Skip \sqcap\!\sqcap b \rightarrow Skip \equiv a \rightarrow b \rightarrow Skip \sqcap b \rightarrow a \rightarrow Skip$$

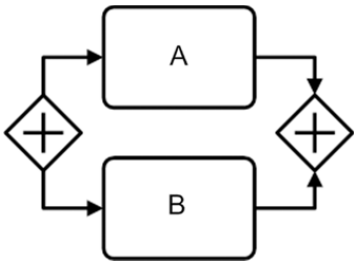

Fig. 4. Parallel execution.

For example, Fig. 4 shows part of a BPMN diagram executing tasks *A* and *B* in parallel. With our timed semantics for BPMN [34] it is possible to specify timing constraints for these tasks, and the diagram may then be interpreted over the timed model. Without a nondeterministic version of the parallel operators, it will be difficult to specify the interleaving of *A* and *B* without considering the ordering of their execution due to their timing constraints.

1.4. Contributions and approach

The contribution of this paper is twofold.

- We present a small property specification language *PL* for specifying behavioural properties of business processes described as BPMN diagrams. *PL* is based on the generalisation of Dwyer et al.'s Property Specification Pattern, allowing the specification of the occurrence of a given pattern of behaviour. The aim of *PL* is to enable behavioural property specifications of BPMN processes accessible to business process designers who cannot be assumed to have knowledge of formal methods. Furthermore, properties described using *PL* are automatically verifiable by model checking.
- We provide some results on behavioural compatibility between interacting BPMN processes. These results lead to a compositional approach for ensuring deadlock freedom of interacting business processes.

1.4.1. Property specification

We provide a CSP formalisation of the set of generalised property specification patterns, in which we consider admissible sequences of patterns of behaviours, rather than individual events, within a scope. The construction of the CSP model for each of the patterns proceeds in two stages:

- We define a small property specification language *PL*, based on the generalised patterns, for describing behavioural properties, and then provide a function that returns a linear temporal logic (*LTL*) expression that specifies the behaviour properties.
- We then translate the given *LTL* expression into its corresponding CSP process based on Lowe's interpretation of *LTL* [16]; using this, one may check whether a workflow system behaves according to a property specification.

Specifically we provide a function which translates each of the property patterns into the *bounded, positive fragment* of *LTL* [16], denoted by *BTL*, defined by the following grammar.

$$\phi \in BTL ::= \phi \wedge \phi \mid \phi \vee \phi \mid \bigcirc\phi \mid \Box\phi \mid \phi\, \mathcal{R}\, \phi \mid a \mid \neg a \mid \quad \textbf{where } a \in \Sigma$$
$$\texttt{available}\, a \mid \texttt{true} \mid \texttt{false} \mid \texttt{live} \mid \texttt{deadlocked}$$

where operators $\neg$, $\wedge$ and $\vee$ are standard logical operators, and $\bigcirc$, $\Box$ and $\mathcal{R}$ are standard temporal operators for *next*, *always* and *release*. This fragment also extends the original logic with atomic formulae for specifying *availability* of events as well as their performance. Here we describe briefly their intended meaning:

- a—the event a is available to be performed initially, and no other events may be performed;
- `available` a—the event a must not be refused initially, and other events may be performed;
- `live` and `deadlock`—the system is live (equivalent to $\bigvee_{a\in\Sigma} a$) or deadlocked (equivalent to $\bigwedge_{a\in\Sigma} \neg a$), respectively;
- `true` and `false`—logical formulae with their normal meanings.

Usually when checking whether a (workflow) system, modelled as a CSP process, satisfies a certain behavioural property, which is also modelled as a CSP process, one would check to see the former refines the latter under the stable failures semantics [26], since this model captures both safety and liveness properties. However, Lowe [16] has shown that the stable failures model is not sufficient to capture temporal logic specifications, and that a finer model known as the refusal traces model ($\mathcal{RT}$) [21] is required. Furthermore, Lowe has also shown that it is impossible to capture the *eventually* ($\Diamond$) and *until* ($\mathcal{U}$) temporal operators as well as the negation operator ($\neg$) in general. This is because the eventually operator deals with *infinite traces*, which are not suitable in general in finite-state checking, and since $\Diamond\,\phi = \texttt{true}\,\mathcal{U}\,\phi$, it is also not possible, in general, to capture the *until* operator. While $\Diamond\,\phi = \neg(\Box\neg\phi)$, it is possible to capture the *always* operator; therefore it is not possible, in general to capture negation, unless it is over atomic formulae as given by the grammar above. Our function

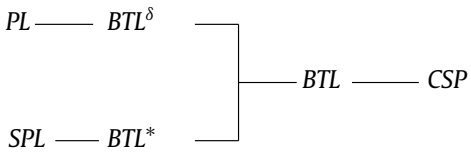

Fig. 5. The flow of language mapping.

reflects this by translating a given generalised pattern into a corresponding expression in *BTL*. We say a system modelled by the CSP process P satisfies a behavioural property, written as $P \models \psi$ where ψ is the temporal logic expression, if and only if $Spec(\psi) \sqsubseteq_{\mathcal{RT}} P$ where $Spec(\psi)$ is the CSP specification for ψ.

Fig. 5 shows the translation flow from behavioural properties specified in *PL* to CSP processes for simple refinement checks. Specifically, a property is specified as an *PL* term. Each nondeterministic pattern of behaviours inside the property's scope is specified as a *SPL* term; *SPL* is a sub-language of *PL*. Each *SPL* term is translated into *BTL* via *BTL*∗. *BTL*∗ is a small augmentation of *BTL* for assisting the translation of *SPL*. The translations from *SPL* to *BTL*∗ and from *BTL*∗ to *BTL* are formally defined in Section 3. The translated *BTL* expression replaces the *SPL* term in the original *PL* term. This *PL* term is then translated into *BTL* via BTL^{δ}. BTL^{δ} is a variant of *BTL* with the addition of two derived operators that can be completely characterised by *BTL*. BTL^{δ} is defined purely to facilitate the translation from *PL* to *BTL*. The translations from *PL* to BTL^{δ} and from BTL^{δ} to *BTL* are described in Section 4. Having obtained the final *BTL* expression of the *PL* term, we apply Lowe's translation on the expression to obtain the corresponding CSP process that is suitable for simple refinement checks.

1.4.2. Behavioural compatibility

When considering business collaborations such as the airline ticket reservation business process shown in Fig. 2, we would like to ensure compatible behaviour between business participants. From the point of view of component interactions, we require the business collaboration to be at least deadlock free. In this paper we formalise behavioural compatibility with respect to our semantic models [33] and provide a compositional approach to ensuring the deadlock freedom of many interacting business participants.

1.5. Assumptions and structure of the paper

In the rest of this paper we assume the behaviour of the system we are interested in is modelled by some non-divergent process P; this can be achieved by either only considering BPMN models with no self-looping elements, that is, elements whose incoming and outgoing sequence flows intersect, or by first checking divergent freedom using a model checker such as FDR [8]. Furthermore, we assume the alphabet of the *specification process* of the property, that is the set of all possible events the process may perform, only falls under the context of the property. This is possible because in CSP, one may always construct some *partial specification* X and prove some system Y satisfies it by checking the refinement assertion $X \sqsubseteq Y \setminus (\alpha Y \setminus \alpha X)$ where αP is the alphabet of P, assuming $\alpha X \subseteq \alpha Y$.

The structure of the remainder of this paper is as follows. Section 2 informally describes BPMN, and gives an overview of CSP. In Section 3 we introduce *SPL*; we define function *pattern*, which takes a nondeterministic system specified in *SPL* and returns its corresponding temporal logic expression in *BTL*. We provide justification for the translation over the refusal traces model. In Section 4 we present the complete language *PL*; we define a function *tl*, which takes a property specification in *PL* and returns its corresponding temporal logic expression in *BTL*. We revisit the travel agent running example and demonstrate how to specify the behavioural property in *PL*. We investigate the behavioural properties of the complete airline ticket reservation business process in Section 5, and present our recent results on behavioural compatibility in Section 6. We discuss related works on BPMN formalisations, LTL model checking for CSP and behavioural compatibility in Section 7.

2. Notations

2.1. BPMN

BPMN [22] is a graphical modelling language for business analysts to specify business processes as workflows. It is the language that bridges the gap between visualisation of the business processes and their executable implementation such as those defined in XML-based languages like Business Process Execution Language [4] (WS-BPEL), which is an XML-based language standardised by OASIS[1] for implementing business processes based on Web Services.

States in our subset of BPMN, shown in Fig. 6, can either be pools, tasks, subprocesses, multiple instances or control gateways, each linked by a normal sequence, an exception sequence flow, or a message flow. A normal sequence flow can

[1] http://www.oasis-open.org/.

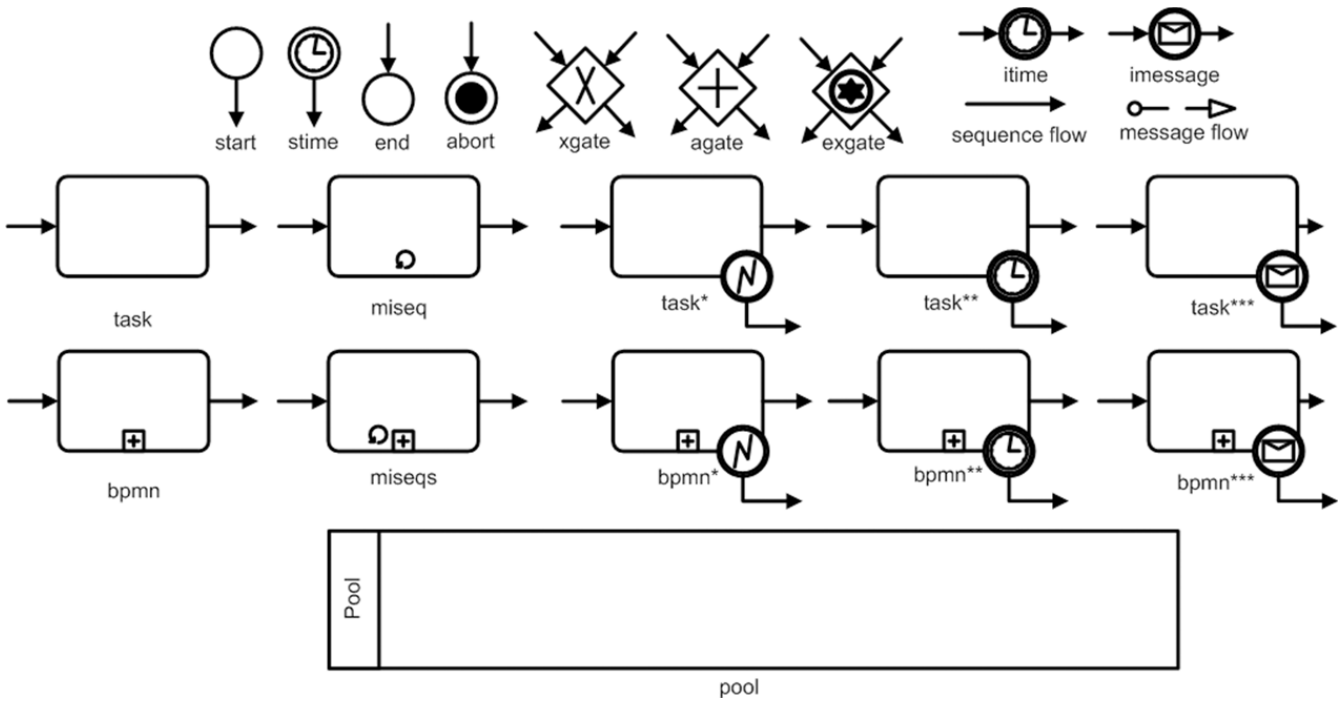

Fig. 6. States of BPMN diagrams.

be either incoming to or outgoing from a state and have associated guards; an exception sequence flow, depicted by the states labelled *task**, *bpmn**, *task*** and *bpmn***, represents an occurrence of error within the state. While sequence flows represent control flows within individual *local* diagrams, message flows represent unidirectional communication between states in different local diagrams. A *global* diagram hence is a collection of local diagrams, connected via message flows.

In Fig. 6, there are two types of start state, *start* and *stime*. A *start* state models the start of the business process in the current scope by initiating its outgoing transition; it has no incoming transition and only one outgoing transition. The *stime* state is a variant start state; it initiates its outgoing transition when a specified duration has elapsed. There are also two types of intermediate state, *itime* and *imessage*. An *itime* state is a delay event; after its incoming transition is triggered, the delay event waits for the specified duration before initiating its outgoing transition. An *imessage* state is a message event; after its incoming transition is triggered, the message event waits until a specified message has arrived before initiating its outgoing transition. Both types of state have a maximum of one incoming transition and one outgoing transition.

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

Our subset of BPMN contains three types of decision state, *xgate*, *exgate* and *agate*. Each of them has one or more incoming sequence flows and one or more outgoing sequence flows. An *xgate* state is a data-based exclusive choice gateway; it accepts one of its incoming flows and takes one of its outgoing flows, based on the evaluation of a Boolean expression using process data [22, page 71]. An *exgate* state, on the other hand, is an event-based exclusive choice gateway; it accepts one of its incoming flows and takes one of its outgoing flows, based on events, such as the receipt of a message, that occur at that point in the process [22, page 75]. An *agate* state is a parallel gateway, which waits for all of its incoming flows before initialising all of its outgoing flows.

A *task* state describes an atomic activity, and has exactly one incoming and one outgoing transition. It takes a unique name for identifying the activity. In the environment of the timed semantic model, each atomic task must take a positive amount of time to complete. A *bpmn* state describes a subprocess state. It is a business process by itself and so it models a flow of BPMN states. In this paper, we assume all our subprocess states are expanded [22, Section 9.4.2]; this means that we model the internal behaviours of the subprocesses. The state labelled *bpmn* in Fig. 6 depicts a collapsed subprocess state where all internal details are hidden; this state has exactly one incoming and one outgoing transition.

Also in Fig. 6 there are graphical notations labelled *task**, *bpmn**, *task***, *bpmn***, *task**** and *bpmn****, which depict a task state and a subprocess state with an exception sequence flow. There are three types of exception associated with task and subprocess states in our subset of BPMN states. Both states *task** and *bpmn** are examples of states with an *ierror* exception flow that models an interruption due to an error within the task or subprocess state; the states *task*** and *bpmn*** are examples of states with a timed exception flow, and model an interruption due to an elapse of the specified duration; the states *task**** and *bpmn**** are examples of states with a message exception flow, and model an interruption upon receiving the specified message. Each task and subprocess state can have a maximum of one timed exception flow, although it may have multiple error and message exception flows.

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

The graphical notation *pool* in Fig. 6 forms the outermost container for each local diagram, representing a single business process; only one execution instance is allowed at any one time. Each local diagram contained in a pool can also be a participant within a business collaboration (global diagram) involving multiple business processes. While *sequence flows* are restricted to an individual pool, *message flows* represent communications between pools.

2.2. CSP

In CSP [26], 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.

$$P, Q ::= P \mid\mid\mid Q \mid P \llbracket A \rrbracket Q \mid P \llbracket A \mid B \rrbracket Q \mid P \setminus A \mid P \triangle Q \mid P \Box Q \mid P \sqcap Q \mid P \fatsemi Q \mid e \rightarrow P \mid \mathit{Skip} \mid \mathit{Stop}$$

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

Process $P \mid\mid\mid Q$ denotes the interleaved parallel composition of processes P and Q. Process $P \llbracket A \rrbracket Q$ denotes the partial interleaving of processes P and Q sharing events in set A. Process $P \llbracket A \mid B \rrbracket Q$ denotes parallel composition, in which P and Q can evolve independently but must synchronise on every event in the set $A \cap B$; the set A is the alphabet of P and the set B is the alphabet of Q, and no event in $A \cup B$ can occur without the cooperation of P and Q respectively. We write $\mid\mid\mid i : I \bullet P(i)$, $\mid\mid [A]\, i : I \bullet P(i)$ and $\mid\mid i : I \bullet A(i) \circ P(i)$ 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 and 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 is equipped with three standard behavioural models: traces, stable failures and failures-divergences, in order of increasing precision. Here we provide an overview of the stable failures semantics before describing the finer refusal traces model.

2.2.1. Stable failures

In the stable failures semantics ($\mathcal{F}$), each CSP process is characterised as a set of *traces* and *failures*. A trace of a process is a sequence of events that the process may perform. The traces model of some process P, denoted by $\mathit{traces}(P)$, is hence a *prefix-closed* set of sequences of events. A failure is a pair (s, X) where $s \in \mathit{traces}(P)$ leading to a stable state and (P/s) **ref** X where P/s represents process P after the trace s, and P **ref** X means that P can refuse X initially. The failures model of P, denoted by $\mathit{failures}(P)$, is a set of all failures such that:

$$\forall s : \mathrm{seq}\, \Sigma;\ r : \mathbb{P}\, \Sigma \bullet (s, r) \in \mathit{failures}(P) \Rightarrow s \in \mathit{traces}(P) \wedge \forall q : \mathbb{P}\, \Sigma \bullet q \subseteq r \Rightarrow (s, q) \in \mathit{failures}(P)$$

We say process Q *failures-refines* process P precisely when $\mathit{traces}(Q)$ is a subset of $\mathit{traces}(P)$ and $\mathit{failures}(Q)$ is a subset of $\mathit{failures}(P)$, that is:

$$P \sqsubseteq_{\mathcal{F}} Q \Leftrightarrow \mathit{traces}(P) \supseteq \mathit{traces}(Q) \wedge \mathit{failures}(P) \supseteq \mathit{failures}(Q)$$

2.2.2. Refusal traces

While the stable failures semantics is the standard model for reasoning about liveness properties of (divergence free) processes, Lowe [16] has demonstrated that these models are inadequate for capturing temporal logic of the form described in the previous section. The solution is to use the refusal traces model ($\mathcal{RT}$) [21].

In the refusal traces model, each CSP process may be denoted as a set of refusal traces; each refusal trace is an alternating sequence of refusal information and events. More precisely, a refusal trace takes the form,

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

where each X_i is a refusal set, and each a_i is an event. This trace represents that the process can refuse X_1, perform a_1, refuse X_2, perform a_2, etc. In this particular example the refusal trace finishes by refusing Σ (the set of all possible events), i.e. deadlocking. Here we write $\langle\rangle$ for the empty sequence, $\langle a, b\rangle$ for a sequence of a followed by b and $s \frown t$ for the concatenation of sequences s and t. If s and t are refusal traces and s is not a deadlocking refusal trace, $s \frown t$ is also a refusal trace. We write $\mathcal{RT}[\![P]\!]$ for the refusal traces of CSP process P. We now present the refusal traces semantics for some of the CSP operators,

$$
\begin{aligned}
\mathcal{RT}[\![Stop]\!] &= \{\langle\rangle, \langle\Sigma\rangle\} \\
\mathcal{RT}[\![a \to P]\!] &= \{\langle\rangle\} \cup \{\langle X, a\rangle \frown tr \mid a \notin X \wedge tr \in \mathcal{RT}[\![P]\!]\} \\
\mathcal{RT}[\![P \sqcap Q]\!] &= \mathcal{RT}[\![P]\!] \cup \mathcal{RT}[\![Q]\!] \\
\mathcal{RT}[\![P \,\Box\, Q]\!] &= \{\langle\rangle\} \cup (\textbf{if } \langle\Sigma\rangle \in \mathcal{RT}[\![P]\!] \cap \mathcal{RT}[\![Q]\!] \textbf{ then } \{\langle\Sigma\rangle\} \textbf{ else } \emptyset) \cup \\
&\quad \{\langle X, a\rangle \frown tr \mid \\
&\qquad a \notin X \wedge \\
&\qquad (\langle X, a\rangle \frown tr \in \mathcal{RT}[\![P]\!] \wedge Q \textbf{ ref } X \vee \\
&\qquad \langle X, a\rangle \frown tr \in \mathcal{RT}[\![Q]\!] \wedge P \textbf{ ref } X)\}
\end{aligned}
$$

Refinement in the refusal traces model is then defined as follows:

$$P \sqsubseteq_{\mathcal{RT}} Q \Leftrightarrow \mathcal{RT}[\![P]\!] \supseteq \mathcal{RT}[\![Q]\!]$$

Both stable failures and refusal traces refinement assertions can be model checked using the FDR tool [8].

3. Patterns of Behaviour

We present a sub-language of our property specification language *PL*, denoted as *SPL*, for assisting developers to construct BPMN-based patterns of behaviour:

$$P \in SPL ::= P \sqcap P \mid P \sqcap\!\sqcap P \mid a \to P \mid End \quad \textbf{where } a \in Atom$$

$$Atom \quad ::= t \mid \texttt{available}\, t \mid \texttt{live} \quad \textbf{where } t \in Task$$

where the basic type *Task* represents the set of names that identify task states in a BPMN diagram, and the type *Atom* describes the performance or the availability of some task t. The behaviour $t \to P$ hence enacts task t and then behaves like P. The atomic term `live` describes the performance of any task state of the BPMN diagram in question. A graphical tool could be implemented to assist BPMN developers to construct specifications in this language.

The language is equipped with operators focusing on specifying nondeterministic concurrent systems that are suitable as process-based specifications. Specifically it contains a subset of standard CSP operators, that is nondeterministic choice ($\sqcap$) and prefix ($\to$), as well as a new *nondeterministic interleaving* operator ($\sqcap\!\sqcap$). Informally the process $P \sqcap\!\sqcap Q$ communicates events from both P and Q, but unlike CSP's interleaving, our operator chooses them nondeterministically. Here we present the step law governing the operator in the form of CSP's algebraic laws [26]: if $P = p \to P'$ and $Q = q \to Q'$ then

$$P \sqcap\!\sqcap Q = (p \to (P' \sqcap\!\sqcap Q)) \sqcap (q \to (P \sqcap\!\sqcap Q')) \qquad [\sqcap\!\sqcap\text{-step}]$$

Since the $\sqcap\!\sqcap$ operator is defined in terms of nondeterministic choice $\sqcap$ and prefix $\to$, we have the following unit law.

$$End \sqcap\!\sqcap Q = Q \qquad [\sqcap\!\sqcap\text{-End}]$$

$\sqcap\!\sqcap$ is both commutative and associative. It also distributes over nondeterministic choice $\sqcap$.

$$P \sqcap\!\sqcap Q = Q \sqcap\!\sqcap P \qquad [\sqcap\!\sqcap\text{-sym}]$$

$$(P \sqcap\!\sqcap Q) \sqcap\!\sqcap R = P \sqcap\!\sqcap (Q \sqcap\!\sqcap R) \qquad [\sqcap\!\sqcap\text{-assoc}]$$

$$P \sqcap\!\sqcap (Q \sqcap R) = (P \sqcap\!\sqcap Q) \sqcap (P \sqcap\!\sqcap R) \qquad [\sqcap\!\sqcap\text{-dist}]$$

This operator allows developers to construct patterns of behaviour representing parallel executions of task states without needing to know more refined detail such as timing information which may restrict possible orders of enactments of states.

Now we present the function *pattern*, which takes a pattern of behaviour described in *SPL* and returns the corresponding formula in *BTL**. Here *BTL** denotes *BTL* augmented with the atomic formula $*$, which has the empty set of refusal traces. We write $event(t)$ to denote an event associated with task t. For all $a \in Atom$, $t \in Task$ and $P, Q \in SPL$,

$$
\begin{aligned}
pattern(End) &= * \\
pattern(a \to P) &= atom(a) \wedge \bigcirc(pattern\, P) \\
pattern(P \sqcap Q) &= pattern(P) \vee pattern(Q) \\
pattern(P \sqcap\!\sqcap Q) &= pattern(npar(P, Q))
\end{aligned}
$$

where *npar* will be defined shortly and the function *atom* is defined as follows:

$$\begin{aligned} atom(\texttt{available}\, t) &= \texttt{available}\, (event(t)) \\ atom(\texttt{live}) &= \texttt{live} \\ atom(t) &= event(t) \end{aligned}$$

According to this translation *End* has an empty semantics.

To convert formulae in *BTL** back to *BTL*, we simply remove $*$ according to the following equivalences:

$$\begin{aligned} \phi \vee * &\equiv \phi \\ * \wedge \phi &\equiv \phi \\ \phi \wedge \bigcirc * &\equiv \phi \end{aligned}$$

Note that both the conjunctive and disjunctive operators are commutative.

We map each of the operators other than $\sqcap\!\sqcap$ directly into their corresponding temporal logic expression. Here we show that the semantics of the prefix operator $\rightarrow$ is preserved by the translation. First we give the semantic definition of $\rightarrow$ over *SPL* in the refusal traces model $\mathcal{RT}$ where *RT* denotes all (finite) refusal traces. For all $a \in \Sigma, X \in \mathbb{P}\,\Sigma$ and $tr \in RT$:

$$\begin{aligned} \mathcal{RT}_{SPL}[\![*]\!] &= \emptyset \\ \mathcal{RT}_{SPL}[\![t \rightarrow P]\!] &= \{\langle\rangle\} \cup \{\langle X, a\rangle \frown tr \mid a = event(t) \wedge a \notin X \wedge tr \in \mathcal{RT}_{SPL}[\![P]\!]\} \end{aligned}$$

Similarly we present Lowe's semantic definition [16] for the operators $\bigcirc$, $\wedge$ over *BTL* and the atomic formula a in $\mathcal{RT}$, where *IRT* denotes the set of all infinite refusal traces. For all $a \in \Sigma, X \in \mathbb{P}\,\Sigma$ and $tr \in RT \cup IRT$:

$$\begin{aligned} \mathcal{RT}_{BTL}[\![a]\!] &= \{\langle\rangle\} \cup \{\langle X, a\rangle \frown tr \mid a \notin X\} \\ \mathcal{RT}_{BTL}[\![\bigcirc\phi]\!] &= \{\langle\rangle, \langle\Sigma\rangle\} \cup \{\langle X, a\rangle \frown tr \mid a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![\phi]\!]\} \\ \mathcal{RT}_{BTL}[\![\psi \wedge \phi]\!] &= \mathcal{RT}_{BTL}[\![\psi]\!] \cap \mathcal{RT}_{BTL}[\![\phi]\!] \end{aligned}$$

According to our translation function $pattern\ (t \rightarrow P) = event(t) \wedge \bigcirc(pattern\, P)$, it is easy to show that

$$\begin{aligned} &\mathcal{RT}_{BTL}[\![event(t) \wedge \bigcirc(pattern\, P)]\!] \\ &= \mathcal{RT}_{BTL}[\![event(t)]\!] \cap \mathcal{RT}_{BTL}[\![\bigcirc(pattern\, P)]\!] && [\text{def of } \wedge] \\ &= \{\,\langle\rangle\,\} \cup \{\,\langle X, a\rangle \frown tr \mid a = event(t) \wedge a \notin X \wedge tr \in \mathcal{RT}_{BTL}\,\} \\ &\quad \cap \mathcal{RT}_{BTL}[\![\bigcirc(pattern\, P)]\!] && [\text{def of } event(t)] \\ &= \{\,\langle\rangle\,\} \cup \{\,\langle X, a\rangle \frown tr \mid a = event(t) \wedge a \notin X \wedge tr \in \mathcal{RT}_{BTL}\,\} && [\text{def of } \bigcirc] \\ &\quad \cap \{\,\langle X, a\rangle \frown tr \mid a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![pattern(P)]\!]\,\} \cup \{\,\langle\rangle, \langle\Sigma\rangle\,\} \\ &= \{\,\langle X, a\rangle \frown tr \mid a = event(t) \wedge a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![pattern(P)]\!]\,\} \\ &\quad \cup \{\,\langle\rangle\,\} && [\text{def of } \cap] \\ &\supset \mathcal{RT}_{SPL}[\![t \rightarrow P]\!] \end{aligned}$$

Since this sub-language is used to describe behaviour inside a property specification, we only need to concentrate on finite refusal traces of the same length. As a result subset inclusion suffices.

The nondeterministic interleaving operator $\sqcap\!\sqcap$ is sequentialised by the function *npar* before being mapped into its CSP equivalent. This function essentially implements the step law of $\sqcap\!\sqcap$ above via the function *initials* below and is defined as follows, where $P, Q \in SPL$.

$$\begin{aligned} npar(End, End) &= End \\ npar(End, Q) &= Q \\ npar(P, End) &= P \\ npar(P, Q) &= (\sqcap(a, X) : initials(P) \bullet a \rightarrow npar(X, Q)) \sqcap \\ &\quad (\sqcap(a, X) : initials(Q) \bullet a \rightarrow npar(X, P)) \end{aligned}$$

Similar to CSP, we write $\sqcap i : I \bullet P(i)$ to denote the nondeterministic choice of a set of indexed terms $P(i)$ where i ranges over I. The function *initials* takes a *SPL* model and returns a set of pairs, each pair containing a possible initial task enactment and the model after enacting that task. For example $initials(a \rightarrow A \sqcap b \rightarrow B)$ returns the set $\{(a, A), (b, B)\}$.

$$\begin{aligned} initials(P \sqcap Q) &= initials(P) \cup initials(Q) \\ initials(P \sqcap\!\sqcap Q) &= initials(npar(P, Q)) \\ initials(a \rightarrow P) &= \{\,(a, P)\,\} \\ initials(End) &= \emptyset \end{aligned}$$

Going back to the example in Fig. 4(b), we are now able to specify the pattern of behaviour $(a \to End) \sqcap\!\sqcap (b \to End)$ which states that tasks A and B are executed in parallel without needing to know their timing constraints. Here the *BTL* formula ϕ describes this pattern of behaviour,

$$\phi = (w.a \wedge \bigcirc w.b) \vee (w.b \wedge \bigcirc w.a)$$

where $w.A$ denotes work done in some task A. We apply Lowe's algorithm [16] to obtain the following semantically equivalent CSP process *Spec* of formula ϕ.

$$\begin{array}{l} Spec = \\ \quad \textbf{let} \\ \quad\quad Spec0 = w.b \to Spec2 \\ \quad\quad Spec1 = w.a \to Spec3 \\ \quad\quad Spec2 = w.a \to Spec4 \\ \quad\quad Spec3 = w.b \to Spec4 \\ \quad\quad Spec4 = Stop \sqcap (\sqcap x : \Sigma \bullet x \to Spec4) \\ \quad \textbf{in} \\ \quad\quad Spec0 \sqcap Spec1 \end{array}$$

This allows us to make the following kinds of refinement assertions under the refusal traces semantics, where the implementation process may represent the behaviour under the timed model and the untimed model respectively.

$$Spec \sqsubseteq_{\mathcal{RT}} w.a \to w.b \to Stop$$
$$Spec \sqsubseteq_{\mathcal{RT}} w.a \to Stop \;|||\; w.b \to Stop$$

Note that all expressions translated from *SPL* are characterised by atomic formulae over $\vee$, $\wedge$ and $\bigcirc$; in particular each *BTL*-translation of *SPL* may be captured by the following grammar E, where a is some atomic formula:

$$E ::= a\,(\wedge \bigcirc E)^* \mid (E \vee E)$$

Moreover, each *BTL*-translation of *SPL* may be translated into an equivalent *BTL* expression in *restricted disjunctive normal form* (*rDNF*). While an ordinary disjunctive normal form expression is one which consists of a disjunction of conjunctions of variables and negations of variables, a *rDNF* expression consists of a disjunction of conjunctions of atomic formulae and terms defined by $\bigcirc$ operators over atomic formulae. Here we provide a formal definition, where $nexts_i a$ is defined by i $\bigcirc$ operators over some formula a; full definition of function *nexts* is provided in Section 4.

Definition 3.1. *A BTL expression is in* ***restricted disjunctive normal form*** *(rDNF) if it has the form,*

$$(a^1_1 \wedge \bigcirc a^1_2 \wedge \cdots \wedge nexts_{k-1} a^1_k) \vee \cdots \vee (a^l_1 \wedge \bigcirc a^l_2 \wedge \cdots \wedge nexts_{j-1} a^l_j)$$

where each a^j_i is an atomic formula.

It is easy to see that any *BTL* expression generated by the grammar E may be translated into *rDNF* by applying the following two laws recursively.

$$\begin{array}{lll} \bigcirc(a \wedge b) & \equiv \bigcirc a \wedge \bigcirc b & [\bigcirc\text{-}\wedge\text{-dist}] \\ a \wedge (b \vee c) & \equiv (a \wedge b) \vee (a \wedge c) & [\wedge\text{-}\vee\text{-dist}] \end{array}$$

Here we show these two laws are valid under $\mathcal{RT}$ for all $a \in \Sigma, X \in \mathbb{P}\,\Sigma, tr \in RT \cup IRT$.

$$\begin{array}{ll} & \mathcal{RT}_{BTL}[\![\bigcirc\phi \wedge \bigcirc\psi]\!] \\ = & \mathcal{RT}_{BTL}[\![\bigcirc\phi]\!] \cap \mathcal{RT}_{BTL}[\![\bigcirc\psi]\!] & [\text{def of } \wedge] \\ = & \{\, \langle X, a\rangle \frown tr \mid a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![\phi]\!] \,\} & [\text{def of } \bigcirc] \\ & \quad \cup \{\, \langle\rangle, \langle\Sigma\rangle \,\} \cap \mathcal{RT}_{BTL}[\![\bigcirc\psi]\!] \\ = & \{\, \langle X, a\rangle \frown tr \mid a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![\phi]\!] \,\} & [\text{def of } \bigcirc] \\ & \quad \cap \{\, \langle X, a\rangle \frown tr \mid a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![\psi]\!] \,\} \cup \{\, \langle\rangle, \langle\Sigma\rangle \,\} \\ = & \{\, \langle X, a\rangle \frown tr \mid a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![\phi \wedge \psi]\!] \,\} & [\text{def of } \cap] \\ & \quad \cup \{\, \langle\rangle, \langle\Sigma\rangle \,\} \\ = & \mathcal{RT}_{BTL}[\![\bigcirc(\phi \wedge \psi)]\!] \end{array}$$

$$\begin{array}{ll} & \mathcal{RT}_{BTL}[\![a \wedge (b \vee c)]\!] \\ = & \mathcal{RT}_{BTL}[\![a]\!] \cap \mathcal{RT}_{BTL}[\![b \vee c]\!] & [\text{def of } \wedge] \\ = & \mathcal{RT}_{BTL}[\![a]\!] \cap \mathcal{RT}_{BTL}[\![b]\!] \cup \mathcal{RT}_{BTL}[\![c]\!] & [\text{def of } \vee] \\ = & (\mathcal{RT}_{BTL}[\![a]\!] \cap \mathcal{RT}_{BTL}[\![b]\!]) \cup (\mathcal{RT}_{BTL}[\![a]\!] \cap \mathcal{RT}_{BTL}[\![c]\!]) & [\cap\text{-}\cup\text{-dist}] \end{array}$$

$= \mathcal{RT}_{BTL}[\![a \wedge b]\!] \cup \mathcal{RT}_{BTL}[\![a \wedge c]\!]$ [def of $\wedge$]
$= \mathcal{RT}_{BTL}[\![(a \wedge b) \vee (a \wedge c)]\!]$ [def of $\vee$]

In next section we show how *BTL*-translation of *SPL* in *rDNF* may be used to assist the formalisation of some of the property patterns.

4. Property patterns

To assist the specification of behavioural properties in terms of the generalised property patterns, we define a property specification language *PL* by the following grammar:

$$x, y \in PL ::= \mathtt{Abs}(p, s) \mid \mathtt{Un}(p, s) \mid \mathtt{Ex}(p, n, s) \mid \mathtt{BEx}(p, b, s) \mid x \vee y \mid x \wedge y \quad \textbf{where } p \in SPL;\ n \in \mathbb{N};\ b \in BL;\ s \in SL$$

$$BL ::= \leq n \mid = n \mid \geq n \quad \textbf{where } n \in \mathbb{N}$$

$$SL ::= \mathtt{always} \mid \mathtt{before}\,(p, n) \mid \mathtt{after}\,p \mid \mathtt{between}\,p\,\mathtt{and}\,(q, n) \mid \mathtt{from}\,p\,\mathtt{until}\,(q, n) \quad \textbf{where } p \in SPL;\ n \in \mathbb{N}$$

where each term in *PL* represents a behavioural property with respect to the property pattern, each term specifies the behavioural constraints over some *bounded*, *nondeterministic* behaviours specified by the sub-language *SL*. Throughout this section we use the term *state* in the sense of a transition system of a CSP process describing a BPMN diagram: a graph showing the states it can go through and actions, each denoted by a single CSP event, that it takes to get from one to another. Algebraically, each transition between states is an application of a step law. We describe each term in *PL* briefly as follows:

- $\mathtt{Abs}(p, s)$ (Absence) states that the pattern of behaviour p must be refused throughout the scope s.
- $\mathtt{Un}(p, s)$ (Universality) states that the pattern of behaviour p must occur throughout the scope s.
- $\mathtt{Ex}(p, n, s)$ (Existence) states that the pattern of behaviour p must occur *at least once* during the scope s. In *LTL* one might model this property using the *eventually* operator; however as discussed earlier, it is not possible to model unbounded *eventually* specifications; therefore we restrict this pattern with a bound and instead state that p must occur *at least once* within the subsequent n states from the start of scope s.
- $\mathtt{BEx}(p, b, s)$ (Bounded Existence) states that the pattern of behaviour p must occur *a specified number of times*, defined by the bound b, throughout the scope s. A bound may either be *exactly* ($=n$), *at least* ($\geq n$) or *at most* ($\leq n$).

Each property may be specified within one of the five different types of scope, which are captured by our sub-language *SL*. Here we describe each one briefly.

- `always` (Global) states that the property in question must hold throughout all possible executions. For example $\mathtt{Abs}(a \vee b, \mathtt{always})$ states that both events a and b must be refused in all possible executions.
- $\mathtt{before}\,(p, n)$ (Before p) states that if there exists the pattern of behaviour p in the subsequent n states, the property in question must hold before p for all possible executions. For example $\mathtt{Un}(\mathtt{available}\,a, \mathtt{before}\,(b, n))$ states that a must not be refused before an occurrence of b in the subsequent n states.
- $\mathtt{after}\,p$ (After p) states that if there exists the pattern of behaviour p in any one of the subsequent states from the start of the execution, then the property in question must hold precisely after that state. For example $\mathtt{BEx}(a \vee b, \leq m, \mathtt{after}\,c)$ states that a sequence of at most m as and bs must occur after the occurrence of the event c.
- $\mathtt{between}\,p\,\mathtt{and}\,(q, n)$ (Between p and q) states that if there exists an occurrence of some pattern of behaviour p that is succeeded by some other pattern of behaviour q in n subsequent states after p, then the property in question must hold after p and before q.
- $\mathtt{from}\,p\,\mathtt{until}\,(q, n)$ (After p until q) states that if there exists an occurrence of some pattern of behaviour p then the property in question must hold after p or if there exists an occurrence of some pattern of behaviour q in the subsequent n states after p then the property in question must hold between p and q. Note that q does not ever have to occur.

Note that *PL*'s grammar does not include the patterns such as *Precedence* or *Response* [7]; we do not see this as a shortcoming, as these patterns, belonging the set of *order* patterns, may be expressed in terms of *generalised* existence patterns where each property is over a set of patterns of behaviours.

For convenience we define the function *next* such that $next(\phi, \psi)$ returns ψ composed with n next operators where n is the largest number of subsequent states about which ϕ makes an assertion. For example the furthest state of the expression $a \vee b$ is 1, and for both expressions $\bigcirc b$ and $a \wedge \bigcirc \mathtt{available}\,c$ it is 2. It is not difficult to calculate the number of states a pattern of behaviour spans, as *SPL* is characterised by $\vee$, $\wedge$ and $\bigcirc$ operators over atomic formulae in *BTL*. We define function $next(\phi, \psi) = nexts(states(\phi), \psi)$, where $states(\phi)$ returns one minus the furthest state the expression ϕ, translated from some pattern of behaviour in *SPL*, specifies. The expression $max(i, j)$ denotes the maximum of i and j.

$$nexts(0, \psi) = \psi$$
$$nexts(n, \psi) = \bigcirc (nexts(n - 1, \psi))$$

$$
\begin{aligned}
states(\phi \vee \psi) &= max(states(\phi), states(\psi)) \\
states(\phi \wedge \psi) &= max(states(\phi), states(\psi)) \\
states(\bigcirc \phi) &= 1 + states(\phi) \\
states(\phi) &= 1 \qquad \texttt{otherwise}
\end{aligned}
$$

The function *nexts* is defined such that $next(n, \phi)$ returns a composition of ϕ with n next operators. For presentation purposes we write $next_\phi(\psi)$ and $nexts_n(\psi)$ to denote $next(\phi, \psi)$ and $nexts(n, \psi)$ respectively, as shown in Definition 3.1. We write the predicate $single(\mu) \Leftrightarrow states(\phi) = 1$ for some *BTL* expression μ, and refer to all such expressions as *single state specifications*.

Also, we extend the grammar of *BTL*, denoted as BTL^δ, with the two derived temporal operators $\lhd\rhd$ and $\widetilde{\mathcal{U}}$ to express *bounded eventuality* and *bounded until*. Informally, for any positive integer n and BTL^δ formulae ϕ and ψ, the expression $\lhd\rhd_n \phi$ holds if and only if ϕ holds for some subsequent state up to the first n states. The expression $\psi \ \widetilde{\mathcal{U}}_n \ \phi$ holds if and only if ϕ holds for some state i up to the first n states, then ψ must hold every state up to $i-1$ states. Since $(\lhd\rhd_n \phi = \texttt{true} \ \widetilde{\mathcal{U}}_n \ \phi)$, it is sufficient to provide the semantics of $\widetilde{\mathcal{U}}$ as follows:

$$
\begin{aligned}
&P \models \psi \ \widetilde{\mathcal{U}}_n \ \phi \equiv \\
&\quad \forall tr : \mathcal{RT}[\![P]\!] \bullet \\
&\qquad (\exists i : 0 \,..\, n \bullet \forall j : 0 \,..\, (i-1) \bullet tr^i \in \mathcal{RT}_{BLT}[\![\phi]\!] \wedge tr^j \in \mathcal{RT}_{BLT}[\![\psi]\!])
\end{aligned}
$$

where $1 \leq n < \#tr$ and we write tr^i for refusal trace tr with the first i events and i refusals removed, for i ranging over the length of tr. We write $P \models \psi$ if every execution of process P satisfies the formula ψ. The following is the derivation of $\widetilde{\mathcal{U}}$ using operators in *BTL*:

$$
\psi \ \widetilde{\mathcal{U}}_n \ \phi = \left(\bigwedge_{i \in \{0..n-2\}} nexts_{i * states(\psi)}(\phi \vee \psi) \right) \wedge nexts_{(n-1) * states(\psi)} \phi \tag{3}
$$

For $\phi, \psi \in BTL^\delta$ and $n \in \mathbb{N}$, we define the function *derive* to convert an expression in BTL^δ back to *BTL* according to Eq. (3):

$$
\begin{aligned}
derive(\lhd\rhd_n \phi) &= derive(derive'(\texttt{true}, \phi, n)) \\
derive(\psi \ \widetilde{\mathcal{U}}_n \ \phi) &= derive(derive'(\psi, \phi, n)) \\
derive(\psi \Rightarrow \phi) &= derive(negate(\psi) \vee (\psi \wedge \phi)) \\
derive(\bigcirc \phi) &= \bigcirc(derive(\phi)) \\
derive(\phi \wedge \psi) &= derive(\phi) \wedge derive(\psi) \\
derive(\phi \wedge \psi) &= derive(\phi) \wedge derive(\psi) \\
derive(\phi \mathcal{R} \psi) &= derive(\phi) \ \mathcal{R} \ derive(\psi) \\
derive(\phi) &= \phi
\end{aligned}
$$

$$
\begin{aligned}
derive'(\psi, \phi, 1) &= \phi \\
derive'(\psi, \phi, n) &= \textbf{if } single(\psi) \textbf{ then } \phi \vee (\psi \wedge \bigcirc(derive'(\psi, \phi, n-1))) \\
&\qquad \textbf{else } \phi \vee (\psi \wedge next_\psi(derive'(\psi, \phi, n-1)))
\end{aligned}
$$

where we write $\phi \Rightarrow \psi$ as a shorthand for $\neg\phi \vee (\phi \wedge \psi)$ where ϕ and ψ are expressions in BTL^δ and ϕ does not include operators $\Box$ and $\mathcal{R}$. The function *negate* is defined such that $negate(\phi)$ negates the formula ϕ by distributing the negation operator over temporal operators except the always ($\Box$) and the release ($\mathcal{R}$) operators.

$$
\begin{aligned}
negate(\phi \vee \psi) &= negate(\phi) \wedge negate(\psi) \\
negate(\phi \wedge \psi) &= negate(\phi) \vee negate(\psi) \\
negate(\phi \Rightarrow \psi) &= \phi \wedge (negate(\phi) \vee negate(\psi)) \\
negate(\lhd\rhd_n \phi) &= (negate \circ derive)(\lhd\rhd_n \phi) \\
negate(\psi \ \widetilde{\mathcal{U}}_n \ \phi) &= (negate \circ derive)(\psi \ \widetilde{\mathcal{U}}_n \ \phi) \\
negate(\bigcirc \phi) &= \bigcirc(negate(\phi)) \\
negate(\texttt{available}\, a) &= \neg a \\
negate(\texttt{live}) &= \texttt{deadlock} \\
negate(\texttt{deadlock}) &= \texttt{live} \\
negate(a) &= \neg a \\
negate(\neg a) &= a
\end{aligned}
$$

Table 1
BTL^{δ} mapping of functions *absence*, *exist*, *universal*, *boundexist* and *bound*.

	Universal (*universal*)	Absence (*absence*)
`always`	$\Box p$	$\Box \neg p$
`before`(ν, n)	$(\Box\ \neg q) \vee (p\ \widetilde{\mathcal{U}}_n\ q)$ or $(\Box\ \neg q) \vee (p \wedge nexts_n(q))$	$(\Box\ \neg q) \vee (\neg p\ \widetilde{\mathcal{U}}_n\ q)$
`after`ν	$\Box(q \Rightarrow (next_q\ (\Box\ p)))$ or $\Box(q \Rightarrow (next_q\ p))$	$\Box(q \Rightarrow (next_q\ (\Box\ \neg p)))$
`between` ν `and` (υ, n)	$\Box(q \Rightarrow next_q\ (\lhd\!\rhd_n\ r \Rightarrow (p\ \widetilde{\mathcal{U}}_n\ r)))$ or $\Box(q \Rightarrow next_q\ (\lhd\!\rhd_n\ r \Rightarrow (p \wedge nexts_n r)))$	$\Box(q \Rightarrow (next_q(\lhd\!\rhd_n\ r \Rightarrow (\neg p\ \widetilde{\mathcal{U}}_n\ r))))$
`from` ν `until` (υ, n)	$\Box(q \Rightarrow (next_q\ (\Box\ p \vee (p\ \widetilde{\mathcal{U}}_n\ r))))$ or $\Box(q \Rightarrow (next_q\ (p \vee nexts_n r)))$	$\Box(q \Rightarrow (next_q\ (\Box \neg p \vee (\neg p\ \widetilde{\mathcal{U}}_n\ r))))$
	Bounded Existence (*boundexist*)	Existence (*exist*)
`always`	$bound(p, \texttt{false}, b)$	$\lhd\!\rhd_n\ p$
`before`(ν, n)	$\lhd\!\rhd_n\ q \Rightarrow \neg q\ \widetilde{\mathcal{U}}_{n-getbound(b)*states(p)}\ bound(p, q, b)$	$\lhd\!\rhd_n\ q \Rightarrow (\neg q\ \widetilde{\mathcal{U}}_m\ p)$
`after`ν	$\Box(q \Rightarrow next_q(bound(p, q, b)))$	$\Box(q \Rightarrow (next_q\ (\lhd\!\rhd_m\ p)))$
`between` ν `and` (υ, n)	$\Box(q \Rightarrow (next_q(\lhd\!\rhd_n\ r \Rightarrow (bound(p, r, b) \wedge bound(p, r, b)\ \mathcal{R}\ \neg r \wedge r\ \mathcal{R}\ \neg q))))$	$\Box(q \Rightarrow next_q\ (\lhd\!\rhd_n\ r \Rightarrow (\neg r\ \widetilde{\mathcal{U}}_m\ p)))$
`from` ν `until` (υ, n)	$\Box(q \Rightarrow (next_q(\neg r\ \widetilde{\mathcal{U}}_1\ bound(p, r \vee q, b))))$	$\Box(q \Rightarrow next_q\ (\neg r\ \widetilde{\mathcal{U}}_m\ p))$
	BTL^{δ} mappings of *bound*	
exactly n of p $(=n)$	$\bigwedge_{i \in \{0..n-1\}} (nexts_{i*states(p)}\ p) \wedge nexts_{n*states(p)}\ (q\ \mathcal{R}\ \neg p)$	
at least n of p $(\geq n)$	$\bigwedge_{i \in \{0..n-1\}} (nexts_{i*states(p)}\ p)$	
at most n of p $(\leq n)$	$nexts_{n*states(p)}\ (q\ \mathcal{R}\ \neg p)$	

This is sufficient, as the function is only applied to patterns of behaviour described in *SPL*, and we have shown in Section 3 that *SPL* can be completely characterised by $\wedge$ and $\bigcirc$ operators over atomic formulae in *BTL*. For example, the formula $\lhd\!\rhd_2\ (a \vee b)$ states that either task a or b must be performed at least once in the next two subsequent states; the corresponding formula in *BTL* is $(a \vee b) \vee (\texttt{true} \wedge \bigcirc(a \vee b))$.

For $n \in \mathbb{N}$, $\mu, \nu \in SPL$, $b \in Bound$, $s \in SL$, and $\sigma, \rho \in PL$, we define a translation function $tl = derive \circ tl'$ as a composition of two functions, taking a property specification in *PL* and returning its corresponding temporal logic expression in *BTL*.

$$
\begin{aligned}
tl'(\sigma \wedge \rho) &= tl'(\sigma) \wedge tl'(\rho) \\
tl'(\sigma \vee \rho) &= tl'(\sigma) \vee tl'(\rho) \\
tl'(\texttt{Abs}(\mu, s)) &= absence(\mu, s) \\
tl'(\texttt{Ex}(\mu, n, s)) &= exist(\mu, n, s) \\
tl'(\texttt{BEx}(\mu, b, s)) &= boundexist(\mu, b, s) \\
tl'(\texttt{Un}(\mu, s)) &= universal(\mu, s)
\end{aligned}
$$

Table 1 shows BTL^{δ} mappings of functions *absence*, *exist*, *universal* and *boundexist*. We assume p is the *BTL* expression of the pattern of behaviour μ, which we are interested in; and $q = pattern(\nu)$ and $r = pattern(\upsilon)$. As a shorthand we write $\neg p$ for some patterns of behaviour of p to represent the negation of p by distributing $\neg$ as described by the function *negate*. The rest of this section describes the formalisation of the patterns and their corresponding functions.

4.1. Universality

For all $\mu, \nu, \upsilon \in SPL$ and $n \in \mathbb{N}$, where $p = pattern(\mu)$, $q = pattern(\nu)$ and $r = pattern(\upsilon)$, we provide a description of our formalisation of the *universality* pattern. Note that it is not possible, in general to express a sequence of events being admissible recursively in *BLT*. For example one cannot express the behaviour of the CSP process $P = a \rightarrow b \rightarrow P$ in *BLT* since this will mean specifying every even state to have to perform the event b which is in general impossible in *LTL*. One would require a temporal logic equipped with a fix-point operator such as the modal mu-calculus [12] for such specification. Our definition of *universal* reflects this difference in expressiveness.

- The global occurrence μ is modelled trivially as $\Box p$ if μ is a single state specification, else it is just modelled as p.
- The occurrence of μ before some behaviour ν is modelled as $(\Box\ \neg q) \vee (p\ \widetilde{\mathcal{U}}_n\ q)$ if μ is a single state specification. This expression states that either the behaviour ν does not exist or μ occurs repeatedly until ν occurs in the subsequent n states where $n > states(p)$. Otherwise, this is modelled as $(\Box\ negate(q)) \vee (p \wedge nexts_n(q))$, which states that either the behaviour ν does not exist or one instance of μ occurs followed by the occurrence of ν in the subsequent n states since the start of μ where $n > states(p)$.
- The occurrence of μ after some behaviour ν is modelled as

$$\Box(q \Rightarrow (next_q\ (\Box\ p)))$$

if μ is a single state specification, which states that either the behaviour ν does not exist or μ must occur repeatedly throughout the whole execution after the occurrence of ν. Otherwise, this is modelled as $\Box(q \Rightarrow (next_q\ p))$ which states that if ν occurs then μ occurs in the next immediate states.

- The occurrence of μ between behaviours ν and υ is modelled as

$$\Box(q \Rightarrow next_q\ (\lhd\rhd_n\ r \Rightarrow (p\ \widetilde{\mathcal{U}}_n\ r)))$$

if μ is a single state specification. This expression states that if the behaviour ν occurs and there exists some behaviour υ in the n subsequent states after ν has occurred, then μ must occur repeatedly until ν occurs. Otherwise this pattern is modelled as $\Box(q \Rightarrow next_q\ (\lhd\rhd_n\ r \Rightarrow (p \wedge nexts_n r)))$ and this states that if the behaviour ν occurs and there exists some behaviour υ in the n subsequent states after ν has occurred, then one instance of μ occurs followed by the occurrence of ν in the subsequent n states since the start of μ where $n > states(p)$.

- The occurrence of μ after behaviours ν until υ is modelled as

$$\Box(q \Rightarrow (next_q\ (\Box\, p \vee (p\ \widetilde{\mathcal{U}}_n\ r))))$$

if μ is a single state specification. This expression states that if the behaviour ν occurs then either μ must occur repeatedly thereafter or μ must occur repeatedly until υ occurs in the subsequent n states after ν has occurred. Otherwise this pattern is modelled as $\Box(q \Rightarrow (next_q\ (p \vee nexts_n r)))$ and this states that if the behaviour ν occurs then either one instance of μ must occur immediately after ν and υ might occur in the subsequent n states after ν has occurred.

For example we could use the pattern "The occurrence of μ between some behaviours ν and υ" to describe the property that only task *A* or *B* can be performed between tasks *D* and *C*. This may be expressed in *PL* as follows,

$$\mathtt{Un}(a \rightarrow End \sqcap b \rightarrow End, \mathtt{between}\, c \rightarrow End\, \mathtt{and}\, (d \rightarrow End, 2))$$

and the following is the CSP specification translated from the corresponding *BTL* expression.

$$\begin{aligned}
Spec = \ & \textbf{let} \\
& Spec0 = Proceed(\{w.d\}, Spec0 \sqcap Spec1) \\
& Spec1 = w.d \rightarrow (Spec2 \sqcap Spec3 \sqcap Spec4 \sqcap Spec5) \\
& Spec2 = Proceed(\{w.c, w.d\}, Spec7 \sqcap Spec1) \\
& Spec3 = w.d \rightarrow (Spec2 \sqcap Spec3 \sqcap Spec5) \\
& Spec4 = w.c \rightarrow (Spec0 \sqcap Spec1) \\
& Spec5 = w.b \rightarrow Spec4 \\
& Spec7 = Proceed(\{w.c, w.d\}, Spec0 \sqcap Spec1) \\
\textbf{in}\ \ & Spec0 \sqcap Spec1
\end{aligned}$$

Here the parameterised process *Proceed* is defined as follows.

$$Proceed(X, P) = Stop \sqcap Skip \sqcap (\sqcap x : \Sigma \setminus X \bullet x \rightarrow P)$$

4.2. Absence

For all $\mu, \nu, \upsilon \in SPL$ and $n \in \mathbb{N}$, where $p = pattern(\mu)$, $q = pattern(\nu)$ and $r = pattern(\upsilon)$, we provide a description of our formalisation of the *absence* pattern.

- The absence of μ globally is modelled trivially as $\Box\neg p$
- The absence of μ before some behaviour ν is modelled as

$$(\Box\, \neg q) \vee (\neg p\ \widetilde{\mathcal{U}}_n\ q)$$

which states that either the behaviour ν does not exist or μ must be refused until ν occurs in the subsequent n states where $n > 1 + states(p)$

- The absence of μ after some behaviour ν is modelled as

$$\Box(q \Rightarrow (next_q\ (\Box\, \neg p)))$$

which states that either the behaviour ν does not exist or μ must be refused after the occurrence of ν. Note while it is not possible to model the unbounded eventuality of some events in the refusal traces model, it is possible on the other hand to model unbounded conditions like this one: "if some event *x* is ever to occur then some other event *y* must occur afterwards but without specifying *x* must be performed."

- The absence of μ between behaviours ν and υ is modelled as

$$\Box(q \Rightarrow (next_q(\lhd\rhd_n\ r \Rightarrow (negate(p)\ \widetilde{\mathcal{U}}_n\ r))))$$

which states that if the behaviour ν occurs and there exists some behaviour υ in the n subsequent states after ν has occurred, then μ must be refused until ν occurs.

- The absence of μ after behaviours ν until υ is modelled as

  $$\Box(q \Rightarrow (next_q\ (\Box\neg p \vee (\neg p\ \widetilde{\mathcal{U}}_n\ r))))$$

  which states that if the behaviour ν occurs then either μ must be refused thereafter or μ must be refused until υ occurs in the subsequent n states after ν has occurred.

For example we could use the pattern "The absence of μ after some behaviour ν" to describe the property neither task A nor B can be executed after task C has been performed; that it may be expressed in *PL* as follows,

$$\texttt{Abs}(a \rightarrow End \sqcap b \rightarrow End, \texttt{after}\ c \rightarrow End)$$

and the following is the CSP specification translated from the corresponding *BTL* expression $\Box((a \vee b) \Rightarrow \bigcirc(\Box\,\neg c))$.

$$\begin{aligned}
Spec = \ & \textbf{let} \\
& Spec0 = Proceed(\{\, w.c \,\}, Spec0 \sqcap Spec1) \\
& Spec1 = w.c \rightarrow (Spec2 \sqcap Spec3) \\
& Spec2 = Proceed(\{\, w.a, w.b, w.c \,\}, Spec2 \sqcap Spec3) \\
& Spec3 = w.c \rightarrow (Spec2 \sqcap Spec3) \\
& \textbf{in}\quad Spec0 \sqcap Spec1
\end{aligned}$$

Going back to our running example, we may use the absence pattern "the absence of μ between some behaviours ν and υ" to specify the requirement of the travel agent. The following *PL* expression specifies this requirement.

$$\texttt{Abs}(Cancel, \texttt{between}\ Book\_Seat \rightarrow End\ \texttt{and}(Send\_Invoice \rightarrow End, 2))$$

where the behaviour *Cancel* is defined as follows:

$$Cancel = Request\_Cancellation \rightarrow End \sqcap Reservation\_Timeout \rightarrow End$$

Here is the corresponding CSP process *Spec*.

$$\begin{aligned}
Obs = \ & \{\, w.Book_Seat, w.Request_Cancellation, \\
& \quad w.Reservation_Timeout, w.Send_Invoice \,\} \\
Spec = \ & Spec0 \sqcap Spec1 \\
Spec0 = \ & Proceed(\{\, w.Book_Seat \,\}, Spec0 \sqcap Spec1) \\
Spec1 = \ & w.Book_Seat \rightarrow (Spec2 \sqcap Spec3 \sqcap Spec4 \sqcap Spec5 \sqcap Spec6) \\
Spec2 = \ & Proceed(\{\, w.Book_Seat, w.Send_Invoice \,\}, Spec7 \sqcap Spec1) \\
Spec3 = \ & w.Send_Invoice \rightarrow (Spec0 \sqcap Spec1) \\
Spec4 = \ & w.Book_Seat \rightarrow (Spec2 \sqcap Spec4 \sqcap Spec8 \sqcap Spec9) \\
Spec5 = \ & Proceed(Obs \setminus \{\, w.Send_Invoice \,\}, Spec3) \\
Spec6 = \ & w.Book_Seat \rightarrow Spec3 \\
Spec7 = \ & Proceed(\{\, w.Book_Seat, w.Send_Invoice \,\}, Spec0 \sqcap Spec1) \\
Spec8 = \ & Proceed(Obs, Spec3) \\
Spec9 = \ & w.Book_Seat \rightarrow (Spec3)
\end{aligned}$$

Now it is possible to see whether the travel agent diagram meets this requirement by checking the following refusal traces refinement assertion using the FDR tool.

$$Spec \sqsubseteq_{\mathcal{RT}} Agent \setminus (\Sigma \setminus Obs)$$

4.3. Existence

For all $\mu, \nu, \upsilon \in SPL$ and $m, n \in \mathbb{N}$, where $p = pattern(\mu), q = pattern(\nu)$ and $r = pattern(\upsilon)$, we provide a description of our formalisation of the *existence* pattern. Similar to the universality pattern, our definition of the existence pattern reflects the impossibility of expressing unbounded eventually properties under the refusal traces model.

- The global existence μ is modelled trivially as $\diamond_n\ p$ which simply states that p will occur in one of the states up to the nth state.
- The existence of μ before some behaviour ν is modelled as

  $$\diamond_n\ q \Rightarrow (\neg q\ \widetilde{\mathcal{U}}_m\ p)$$

  which states that if ν occurs in one of the subsequent n states, then ν may only occur after μ occurs in one of the subsequent m states where n has to be larger than the sum of m plus the number of states μ specifies.

- The existence of μ after some behaviour ν is modelled as

  $$\Box(q \Rightarrow (next_q\ (\lhd\rhd_m\ p)))$$

  which states that if ν occurs at all then μ occurs in one of the subsequent m states after ν.
- The existence of μ between behaviours ν and υ is modelled as

  $$\Box(q \Rightarrow next_q\ (\lhd\rhd_n\ r \Rightarrow (negate(r)\ \widetilde{\mathcal{U}}_m\ p)))$$

  which states that if the behaviour ν occurs and there exists some behaviour υ in the n subsequent states after ν has occurred, then υ cannot occur until μ occurs in one of the subsequent m states after ν occurs. Here n must be larger than the sum of m plus the number of states μ specifies.
- The existence of μ after behaviours ν until υ is modelled as

  $$\Box(q \Rightarrow next_q\ (negate(r)\ \widetilde{\mathcal{U}}_m\ p))$$

  which states that if the behaviour ν occurs then μ must occur in one of the subsequent m states after ν occurs. While the behaviour υ may not occur before μ has occurred, υ could occur after μ has occurred.

For example we could use the pattern "The existence of μ after ν" to describe the property that task *A* followed by task *B* has to be executed within the two subsequent states after either task *C* or *D* has occurred. This may be expressed in *PL* as follows:

$$\mathtt{Ex}(a \rightarrow b \rightarrow End, 2, \mathtt{after}\ c \rightarrow End \sqcap d \rightarrow End)$$

and the following is the CSP specification translated from the corresponding *BTL* expression.

$$\begin{aligned}
Spec = \ & \textbf{let} \\
& Spec0 = Proceed(\{w.c, w.d\}, Spec0 \sqcap Spec1 \sqcap Spec2) \\
& Spec1 = w.d \rightarrow (Spec3 \sqcap Spec4 \sqcap Spec5 \sqcap Spec6) \\
& Spec2 = w.c \rightarrow (Spec3 \sqcap Spec4 \sqcap Spec5 \sqcap Spec6) \\
& Spec3 = w.a \rightarrow Spec7 \\
& Spec4 = Proceed(\{w.c, w.d\}, Spec3) \\
& Spec5 = w.d \rightarrow Spec3 \\
& Spec6 = w.c \rightarrow Spec3 \\
& Spec7 = w.b \rightarrow (Spec0 \sqcap Spec1 \sqcap Spec2) \\
& \textbf{in}\quad Spec0 \sqcap Spec1 \sqcap Spec2
\end{aligned}$$

4.4. Bounded existence

Unlike other property patterns, which only calculate the *maximum* number of states of the patterns of behaviour when specifying properties, for the *bounded existence* pattern, it is necessary to calculate *all possible numbers of states* of the pattern of behaviour for specifying properties. This is because to express a context over a bounded number of occurrences of some pattern of behaviour μ, we need to know *exactly* the number of states all occurrences of μ span. For example the maximum number of states for the pattern of behaviour $a \wedge (\bigcirc b \vee \bigcirc(c \wedge \bigcirc d)$ is three, while it also specifies a behaviour that only spans two states, namely $a \wedge \bigcirc b$; therefore the number of states covered by two occurrences of this pattern of behaviour may either be four, five or six. For all $p, q \in BTL$ and $bs \in \mathrm{seq}_1\ BTL$ and $n \in \mathbb{N}$, assuming p, q and all elements in bs are in *rDNF*, we define the function *combine* such that $combine(p, n)$ returns a set of patterns of behaviour, each defining a disjunction of n possible occurrences of behaviour specified by p such that each disjunct covers an equal number of states.

$$\begin{aligned}
combine(p, 1) &= disjunct(p) \\
combine(p, n) &= \{join(y) \mid x \in disjunct(p), y \in repeat(p, \langle x \rangle, n-1)\}
\end{aligned}$$

$$\begin{aligned}
joins(\langle p \rangle) &= p \\
joins(\langle p \rangle \frown bs) &= p \wedge next_p(joins(bs))
\end{aligned}$$

$$\begin{aligned}
repeat(p, bs, 1) &= \{fs \frown \langle x \rangle \mid x \in disjunct(p)\} \\
repeat(p, bs, n) &= \left\{ \bigcup (repeat(p, bs \frown \langle p \rangle, n-1)) \mid x \in disjunct(p) \right\}
\end{aligned}$$

$$\begin{aligned}
disjunct(p \vee q) &= disjunct(p) \cup disjunct(q) \\
disjunct(p) &= \{p\}
\end{aligned}$$

For example, the two occurrences of the behaviour $a \wedge (\bigcirc b \vee \bigcirc(c \wedge \bigcirc d)$ would give the following set

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

Having obtained a set of patterns of behaviour, we now describe the function *bound* such that $bound(p, q, b)$ returns the corresponding expression in BTL^{δ} stating a bounded existence of behaviour p with no scope.

- The expression to model exactly n ($=n$) occurrences of behaviour p may be written as $\bigwedge_{i\in\{0..n-1\}}(nexts_{i*states(p)}\ p) \wedge nexts_{n*states(p)}\ (q\ \mathcal{R}\ \neg p)$. Note that since it is not possible to model unbounded eventually, and hence also unbounded until, we restrict this pattern so that all the instances of p occur consecutively. This is not a problem as it is always possible to conceal all the other behaviours within the diagram in question via the CSP hiding operator. The condition $(q\ \mathcal{R}\ \neg p)$ is to ensure that p may not occur until some other behaviour q occurs, signifying the start of the pattern's scope. It is `false` if the scope is global.
- The expression to model at least n ($\geq n$) occurrences of behaviour p may be written as $\bigwedge_{i\in\{0..n-1\}}(nexts_{i*states(p)}\ p)$. Since the bound is greater than or equal, the condition $(q\ \mathcal{R}\ \neg p)$ is not required.
- The expression to model at most n ($\leq n$) occurrences of behaviour p may be written as $nexts_{n*states(p)}\ (q\ \mathcal{R}\ \neg p)$. This expression states that each of the n instances of p may or may not occur.

For all $\mu, \nu, \upsilon \in SPL$, $b \in Bound$, $n \in \mathbb{N}_1$ and $s \in SL$, where $p = pattern(\mu)$, $q = pattern(\nu)$ and $r = pattern(\upsilon)$, we provide a description of our formalisation of the *bounded existence* pattern. We write $getbound(b)$ for some bound b to denote the number part of the value, for example $getbound(\leq 1) = 1$. Similar to the universality and existence patterns, our definition of the bounded existence pattern reflects the impossibility of expressing unbounded eventually under the refusal traces model.

- The global existence μ with bound b is modelled trivially as $bound(p, \texttt{false}, b)$.
- The existence of μ with bound b before some behaviour ν is modelled as

  $$\diamond_n\ q \Rightarrow \neg q\ \widetilde{\mathcal{U}}_{n-getbound(b)*states(p)}\ bound(p, q, b)$$

  which states that if ν occurs in one of the subsequent n states, then ν may only occur after the bounded number of μ occurs within the subsequent $n - getbound(b) * states(p)$ states.
- The existence of μ with bound b after some behaviour ν is modelled as $\Box(q \Rightarrow next_q(bound(p, q, b)))$, which states that if ν occurs at all then the bounded number of μ occurs immediately after ν.
- The existence of μ with bound b between behaviour ν and υ is modelled as

  $$\Box(q \Rightarrow (next_q \diamond_n r \Rightarrow (bound(p, r, b) \wedge bound(p, r, b)\ \mathcal{R}\ \neg r \wedge r\ \mathcal{R}\ \neg q)))$$

  which states that if the behaviour ν occurs and there exists some behaviour υ in the n subsequent states after ν has occurred, then υ cannot occur until a bounded number of instances of μ occur after ν occurs. Here n must be strictly larger than $getbound(b) * states(p)$, and we restrict this pattern so that ν may only occur again after υ has occurred.
- The existence of μ after behaviour ν until υ is modelled as

  $$\Box(q \Rightarrow (next_q \neg r\ \widetilde{\mathcal{U}}_1\ bound(p, r \vee q, b)))$$

  which states that if the behaviour ν occurs then either the bounded number of instances of μ must occur immediately after ν occurs. While the behaviour υ may not occur before the instances of μ have occurred, υ could occur after.

For example we could use the pattern "The bounded existence of μ after ν" to describe the property that either task *A* or *C* has to occur followed by either one of them again after task *B* has occurred. This may be expressed in *PL* as follows:

$$\texttt{BEx}(a \rightarrow End \sqcap c \rightarrow End, = 2, \texttt{after}\ b \rightarrow End)$$

and the following is the CSP specification translated from the corresponding *BTL* expression.

$$\begin{aligned} Spec = \textbf{let}\ & Spec0 = Proceed(\{\, w.b \,\}, Spec0 \sqcap Spec1)\\ & Spec1 = w.b \rightarrow (Spec2 \sqcap Spec3)\\ & Spec2 = w.c \rightarrow (Spec4 \sqcap Spec5)\\ & Spec3 = w.a \rightarrow (Spec4 \sqcap Spec5)\\ & Spec4 = w.c \rightarrow (Spec7 \sqcap Spec6)\\ & Spec5 = w.a \rightarrow (Spec7 \sqcap Spec6)\\ & Spec6 = w.b \rightarrow (Spec2 \sqcap Spec3)\\ & Spec7 = Proceed(\{\, w.a, w.b, w.c \,\}, Spec7 \sqcap Spec6)\\ \textbf{in}\ \ & Spec0 \sqcap Spec1 \end{aligned}$$

5. Airline ticket reservation

We now consider the whole airline ticket reservation business process shown in Fig. 2. This example has been adopted from the WSCI specification document [30]. The business process describes the collaboration of three participants: a traveller, a travel agent, and an airline reservation system, which are specified by BPMN pools *Traveller*, *Agent* and *Airline* respectively. In Section 1.1, we have already described the workflow of the travel agent and subsequently verified its BPMN specification against one of its behavioural requirements. We first describe the workflows of the remaining two participants and in Section 5.3 we investigate behavioural properties concerning individual participants as well as their collaboration.

5.1. Traveller

The traveller can order a trip by setting up an itinerary for airline tickets; thereafter she can reserve the seats and subsequently proceed with the booking, after which the travel agent and the airline will send her the invoice and the tickets respectively.

Specifically, the traveller may choose her travel plan (from a catalogue, independently), and submit her choice to her travel agent via her local web service (e.g. web form) (*Order_Trip*). For various reasons the traveller may choose to change her itinerary (*Change_Itinerary*); she may also decide not to take the trip, and in which case she may cancel her order (*Cancel_Itinerary*). In case she decides to accept the proposed itinerary, she may proceed to reserve this itinerary (*Send_Confirmation*) and provide her credit card information to the travel agent. After this the traveller may either confirm her tickets (*Book_Tickets*) or cancel them (*Cancel_Tickets*); if she chooses to cancel her tickets, she will have to wait for her cancellation to be processed (*Accept_Cancellation*). Also if she takes too long to confirm her tickets (the period in between confirming her itinerary and booking her tickets), a timeout will occur; the tickets will then be released from the traveller and she will receive a cancellation notification (*Accept_Cancellation*). From the traveller's point of view, the time restriction would normally be enforced by the travel agent, therefore timeout is modelled as a message exception flow attached to the task *Book_Tickets*. After the traveller confirmed her tickets, she will receive the invoice from the travel agent (*Receive_Invoice*) and tickets from the airline (*Receive_Tickets*). Note that from the point of view of the traveller's workflow, the order in which she receives the invoice and tickets is not important.

5.2. Airline reservation system

The airline reservation system receives a request to check for seating availability, it checks for availability, it reserves the seats, then it completes the order and delivers the tickets.

Specifically, upon receiving an order, the airline verifies if the order is correct (*Verify_Orders*). The airline may also receive changes to the order, in which case, the reservation system needs to verify the changes (*Verify_Seats*) accordingly. The number of times seats may be checked per session may be assumed to be determined by the particular policy of the airline. The reservation system then waits for the request message and proceeds with reserving the seats (*Reserve_Reservation*). At this point the reservation system may receive a reservation cancellation notification, in which case the reservation system "unreserves" the seats (*Receive_Cancellation*) and sends out a cancellation notification (*Notify_Cancellation*), or it may receive the order to perform the final booking (*Perform_Booking*). Note the session may timeout if the booking is not performed within the time limit after reservations have been made; this is modelled by the timer exception flow (*Itimer*) attached to the task *Perform_Booking*. If a timeout occurs the booking is then cancelled automatically and a timeout notification will be sent out (*Notify_Timeout*). A more general cancellation notification will also be sent out (*Notify_Cancellation*). Otherwise when the booking is complete, the tickets for the corresponding seats will be sent out (*Send_Tickets*).

5.3. Requirements

We would like to check the following two general requirements (G1 and G2) and four specific requirements (S1, S2, S3, S4 and S5) of the collaboration.

G1 All participants must be deadlock free.
G2 The collaboration must be deadlock free.
S1 Once the traveller has confirmed her order, she receive either both tickets and invoice or a notification of cancellation.
S2 It is not possible for the travel agent to request cancellation and for the traveller to receive tickets.
S3 If the traveller issues a cancellation, she must receive a notification.
S4 If the airline reservation system issues a timeout during the booking process, the traveller must receive a notification of cancellation.
S5 The traveller must be able to change or cancel the itinerary she has ordered.

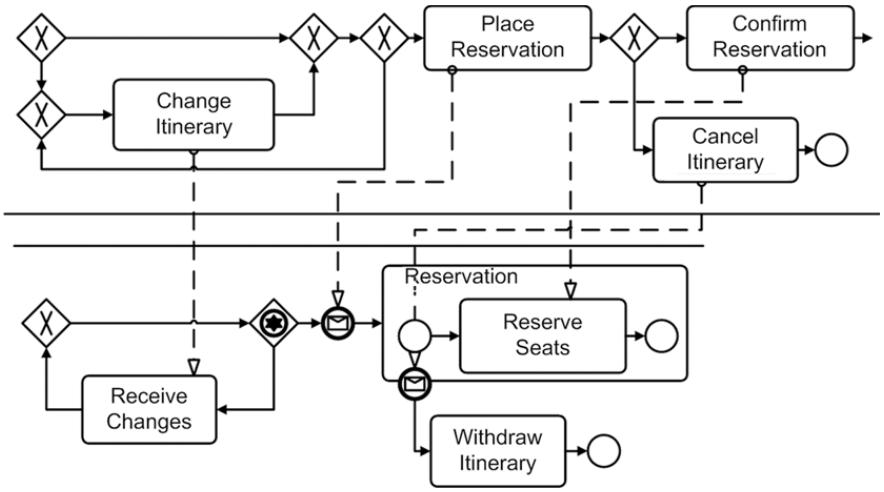

Fig. 7. Correcting the reservation phase.

5.3.1. Requirement G1

Requirement G1 checks whether the workflow of an individual participant is deadlock free, that is, we are interested to know if any possible order of execution could lead to a situation where the business transaction has not been completed and yet no more progress can be made. In our process semantics [33], the property of deadlock freedom may be asserted by the following predicate,

$$\forall i : \{Travel, Agent, Airline\} \bullet DF(\alpha P(i)) \sqsubseteq_{\mathcal{F}} P(i) \tag{4}$$

where $P(i)$ denotes the process semantics of participant i, and $DF(E)$ is a characteristic deadlock free process defined recursively as follows.

$$DF(E) = \sqcap e : E \bullet e \rightarrow DF(E) \sqcap Skip$$

It is sufficient to reason about deadlock freedom under $\mathcal{F}$; specifically, Property (4) may be split into individual refinement assertions and verified using the FDR tool [8]. Note that both processes $P(Travel)$, $P(Agent)$ and $P(Airline)$ and the deadlock freedom assertions can be automatically constructed using our implementation of the process semantics [31].

Requirement G1 is true.

5.3.2. Requirement G2

Requirement G2 checks whether the collaboration is also deadlock free. Since we know individual participants are themselves deadlock free, it is sufficient to show that they are *compatible*. While we need a formal notion of compatibility, we relegate this discussion to Section 6, in which we provide its formal definition and the results required here.

We write $compatible(P, Q)$ if BPMN processes P and Q are compatible. Since this binary relation *compatible* is symmetric by Definition 6.1 and due to Theorem 6.5, it suffices to prove the following predicate expression:

$$compatible(Travel, Agent) \wedge compatible(Travel, Airline) \wedge compatible(Agent, Airline)$$

By using the mechanical procedure devised for responsiveness, we initially have found that $compatible(Travel, Agent)$ does not hold. The FDR tool returns the following counterexample in which the CSP process $P(Travel) \llbracket \alpha P(Travel) \mid \alpha P(Agent) \rrbracket P(Agent)$ deadlocks after the following trace.

$$\langle w.Order\_Trip, w.Cancel\_Itinerary, w.Receive\_Order, w.Reserve\_Seat \rangle$$

The problem is that while the traveller decides to cancel her itinerary, her intention could not be properly communicated to the travel agent. Specifically after the reservation process has completed, it is still possible for the traveller to cancel her itinerary (*Cancel_Itinerary*) by sending a message to the travel agent's reservation process (*Reserve_Seat*), which causes the deadlock. One possible set of changes to the collaboration between the traveller and the travel agent is shown in Fig. 7. We describe the changes as follows:

- Traveller — First insert two new tasks *Place_Reservation* and *Confirm_Reservation* before task *Send_Confirmation*, then move the original XOR gateway between *Cancel_Itinerary* and *Send_Confirmation* to between *Cancel_Itinerary* and *Confirm_Reservation*.
- Agent — Change task *Reserve_Seats* into subprocess *Reservation* embedding task *Reserve_Seats*, whereby the message exception flow originally attached to task *Reserve_Seats* is now attached to subprocess *Reservation*. Place an intermediate message event in front of the subprocess.
- Interactions — Connect the following message flows:

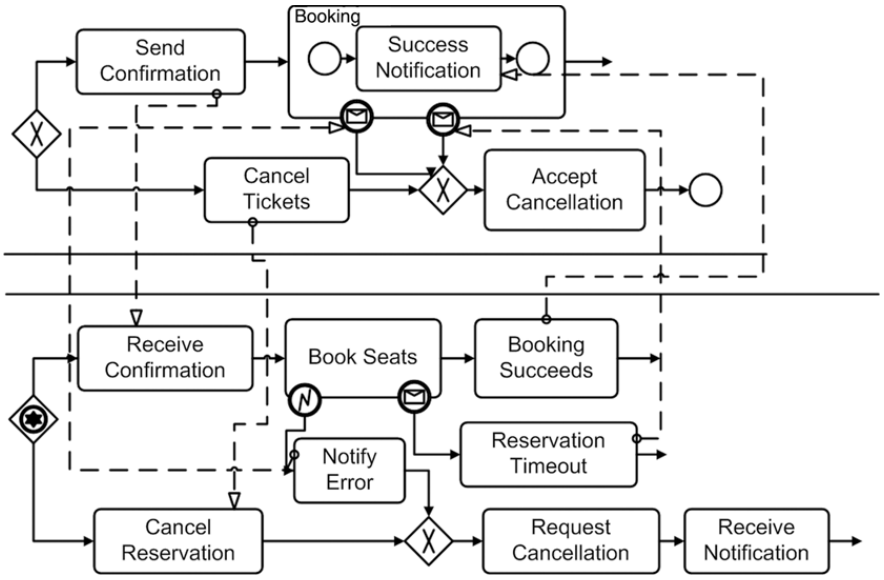

Fig. 8. Correcting the booking phase.

- from *Place_Reservation* to the intermediate message event
- from *Confirm_Reservation* to *Reserve_Seats*
- from *Cancel_Itinerary* to message event attached to subprocess *Reservation*

After the above changes, we have discovered another inconsistency in which the interaction between the workflows of traveller and travel agent deadlock after the following trace.

$$\langle w.Order\_Trip, w.Place\_Reservation, w.Receive\_Order, w.Confirm\_Reservation, w.Send\_Confirmation, w.Reserve\_Seats, w.Book\_Tickets, w.Receive\_Confirm, w.Reservation\_Timeout, w.Receive\_Tickets \rangle$$

The problem is that while a timeout (*Reservation_Timeout*) occurs during task *Book_Seats*, the travel agent is unable to communicate this information to the traveller, hence the traveller assumes the booking is successful; the fact that *w.Receive_Tickets* is performed is irrelevant as task *Receive_Tickets* does not interact with the travel agent. Assuming the traveller is using a web site to carry out the booking, this misinformation could lead to a HTTP 404 error response, which gives no information about the current transaction!

Fig. 8 shows one possible set of changes to the workflows of the traveller and travel agent; the description of the changes is similar to those described for Fig. 7 and thus has been omitted.

Fig. 9 shows a modified BPMN diagram describing the airline ticket reservation business process for which both Requirements G1 and G2 hold. The modification is based on further investigation to establish *compatible*(*Travel*, *Airline*) and *compatible*(*Agent*, *Airline*). We have omitted the textual description of the modification. Note that the changes may not reflect those shown in Fig. 7 and 8 as the modification in Fig. 9 has taken into account compatibilities between *Agent* and *Airline*, and *Traveller* and *Airline*.

Note that the inconsistencies uncovered in this section are not meant to imply the original example in the WSCI specification document [30] to be incorrect. While we provide an automatic translation from BPMN models to CSP processes, the construction of the high-level BPMN model from an informal description of a business process is a manual procedure and can therefore be prone to human error. Formal analyses such as those described in this section therefore also aim to help producing consistent BPMN models.

This concludes the investigation into the general requirements of the airline ticket reservation business process. We consider the specific requirements in the next section.

5.3.3. Requirement S1

Requirement S1 states that if the traveller has confirmed her order, she receives either both tickets and invoice or a notification of cancellation. Since we could abstract the process semantics by hiding irrelevant events as described in Section 1.5, we use the pattern "the existence of μ after ν" to model this requirement. This may be expressed in *PL* as follows:

$$\mathtt{Ex}(Ending, 1, \mathtt{after}Send\_Confirmation \rightarrow End)$$

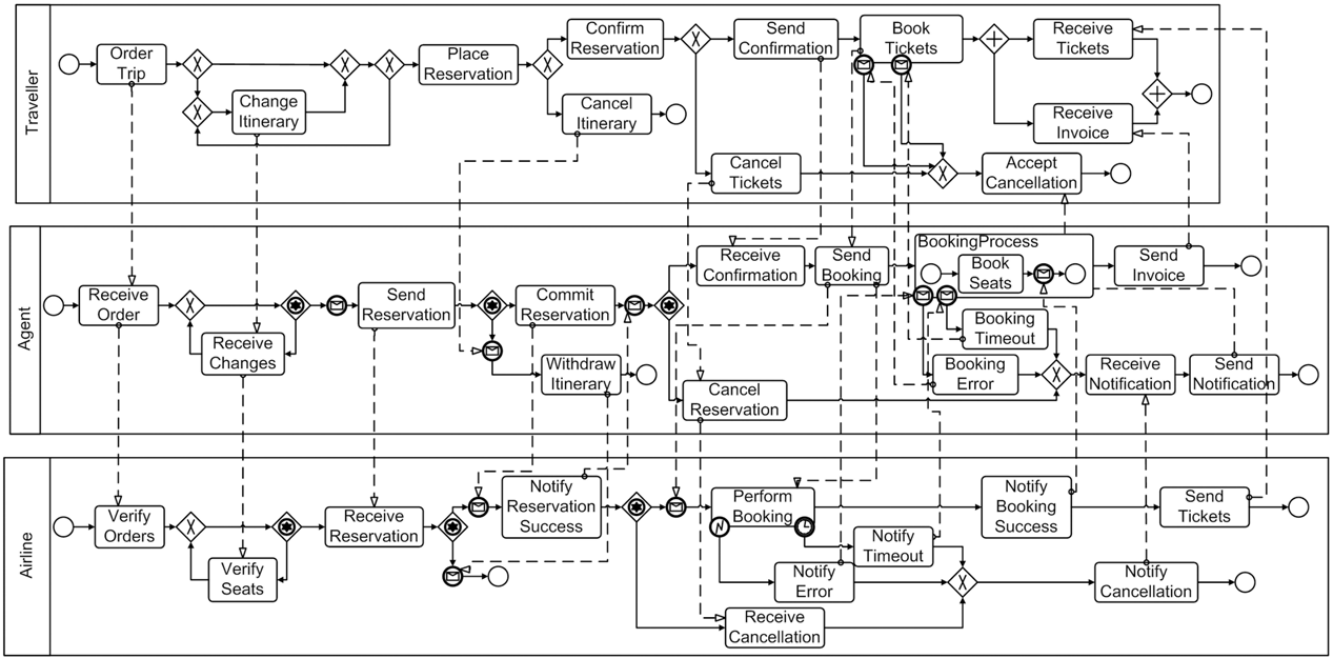

Fig. 9. Corrected airline ticket reservation.

where the behaviour *Ending* is defined as follows:

$$(Receive\_Invoice \rightarrow End \sqcap\sqcap Receive\_Tickets \rightarrow End) \sqcap Accept\_Cancellation \rightarrow End$$

Here is the corresponding CSP process *Spec*.

$$Spec = Spec0 \sqcap Spec1$$
$$Spec0 = Proceed(\{w.Send\_Confirmation\}, Spec0 \sqcap Spec1)$$
$$Spec1 = w.Send\_Confirmation \rightarrow Spec2 \sqcap Spec3 \sqcap Spec4$$
$$Spec2 = w.Receive\_Tickets \rightarrow Spec5$$
$$Spec3 = w.Receive\_Invoice \rightarrow Spec6$$
$$Spec4 = w.Accept\_Cancellation \rightarrow Spec0 \sqcap Spec1$$
$$Spec5 = w.Receive\_Invoice \rightarrow Spec0 \sqcap Spec1$$
$$Spec6 = w.Receive\_Tickets \rightarrow Spec0 \sqcap Spec1$$

We verify that the business collaboration diagram satisfies this property by checking the following refusal traces refinement assertion using the FDR tool,

$$Spec \sqsubseteq_{\mathcal{RT}} Collab \setminus (\Sigma \setminus A)$$

where A is the set of events used in this property specification, and M is the set of events denoting the message flows used in this model. *Collab* is the process semantics of the business collaboration and is defined as follows,

$$A = \{w.Send\_Confirmation, w.Accept\_Cancellation,$$
$$w.Receive\_Tickets, w.Receive\_Invoice\}$$
$$Collab = (\ \|\ i : \{Traveller, Agent, Airline\} \bullet \alpha P(i) \circ P(i)) \setminus M$$

Note that process *Collab* is automatically derivable using our implementation of the process semantics [31]. In addition, set A can be mechanically calculated by finding the intersection between the alphabet of *Collab* and *Spec*.

Requirement S1 is true.

5.3.4. Requirement S2

Requirement S2 states that it is not possible for the travel agent to request cancellation and for the traveller to receive tickets. We use the pattern "the absence of μ after ν" to model this requirement. This may be expressed in *PL* as follows:

$$\texttt{Abs}(Receive\_Tickets \rightarrow End, \texttt{after}Cancel)$$

where the behaviour *Cancel* is defined as follows:

$$Cancel\_Itinerary \rightarrow End \sqcap Cancel\_Tickets \rightarrow End$$

Here is the corresponding CSP process *Spec*.

$Spec = Spec0 \sqcap Spec1 \sqcap Spec2$
$Spec0 = Proceed(\{w.Cancel\_Itinerary, w.Cancel\_Tickets\}, Spec0 \sqcap Spec1 \sqcap Spec2)$
$Spec1 = w.Cancel\_Tickets \rightarrow Spec3 \sqcap Spec1 \sqcap Spec2$
$Spec2 = w.Cancel\_Itinerary \rightarrow Spec3 \sqcap Spec1 \sqcap Spec2$
$Spec3 = Proceed(A, Spec3 \sqcap Spec1 \sqcap Spec2)$

where $A = \{w.Cancel\_Itinerary, w.Cancel\_Tickets, w.Receive\_Tickets\}$.

We verify that the business collaboration diagram satisfies this property by checking the following refusal traces refinement assertion using the FDR tool.

$$Spec \sqsubseteq_{\mathcal{RT}} Collab \setminus (\Sigma \setminus A)$$

Requirement S2 is true.

5.3.5. Requirement S3

Requirement S3 states that if the traveller issues a cancellation, she must receive a notification. We use the pattern "the existence of μ after ν" to model this requirement. This may be expressed in *PL* as follows:

$$\mathtt{Ex}(Accept\_Cancellation \rightarrow End, 1, \mathtt{after}Cancel\_Tickets \rightarrow End)$$

Here is the corresponding CSP process *Spec*.

$Spec = Spec0 \sqcap Spec1$
$Spec0 = Proceed(\{w.Cancel\_Tickets\}, Spec0 \sqcap Spec1)$
$Spec1 = w.Cancel\_Tickets \rightarrow Spec2$
$Spec2 = w.Accept\_Cancellation \rightarrow Spec0 \sqcap Spec1$

We verify that the business collaboration diagram satisfies this property by checking the following refusal traces refinement assertion using the FDR tool.

$$Spec \sqsubseteq_{\mathcal{RT}} Collab \setminus (\Sigma \setminus \{w.Cancel\_Tickets, w.Accept\_Cancellation\})$$

Requirement S3 is true.

5.3.6. Requirement S4

Requirement S4 states that if airline reservation system issues a time out during the booking process, the traveller must receive a notification of cancellation. Again we use the pattern "the existence of μ after ν" to model this requirement. This may be expressed in *PL* as follows:

$$\mathtt{Ex}(Accept\_Cancellation \rightarrow End, 1, \mathtt{after}Notify\_Timeout \rightarrow End)$$

Here is the corresponding CSP process *Spec*.

$Spec = Spec0 \sqcap Spec1$
$Spec0 = Proceed(\{w.Notify\_Timeout\}, Spec0 \sqcap Spec1)$
$Spec1 = w.Notify\_Timeout \rightarrow Spec2$
$Spec2 = w.Accept\_Cancellation \rightarrow Spec0 \sqcap Spec1$

We verify that the business collaboration diagram satisfies this property by checking the following refusal traces refinement assertion using the FDR tool.

$$Spec \sqsubseteq_{\mathcal{RT}} Collab \setminus (\Sigma \setminus \{w.Notify\_Timeout, w.Accept\_Cancellation\})$$

Requirement S4 is true.

5.3.7. Requirement S5

Requirement S5 states that the traveller must be able to change or cancel the itinerary she has ordered. For this requirement we may use the pattern "the bounded existence of μ after ν". This may be expressed in *PL* as follows,

$$\mathtt{BEx}(p, \geq 1, \mathtt{after}Order\_Trip \rightarrow End)$$

where p is the following *SPL* term.

$$(\mathtt{available}\ Change\_Itinerary) \rightarrow End \sqcap (\mathtt{available}\ Cancel\_Itinerary) \rightarrow End$$

Here is the corresponding CSP process *Spec*.

$$
\begin{aligned}
&Spec = Spec0 \sqcap Spec1\\
&Spec0 = Proceed(\{w.Order\_Trip\}, Spec0 \sqcap Spec1)\\
&Spec1 = w.Order\_Trip \rightarrow (Spec2 \sqcap Spec3 \sqcap Spec4 \sqcap Spec5)\\
&Spec2 = Spec0 \,\Box\, w.Change\_Itinerary \rightarrow Spec\\
&Spec3 = Spec0 \,\Box\, w.Cancel\_Itinerary \rightarrow Spec\\
&Spec4 = (w.Order\_Trip \rightarrow (Spec2 \sqcap Spec3 \sqcap Spec4 \sqcap Spec5)) \,\Box\\
&\qquad (w.Change\_Itinerary \rightarrow Spec)\\
&Spec5 = (w.Order\_Trip \rightarrow (Spec2 \sqcap Spec3 \sqcap Spec4 \sqcap Spec5)) \,\Box\\
&\qquad (w.Cancel\_Itinerary \rightarrow Spec)
\end{aligned}
$$

We verify that the business collaboration diagram satisfies this property by checking the following refusal traces refinement assertion using the FDR tool,

$$Spec \sqsubseteq_{\mathcal{RT}} Collab \setminus (\Sigma \setminus A)$$

where $A = \{w.Order\_Trip, w.Change\_Itinerary, w.Cancel\_Itinerary\}$.

Requirement S5 is not satisfied. A counterexample shows that it is possible for *Collab* to perform *w.Order_Trip* and terminate without offering the events *w.Change_Itinerary* or *w.Cancel_Itinerary*. Specifically, while the traveller may change her itinerary after ordering, the decision of whether to change the itinerary is internal to her business process. Similarly the traveller places her reservation, and the decision of whether to cancel the itinerary is also internal to her business process.

6. Compatibility

In this section we report some recent results in our study of compatibility between business processes. We provide the binary relation *compatible* based on Reed et al.'s notion of responsiveness [25], which supersedes the definition provided in our earlier work [35].

Definition 6.1 (*Compatibility*)**.** *For any BPMN processes P and Q, which interact via the set of message flows M, they are compatible, denoted by the predicate compatible(P, Q), iff*

$$
\begin{aligned}
&DF \sqsubseteq_{\mathcal{F}} P' \wedge\\
&DF \sqsubseteq_{\mathcal{F}} Q' \wedge\\
&(P' \setminus H \; RespondsTo \; Q' \setminus H \vee Q' \setminus H \; RespondsTo \; P' \setminus H)
\end{aligned}
$$

where $H = \Sigma \setminus M'$; M' is the set of events denoting message flows in M, and P' and Q' are the process semantics of P and Q respectively.

Formally Reed et al. [25] provided a binary relation *RespondsTo* over CSP processes under the stable failures model as follows:

Definition 6.2 (*Responsiveness*)**.** *For any processes P and Q where there exists a set J of shared events, Q RespondsTo P iff for all traces $s \in \text{seq}(\alpha P \cup \alpha Q)$ and event sets X, such that $u = s \upharpoonright \alpha P$ and $t = s \upharpoonright \alpha Q$:*

$$
\begin{aligned}
&(u, X) \in failures(P) \wedge (initials(P/u) \cap J^{\checkmark}) \setminus X \neq \emptyset \Rightarrow\\
&\qquad\qquad (t, (initials(P/u) \cap J^{\checkmark}) \setminus X) \notin failures(Q)
\end{aligned}
$$

where $initials(P/s)$ is the set of possible events for P after trace s and $J^{\checkmark} = J \cup \{\checkmark\}$.

To paraphrase this: Given a component, modelled by some process P, which is itself deadlock free, and placed in parallel with another component, modelled by some process Q, Q will not cause P to block.

Moreover, given any two processes P and Q, Reed et al. demonstrate [25] how to mechanically construct the refinement assertion that corresponds to the binary relation Q *RespondsTo* P. Verification of this binary relation can then achieved by model checking the refinement assertion using the FDR tool. Due to property of responsiveness, we have recently established some results about compatibility.

Specifically, we have shown the collaboration of two compatible processes is deadlock free (Theorem 6.3); the property of compatibility is *refinement-closed* under the stable failures model (Theorem 6.4), and deadlock freedom of a network of processes is preserved when adding additional processes, which are compatible with individual processes of the network (Theorem 6.5). Proofs of theorems may be found in the first author's thesis [32, Section 4.5]. A property is refinement-closed if and only if for any process P that satisfies this property, then so do all P's refinements.

Theorem 6.3. *For any BPMN processes P and Q, if compatible(P, Q) then their collaboration is deadlock free, that is,*

$$compatible(P, Q) \Rightarrow DF \sqsubseteq_{\mathcal{F}} (P' \,[\![\, \{\!|m|\!\} \,]\!]\, Q')$$

where P' and Q' are the process semantics of P and Q respectively.

Theorem 6.4. *For any BPMN processes P and Q, if compatible(P, Q) and we have: $X \sqsubseteq_{\mathcal{F}} X' \wedge Y \sqsubseteq_{\mathcal{F}} Y'$, then compatible$(P', Q')$, where X, Y, X' and Y' are the process semantics of P, Q, P' and Q' respectively.*

Theorem 6.4 encourages *independent* compositional development using the monotonic operations [32, Section 4.4.3]. Specifically let G be a monotonic operation with respect to $\mathcal{F}$, if *compatible*$(G(P), G(Q))$, for all $P \sqsubseteq_{\mathcal{F}} P'$ and $Q \sqsubseteq_{\mathcal{F}} Q'$, *compatible*$(G(P'), G(Q'))$.

Theorem 6.5. *Let $B = \{i : I \bullet B(i)\}$, indexed by I, be a deadlock free collaboration of business participants (BPMN processes). That is, the process $C = \|\, i : I \bullet \alpha P(i) \circ P(i)$ is deadlock free, where the process semantics of each business participant $B(i)$ is denoted by process $P(i)$. Suppose there is a new business participant R, whose process semantics is denoted by CSP process Q, such that $\alpha Q \cap (\alpha P(i) \cap \alpha P(j)) = \emptyset$ for $i, j : I$ and $i \neq j$ (an appropriate assumption as by definition each message flow connects exactly two participants). Then if*

$$\exists\, i : I \bullet \alpha Q \cap \alpha P(i) \neq \emptyset \wedge \forall\, i : I \bullet \alpha Q \cap \alpha P(i) \neq \emptyset \Rightarrow \mathit{compatible}(R, B(i))$$

then $E = C \;[\![\bigcup\{i : I \bullet \alpha P(i)\} \mid \alpha Q]\!]\; Q$ is also deadlock free.

Theorem 6.5 allows one to construct business collaboration *incrementally*, even in a complex situation.

7. Related work

We have so far considered the property specification and LTL model checking of BPMN diagrams with respect to a CSP semantics. We also studied the notion of behavioural compatibility between BPMN processes in terms of CSP. In this section we provide an overview of related work in BPMN formalisation, LTL model checking for CSP and the study of behavioural compatibility.

7.1. BPMN formalisation

Related research work on BPMN has been aiming to provide a suitable formalisation for BPMN. Notable formal models at the time of constructing our formalisations include Dijkman et al.'s Petri net semantics [6]. Since our development of the semantics, there have been many other attempts to provide formalisations to BPMN. These include Arbab et al.'s formalisation in Reo [3], and Prandi et al.'s translation from BPMN to Calculus for Orchestrating Web Services (COWS) [24].

Dijkman et al. [6] provide a formalisation of BPMN in Petri nets. Similar to our process semantics, they model task, event and gateway elements as instantaneous actions/events, and message flows as synchronous communications between BPMN pools. However, their semantics of multiple instance activities is implicit, expressed by translating each multiple instance activity into atomic activities and gateways. Also they restrict their parallel and sequential multiple instance activities to have fixed numbers of instances, that is, the number of instances is known before the execution of the activity, while our process semantics permits both fixed and nondeterministic number of instances, modelling situations where the number of instances is only known at run time. Their formalisation lends itself to tool support for static analysis on properties like absence of dead tasks, that is, to check if there is a task that may never be performed within a model, and deadlock freedom.

In Arbab et al.'s approach [3], a translation from BPMN to the coordination language Reo [2], which has several operational semantics using different variations (such as quantitative and timed variants) of constraint automata, is provided. As a consequence, like our (relative) timed formalisation of BPMN [34], their semantics is one of several recent attempts to provide timed formalisations to BPMN. Moreover, their formalisation models both synchronous and asynchronous communications between BPMN pools. However, in their paper they only discussed informally how their formalisations may be used to verify properties concerning legal, regulatory, and business compliance.

In Prandi et al.'s approach [24], a translation from BPMN to the Calculus of Orchestrating Web Services (COWS) [13] is given. COWS is a process algebra, which is defined specifically for reasoning about the behaviour of web services. Due to the recent stochastic extension to COWS, in which Prandi and Quaglia implemented a semantics for the tool PRISM [11], their formalisation of BPMN may be extended with the facilities for quantitative analysis.

7.2. LTL model checking for CSP

To the best of our knowledge the earliest study of LTL model checking for the language of CSP has been carried by Leuschel et al. [14]. Similar to Lowe's work, they translate the satisfaction of LTL formulae into CSP refinement checks that can be carried out using FDR. However, their approach is inefficient and one of the reasons is that their translation leaves the larger, more complex process on the left side of the refinement check, which severely limits the applicability of the technique.

Sun et al. [29] consider bounded model checking of CSP processes based on LTL formulae. They provide a compositional encoding of hierarchical CSP processes as a set of Boolean formulae (SAT problems). Let ε be the SAT-based encoding of some CSP process P and ϕ be a temporal property specified in LTL extended with events. The satisfiability $P \models \phi$ is reduced into the following steps. First they encode $\neg\, \phi$ as a Büchi automaton B, they then compose ε with B such that $P \not\models \phi$ if and only if the language of $P \times B$ is not empty. This technique has been implemented in the Process Analysis Toolkit (PAT).

Plagge et al. [23] provide an extension of LTL that allows the specification of LTL formulae over Kripke structures with deadlock states and labelled transitions. This extension includes the $[t]$ construct for specifying that some operation t will be executed next. Their model checking algorithm has been implemented in the ProB tool.

7.3. Behavioural compatibility

Related studies on the notion of behavioural compatibility include Foster et al.'s [10,9] model-based approach, in which they translate both BPEL and WS-CDL into the Finite States Process notation (FSP) [17]. General liveness behavioural properties such as deadlock freedom may directly be verified using a model checker. More specific properties such as *obligation* [9], which describes what activities a subject must or must not do to a set of target objects, requires high-level specification of the corresponding policy in Message Sequence Charts [20] (MSCs). MSCs are translated into FSPs, and verification is then carried out by showing behavioural equivalence between the respective FSP processes via model checking.

A stronger notion of compatibility is known as compliance. Bravetti and Zavattaro [5,19] formalised the notion of strong service compliance. A composition of services is strongly compliant if their composition is both deadlock and livelock free, and whenever one service is to initiate an interaction with another service (via messaging), this other service must be prepared to engage. They then further developed this formal notion by considering service refinement, in which they consider a suitable pre-order between services such that substitutions of individual services in a composition by their refinements preserve compliance.

A related notion to end-point projection is *realisability*. Salaün and Bultan [27] define realisability to indicate whether participants can be generated from a high-level view of the collaboration such that they will behave exactly as formalised in its specification. If such specification is not realisable, they provide a technique for extending the behaviour between participants to realise the collaboration. They use collaboration diagrams [1] for the specification of the high-level view of the collaboration and provide an encoding of the diagram's abstract syntax into LOTOS process algebra [15].

8. Conclusion

In this paper we considered the application of Dwyer et al.'s Property Specification Patterns for constructing behavioural properties, against which CSP models of BPMN diagrams may be verified. We proposed a property specification language *PL* for capturing the generalisation of the property patterns in which constraints are specified over patterns of behaviours rather than individual events. We then described the translation from *PL* into a bounded, positive fragment of *LTL*, which can then be translated automatically into a corresponding CSP specification for simple refinement checks. We demonstrated the application of our specification language via a couple of small examples. We presented a detailed example on the behavioural properties of a multiple business process collaboration specifying an airline ticket reservation procedure. We also described some recent results on behavioural compatibility; these results lead to a compositional approach to ensuring deadlock freedom of interacting business processes.

A prototypical implementation of the translation, using Lowe's earlier implementation, may be found in the first author's web site [31]. Our intention is to implement tool support allowing developers to build property specifications without knowledge of *PL*, *LTL* or *CSP*.

Acknowledgements

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

References

[1] Unified modelling language: superstructure, Technical report, Object Management Group, 2004. Available at www.omg.org.
[2] F. Arbab, Reo: a channel-based coordination model for component composition, Mathematical Structures in Computer Science 14 (03) (2004).
[3] F. Arbab, N. Kokash, M. Sun, Towards using reo for compliance-aware business process modelling, in: Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, in: CCIS, vol. 17, Springer, 2008.
[4] Business Process Execution Language for Web Services, Version 1.1., May 2003. http://www.ibm.com/developerworks/library/ws-bpel.
[5] Mario Bravetti, Gianluigi Zavattaro, Towards a unifying theory for choreography conformance and contract compliance, in: Proc. of 6th International Symposium on Software Composition, SC'07, 2007.
[6] R.M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Information and Software Technology (2008).
[7] Matthew B. Dwyer, George S. Avrunin, James C. Corbett, Patterns in property specifications for finite-state verification, in: Proceedings of the 21st International Conference on Software Engineering, 1999.
[8] Formal Systems (Europe) Ltd., Failures-Divergences Refinement, FDR2 User Manual, 1998. www.fsel.com.
[9] H. Foster, S. Uchitel, J. Magee, J. Kramer, Model-based analysis of obligations in web service choreography, in: IEEE International Conference on Internet and Web Applications and Services, 2006.
[10] Howard Foster, Sebastian Uchitel, Jeff Magee, Jeff Kramer, Compatibility verification for web service choreography, in: IEEE International Conference on Web Services, 2004.
[11] A. Hinton, M. Kwiatkowska, G. Norman, D. Parker, Prism: a tool for automatic verification of probabilistic systems, in: Tools and Algorithms for the Construction and Analysis of Systems, in: LNCS, vol. 3920, 2006.
[12] Dexter Kozen, Results on the propositional mu-calculus, Theoretical Computer Science 27 (3) (1983).
[13] A. Lapadula, R. Pugliese, F. Tiezzi, Calculus for orchestration of web services, in: Programming Languages and Systems, in: LNCS, vol. 4421, 2007.
[14] Michael Leuschel, Thierry Massart, Andrew Currie, How to make FDR Spin LTL model checking of CSP by refinement, in: FME, 2001, pp. 99–118.
[15] LOTOS—A formal description technique based on the temporal ordering of observational behaviour, Technical Report Technical Report 8807, International Standards Organisation, 1989.
[16] Gavin Lowe, Specification of communicating processes: temporal logic versus refusals-based refinement, Formal Aspects of Computing 20 (3) (2008).

[17] J. Magee, J. Kramer, Concurrency—State Models and Java Programs, John Wiley, 1999.
[18] Zohar Manna, Amir Pnueli, The Temporal Logic of Reactive and Concurrent Systems, Springer-Verlag, 1992.
[19] Mario Bravetti, Gianluigi Zavattaro, A theory for strong service compliance, in: Proceedings of the 9th International Conference on Coordination Models and Languages, 2007.
[20] Message Sequence Charts, Technical Report Recommendation Z.120, International Telecommunications Union, 1996.
[21] Abida Mukarram, A refusal testing model for CSP, D. Phil thesis, University of Oxford, 1992.
[22] Object Management Group. Business Process Modeling Notation (BPMN) Specification, February 2006. www.bpmn.org.
[23] Daniel Plagge, Michael Leuschel, Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more, STTT 12 (1) (2010) 9–21.
[24] D. Prandi, P. Quaglia, N. Zannone, Formal analysis of bpmn via a translation into cows, in: Coordination Models and Languages, in: LNCS, vol. 5052, 2008.
[25] J.N. Reed, J.E. Sinclair, A.W. Roscoe, Responsiveness of interoperating components, Formal Aspects of Computing 16 (4) (2004) 394–411.
[26] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice-Hall, 1998.
[27] G. Salaün, T. Bultan, Realizability of choreographies using process algebra encodings, in: Proceedings of 7th International Conference on Integrated Formal Methods, in: LNCS, vol. 5423, 2009.
[28] Spec patterns. http://patterns.projects.cis.ksu.edu/.
[29] Jun Sun, Yang Liu, Jin Song Dong, Jing Sun, Bounded model checking of compositional processes, in: Proceedings of the Second IEEE International Symposium on Theoretical Aspects of Software Engineering, IEEE, 2008, pp. 23–30.
[30] W3C. Web service choreography interface (WSCI) 1.0, November 2002. www.w3.org/TR/wsci.
[31] Peter.Y. H. Wong, Model checking BPMN. http://www.comlab.ox.ac.uk/peter.wong/bpmn/.
[32] Peter Y.H. Wong, Formalisations and application of business process modelling notation, D. Phil thesis, University of Oxford, 2010. Draft available at http://www.comlab.ox.ac.uk/peter.wong/pub/thesis.pdf.
[33] Peter Y.H. Wong, Jeremy Gibbons, A process semantics for BPMN, in: Proceedings of 10th International Conference on Formal Engineering Methods, in: LNCS, vol. 5256, 2008, Extended version available at http://www.comlab.ox.ac.uk/peter.wong/pub/bpmnsem.pdf.
[34] Peter Y. H. Wong, Jeremy Gibbons, A relative-timed semantics for BPMN, in: Proceedings of 7th International Workshop on the Foundations of Coordination Languages and Software Architectures, ENTCS, 2008. Available at http://www.comlab.ox.ac.uk/peter.wong/pub/foclasa08.pdf.
[35] Peter Y.H. Wong, Jeremy Gibbons, Verifying business process compatibility, in: Proceedings of 8th International Conference on Quality Software, IEEE Computer Society, 2008, pp. 126–131.