Property Specifications for Workflow Modelling

Peter Y.H. Wong and Jeremy Gibbons

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

Abstract. Previously we provided two formal behavioural semantics for 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 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 demonstrate its application via a simple example.

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) [6], for which we previously provided two formal semantic models [8,9] in the process algebra CSP [7]. 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, due to the fact that the expressiveness of BPMN is strictly less than that of CSP, 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.

As a running example for this paper, consider the BPMN diagram describing a travel agent shown in Figure 1. The main purpose of the travel agent is to mediate interactions between the traveller who wants to buy airline tickets and the airline who supplies them. Specifically, once the travel agent receives an initial order from the traveller (*Receive_Order*), he needs to verify with the airline if the seats are available for the desired trip (*Check_Seats*). In order to cater for the possibility of the traveller making changes to her itinerary, for every change of her itinerary (*Change_Itin_TA*), the travel agent verifies with the airline the availability of the seats (*Check_Seats_2*). Once the traveller has agreed upon a particular itinerary (*Receive_Reservation*), the travel agent reserves the seats for the traveller (*Reserve_Seats*). During the reservation period, modelled by the *Reservation*

M. Leuschel and H. Wehrheim (Eds.): IFM 2009, LNCS 5423, pp. 56–71, 2009.

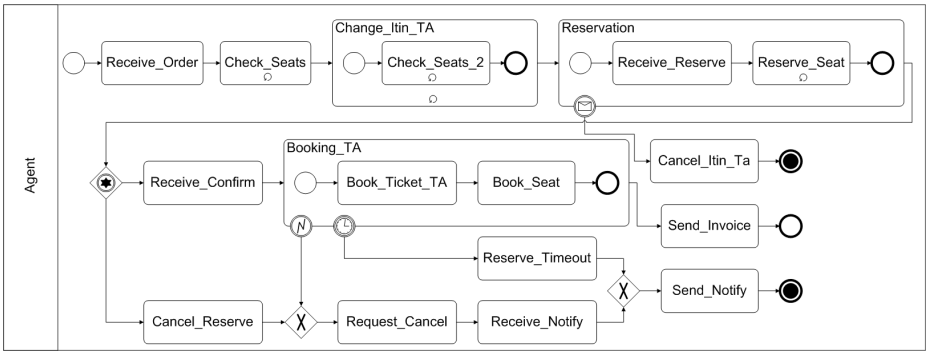

Fig. 1. Travel Agent

subprocess state, the traveller may cancel her itinerary, thereby "unreserving" the seats; this is modelled as a message exception flow (*Imessage*) of the *Reservation* subprocess. Once the reservation has been completed, the travel agent may receive a confirmation notice from the traveller (*Receive_Confirm*), in which case he receives the credit card information from the the traveller (*Book_Ticket_TA*) and proceeds with the booking (*Book_Seat*). The travel agent may also receive cancellation of the reservation (*Cancel_Reserve*), in which case he will request a cancellation from the airline (*Request_Cancel*), wait for a notification confirming the cancellation from the airline (*Receive_Notify*), and send it to the traveller (*Send_Notify*). During the booking phase, either an error (e.g. incorrect card information) or a time out (*Reserve_Timeout*) may occur; in both cases, corresponding notification confirming the cancellation will be sent to the traveller. Otherwise, a corresponding invoice on the booking will be sent to the traveller for billing (*Send_Invoice*).

One of the properties this travel agent description must satisfy is that the agent must not allow any kind of cancellation after the traveller has booked her tickets, if 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 Figure 2(a) 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_Cancel*, *Request_Timeout* and *Send_Invoice*. However, while this behavioural property should also permit other behaviours such as task *Request_Cancel* being performed before task *Book_Seat*, it could be difficult to specify all these behaviours in the same BPMN diagram. Since 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 which will allow domain specialists to express property specifications for verification of workflow processes.

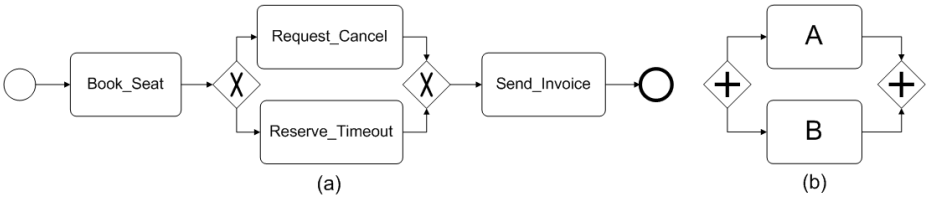

Fig. 2. (a) A BPMN diagram capturing requirement and (b) Parallel execution

1.1 Property Specification Patterns

This paper proposes the application of Dwyer et al's *Property Specification Patterns* [1] to assist domain specialists to specify behavioural properties for BPMN processes[1]. 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. 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 (*LTL*) [4] 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 to the best of our knowledge there is currently no formalisation of property patterns in CSP.

1.2 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". While CSP is equipped with nondeterministic 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.

[1] We assume readers have basic knowledge of CSP and that they are not required to have knowledge of BPMN.

$$a \rightarrow Skip \sqcap b \rightarrow Skip \sqsubseteq_{\mathcal{F}} a \rightarrow Skip \quad (1)$$

$$a \rightarrow Skip \;|||\; b \rightarrow Skip \sqsubseteq_{\mathcal{F}} a \rightarrow b \rightarrow Skip \quad (2)$$

This is because the parallel operators in CSP may be defined using the deterministic choice $\Box$ operator; here we show the interleaving of two processes and its equivalent 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, may be very useful for specifying behavioural properties for workflow processes. For example, Figure 2(b) shows part of a BPMN diagram executing tasks A and B in parallel. With our timed semantics for BPMN [9] it is possible to specify timing constraints for these tasks, and the diagram may then be interpreted over the timed model. It is easy to see the possibility of one asserting a behavioural property about tasks A and B within a wider scope without considering the ordering of their execution due to their timing constraints.

1.3 Our Approach

Our objective is to 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 first 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* [3]; 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* [3], 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 [7], since this model captures both safety and liveness properties. However, Lowe [3] 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 ($\mathcal{RT}$) [5] 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 eventual 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. Also $\Diamond \phi = \neg(\Box \neg \phi)$ and it is possible to capture the *always* operator, therefore it is not possible, in general to capture negation, unless only over atomic formulae as given by the grammar above. Our function reflects this by translating a given generalised pattern into a corresponding expression *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 $\mathit{Spec}(\psi) \sqsubseteq_{\mathcal{RT}} P$ where $\mathit{Spec}(\psi)$ is the CSP specification for ψ.

1.4 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. 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 gives a brief overview of the refusal traces model. In Section 3 we introduce *SPL*, a sub-language of our property specification language, for specifying nondeterministic patterns of behaviours; 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* for specifying behavioural properties based on generalised property patterns. We then define a function *makeTL*, which takes a property specification in *PL* and returns its corresponding temporal logic expression in *BTL*, and finally we revisit the travel agent running example and demonstrate how to specify the behavioural property in *PL*.

2 Refusal Traces Model

CSP [7] is equipped with three standard behavioural models: traces, stable failures and failures-divergences, in order of increasing precision. However, Lowe [3] has demonstrated that these models are inadequate for capturing temporal logic of the form described in previous section. The solution is to use the refusal traces model ($\mathcal{RT}$) [5].

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 $\langle X_1, a_1, X_2, a_2, .., X_n, a_n, \Sigma\rangle$, where each X_i is a refusal set, and each a_i is an event. This test 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 $\mathcal{RT}[\![P]\!]$ for the refusal traces of CSP process P. We now present the refusal traces semantics for some of the CSP operators,

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

where $Q \,\textbf{ref}\, X$ means that Q can refuse X initially.

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

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

Currently the CSP model checker, FDR [2], is being extended to include the checking of refinement in this model.

3 Patterns of Behaviour

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

$$
\begin{aligned}
P \in \mathit{SPL} &::= P \sqcap P \mid P \sqcap\!\sqcap P \mid a \to P \mid \mathit{End} \quad \textbf{where } a \in \mathit{Atom} \\
\mathit{Atom} &::= t \mid \texttt{available}\, t \mid \texttt{live} \quad \textbf{where } t \in \mathit{Task}
\end{aligned}
$$

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. An user interface for this language could be implemented to assist BPMN developers to construct specifications.

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 ($\rightarrow$), as well as a new *nondeterministic interleaving* operator ($\fatsemi$). Informally the process $P \fatsemi Q$ 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 [7]: if $P = p \rightarrow P'$ and $Q = q \rightarrow Q'$ then

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

and we present the laws of this operator over *end*:

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

The operator $\fatsemi$ is both commutative and associative and is defined in terms of nondeterministic choice $\sqcap$ and prefix $\rightarrow$. 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 $P, Q \in SPL$,

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

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

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

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

To convert formulae in BTL^* back to BTL, we simply remove $*$ according to the following equivalences: $\phi \vee * \equiv \phi$, $* \wedge \phi \equiv \phi$ and $\phi \wedge \bigcirc * \equiv \phi$; note both the conjunctive and disjunctive operators are commutative.

We map each of the operators other than $\fatsemi$ 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 $tr \in RT$:

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

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 the sequences s and t. Similarly we present Lowe's semantic definition [3] 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 $tr \in RT \cup IRT$:

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

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

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

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

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

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

Similar to CSP [7], 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 contains a possible initial

task enactment and the model after enacting that task. For example *hp* takes $a \to A \sqcap b \to B$ and returns the set $\{ (a, A), (b, B) \}$.

$$initials(P \sqcap Q) = initials(P) \cup initials(Q)$$
$$initials(P \mathbin{[\!]} Q) = initials(npar(P, Q))$$
$$initials(a \to P) = \{ (a, P) \}$$
$$initials(End) = \emptyset$$

Going back to the example in Figure 2(b), we are now able to specify the pattern of behaviour $(a \to End) \mathbin{[\!]} (b \to End))$ 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:

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

and *Spec* is the corresponding CSP process of ϕ. We use event a to associate with some task A.

$$\begin{aligned} Spec = \mathbf{let}\ & Spec0 = b \to Spec2 \quad Spec1 = a \to Spec3 \\ & Spec2 = a \to Spec4 \quad Spec3 = b \to Spec4 \\ & Spec4 = Stop \sqcap (\sqcap x : \Sigma \bullet x \to Spec4) \\ \mathbf{in}\ & Spec0 \sqcap Spec1 \end{aligned}$$

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.

$$Spec \sqsubseteq_{\mathcal{RT}} a \to b \to Stop \qquad Spec \sqsubseteq_{\mathcal{RT}} a \to Stop \mid\mid\mid b \to Stop$$

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:

$$\begin{aligned} x, y \in PL ::= {} & \mathtt{Abs}(p, s) \mid \mathtt{Un}(p, s) \mid \mathtt{Ex}(p, n, s) \mid \mathtt{BEx}(p, b, s) \mid \\ & x \vee y \mid x \wedge y \qquad \mathbf{where}\ p \in SPL;\ n \in \mathbb{N};\ b \in BL;\ s \in SL \\ BL ::= {} & \leq n \mid = n \mid \geq n \qquad \mathbf{where}\ n \in \mathbb{N} \\ SL ::= {} & \mathtt{always} \mid \mathtt{before}\,(p, n) \mid \mathtt{after}\,p \mid \qquad \mathbf{where}\ p \in SPL;\ n \in \mathbb{N} \\ & \mathtt{between}\,p\,\mathtt{and}\,(q, n) \mid \mathtt{from}\,p\,\mathtt{until}\,(q, n) \end{aligned}$$

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 this is where each transition between states is an application of a step law. We describe each term in *PL* briefly as follows:

- $\texttt{Abs}(p, s)$ (Absence) states that the pattern of behaviour p must be refused throughout the scope s;
- $\texttt{Un}(p, s)$ (Universality) states that the pattern of behaviour p must occur throughout the scope s;
- $\texttt{Ex}(p, n, s)$ (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* specification, 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;
- $\texttt{BEx}(p, b, s)$ (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 execution. For example $\texttt{Abs}(a \vee b, \texttt{always})$ states that both events a and b must be refused in all possible executions;
- $\texttt{before}\,(p, n)$ (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 $\texttt{Un}(\texttt{available}\,a, \texttt{before}\,(b, n))$ states that a must not be refused before an occurrence of b in the subsequent n states.
- $\texttt{after}\,p$ (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 $\texttt{BEx}(a \vee b, \leq m, \texttt{after}\,c)$ states that a sequence of at most m as and bs must occur after the occurrence of the event c.
- $\texttt{between}\,p\,\texttt{and}\,(q, n)$ (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.
- $\texttt{from}\,p\,\texttt{until}\,(q, n)$ (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 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 *PL*'s grammar does not include the patterns such as *Precedence* or *Response* [1]; 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 make

an assertion. For example the furthest state of the expression $a \vee b$ is 1; for both expressions $\bigcirc b$ and $a \wedge \bigcirc \texttt{available}\, c$ 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*. The function *next* is defined as the functional composition ($nexts \circ states$) where $states(\phi)$ returns one minus the furthest state the expression ϕ, translated from some pattern of behaviour in *SPL*, specifies. The function *nexts* is defined such that $nexts_n \phi$ returns a composition of ϕ with n next operators, assuming $nexts_0 \phi = \phi$. We write the predicate *single* such that some *BTL* expression μ satisfies it, denoted as $single(\mu)$, if and only if μ specifies behaviours for only a single state; we say such expressions are *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*. Since $\lhd\!\rhd_n\ \phi = \texttt{true}\ \widetilde{\mathcal{U}}_n\ \phi$, we only define the semantics of $\widetilde{\mathcal{U}}$ as follows:

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

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*:

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

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 $(a \vee b) \vee (\texttt{true} \wedge \bigcirc(a \vee b))$. 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 $\mathcal{R}$.

To assist our translation we define the partial function *negate* 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. 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*. Here we only provide the partial definition of *negate*, where $\phi, \psi \in BTL^\delta$ and $n \in \mathbb{N}$, omitting the more trivial part of the definition.

$$negate(\phi \Rightarrow \psi) = \phi \wedge (negate(\phi) \vee negate(\psi))$$
$$negate(\lhd\!\rhd_n\ \phi) = (negate \circ derive)(\lhd\!\rhd_n\ \phi)$$
$$negate(\psi\ \widetilde{\mathcal{U}}_n\ \phi) = (negate \circ derive)(\psi\ \widetilde{\mathcal{U}}_n\ \phi)$$
$$negate(\bigcirc\phi) = \bigcirc(negate(\phi))$$

The function *derive* converts an expression in BTL^δ back to *BTL* according to the definition given in equation 3. The full definition may be found in the technical report version of this paper [10].

We define a translation function *makeTL* to be the functional composition $(derive \circ tl')$ that takes a property specification in PL and returns its corresponding temporal logic expression in BTL. The definition of tl' is as follows:

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

where $n \in \mathbb{N}$, $\mu, \nu \in SPL$, $b \in Bound$, $s \in SL$, and $\sigma, \rho \in PL$. 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 both $q = pattern(\nu)$ 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*. For reasons of space we have chosen to describe only the formalisation of the *bounded existence* pattern and its corresponding function *boundexist*. Explanations for other patterns may be found in the longer technical report version of this paper [10].

4.1 Bounded Existence

The function *boundexist* takes a pattern of behaviour μ, a bound b and a scope s and returns the corresponding expression in BTL^{δ} stating μ must occur for the number of times specified by b within s and other behaviours may also within s. For reasons of space we assume every possible sequence of events defined by μ covers the same number of states, that is the *maximum* number of states. The longer technical report version of this paper [10] gives a complete formal treatment where this assumption is relaxed. Our definition also reflects the impossibility of expressing the *unbounded eventually* operator under the refusal traces model. We first define 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. Table 1 lists BTL^{δ} mappings for *bound*. Here we describe the formalisation for each type of bounds.

The expressions 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 since it is not possible to model unbounded eventually, and hence unbounded until operator, we restrict this pattern with all the occurrences of p occurring 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 expressions 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 any of the n instances of p may or may not occur and after which behaviour p may not occur until some other behaviour q occurs.

We now provide a description of our formalisation similar to the format when describing the absence pattern, assuming $p = pattern(\mu)$, $q = pattern(\nu)$ and $r = pattern(\upsilon)$. We write $getbound(b)$ for some bound b to denote the number part of the value.

- 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 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 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 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 $\texttt{BEx}(a \rightarrow End \sqcap c \rightarrow End, = 2, \texttt{after}\ b \rightarrow End)$, where a,b and c correspond to tasks A, B and C. The following is the CSP specification translated from the corresponding BTL expression,

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

where 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 Revisiting the Example

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 property of the travel agent. We denote task *Book_Seat* by event *bookseat*, and similarly for *Request_Cancel*, *Request_Timeout* and *Send_Invoice*; the following is the corresponding *PL* expression specifying this property.

$$\texttt{Abs}(\mathit{Cancel}, \texttt{between}\ \mathit{bookseat} \rightarrow \mathit{End}\ \texttt{and}(\mathit{sendinvoice} \rightarrow \mathit{End}, 2))$$

where the behaviour *Cancel* is defined as follows:

$$\mathit{Cancel} = \mathit{requestcancel} \rightarrow \mathit{End} \sqcap \mathit{reservetimeout} \rightarrow \mathit{End}$$

Here is the corresponding CSP process.

$$\begin{aligned}
\mathit{Spec} = \ & \textbf{let} \\
& \mathit{Spec}0 = \mathit{Proceed}(\{\ \mathit{bookseat}\ \}, \mathit{Spec}0 \sqcap \mathit{Spec}1) \\
& \mathit{Spec}1 = \mathit{bookseat} \rightarrow (\mathit{Spec}2 \sqcap \mathit{Spec}3 \sqcap \mathit{Spec}4 \sqcap \mathit{Spec}5 \sqcap \mathit{Spec}6) \\
& \mathit{Spec}2 = \mathit{Proceed}(\{\ \mathit{bookseat}, \mathit{sendinvoice}\ \}, \mathit{Spec}7 \sqcap \mathit{Spec}1) \\
& \mathit{Spec}3 = \mathit{sendinvoice} \rightarrow (\mathit{Spec}0 \sqcap \mathit{Spec}1) \\
& \mathit{Spec}4 = \mathit{bookseat} \rightarrow (\mathit{Spec}2 \sqcap \mathit{Spec}4 \sqcap \mathit{Spec}8 \sqcap \mathit{Spec}9) \\
& \mathit{Spec}5 = \mathit{Proceed}(\{\ \mathit{bookseat}, \mathit{requestcancel}, \mathit{reservetimeout}\ \}, \mathit{Spec}3) \\
& \mathit{Spec}6 = \mathit{bookseat} \rightarrow (\mathit{Spec}3) \\
& \mathit{Spec}7 = \mathit{Proceed}(\{\ \mathit{bookseat}, \mathit{sendinvoice}\ \}, \mathit{Spec}0 \sqcap \mathit{Spec}1) \\
& \mathit{Spec}8 = \textbf{let}\ \mathit{poss} = \{\mathit{bookseat}, \mathit{requestcancel}, \mathit{reservetimeout}, \mathit{sendinvoice}\} \\
& \qquad\qquad \textbf{in}\ \mathit{Proceed}(\mathit{poss}, \mathit{Spec}3) \\
& \mathit{Spec}9 = \mathit{bookseat} \rightarrow (\mathit{Spec}3) \\
& \textbf{in}\quad \mathit{Spec}0 \sqcap \mathit{Spec}1
\end{aligned}$$

Now it is possible to see if the travel agent diagram satisfies this property by checking the following refusal traces refinement assertion using the FDR tool.

$$\mathit{Spec} \sqsubseteq_{\mathcal{RT}} \mathit{Agent} \setminus \Sigma \setminus \{\mathit{bookseat}, \mathit{requestcancel}, \mathit{reservetimeout}, \mathit{sendinvoice}\}$$

5 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 describe the translation from *PL* into a bounded, positive fragment of *LTL*, which can then be translated automatically into its corresponding CSP specification for simple refinement checks. We have demonstrated the application of our specification language via a couple of small examples. We have implemented a Haskell prototype of the translation[2], using Lowe's implementation. Our intention is to implement tool support allowing developers to build property specifications without the knowledge of *PL*, *LTL* or *CSP*.

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

Acknowledgements

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

References

1. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proceedings of the 21st International Conference on Software Engineering (1999)
2. Formal Systems (Europe) Ltd. Failures-Divergences Refinement, FDR2 User Manual (1998), http://www.fsel.com
3. Lowe, G.: Specification of communicating processes: temporal logic versus refusals-based refinement. Formal Aspects of Computing 20(3) (2008)
4. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1992)
5. Mukarram, A.: A Refusal Testing Model for CSP. D.Phil thesis, University of Oxford (1992)
6. Object Management Group. BPMN Specification (February 2006), http://www.bpmn.org
7. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
8. Wong, P.Y.H., Gibbons, J.: A Process Semantics for BPMN. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 355–374. Springer, Heidelberg (2008), http://www.comlab.ox.ac.uk/peter.wong/pub/bpmnsem.pdf
9. Wong, P.Y.H., Gibbons, J.: A Relative-Timed Semantics for BPMN. In: Proceedings of 7th International Workshop on the Foundations of Coordination Languages and Software Architectures, July 2008. ENTCS (2008); Invited for special issue in Science of Computer Programming, http://www.comlab.ox.ac.uk/peter.wong/pub/foclasa08.pdf
10. Wong, P.Y.H., Gibbons, J.: Property Specifications for Workflow Modelling. Technical Report, University of Oxford (2008), http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/psp.pdf

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 $(\leq n)$	$nexts_{n * states(p)}\, (q\, \mathcal{R}\, \neg p)$