Property Specifications for Workflow Modelling

Peter Y. H. Wong and Jeremy Gibbons

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 construct behavioural properties against which BPMN models may be verified. This paper considers a pattern-based approach for capturing these 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 necessarily be easy or even possible 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. Specifcally 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 for the availability of the seats (*Check_Seats_2*). Once the traveller

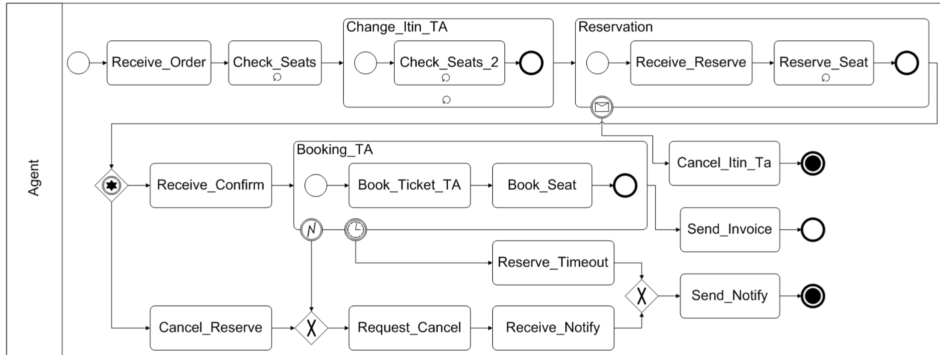

Figure 1: Travel Agent

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* 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 on the reservation (*Cancel_Reserve*), in which case he will request a cancellation from the airline (*Request_Cancel*), waits 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 agent must not allow any kind of cancellation after the traveller has booked his 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 permits 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 *avail-*

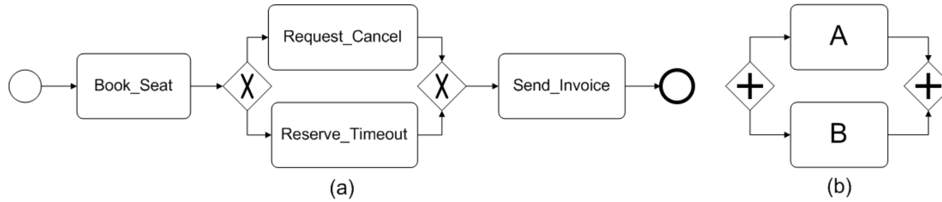

Figure 2: (a) A BPMN diagram capturing requirement and (b) Parallel execution

ability 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.

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. While it is possible to express infinite states systems, the subset of BPMN used to which the semantic models apply only models finite-state systems. Figure 3 illustrates the hierarchy of the property patterns[2]. 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.

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 nonde-

[1] We assume readers have basic knowledge of CSP.
[2] adopted from `http://patterns.projects.cis.ksu.edu/documentation/patterns.shtml`

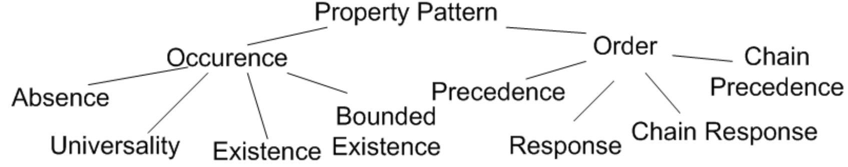

Figure 3: Pattern hierarchy

terministic choice as one of its standard operators, there is no nondeterministic version of parallel composition, this means that while assertion (1) holds under the failures refinement, assertion (2) does not.

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

$$a \rightarrow Skip \mid\mid\mid b \rightarrow Skip \quad \sqsubseteq_{\mathcal{F}} \quad a \rightarrow b \rightarrow Skip \tag{2}$$

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

$$a \rightarrow Skip \mid\mid\mid b \rightarrow Skip \equiv a \rightarrow b \rightarrow Skip \Box b \rightarrow a \rightarrow Skip$$

A nondeterministic version of the parallel operator, 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.

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 which 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.

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

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 *eventual* ($\Diamond$) 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.

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* 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 this paper is as follows. In Section 2 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 3 we present to the complete language *PL* for specifying behavioural properties based on the 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 running example travel agent and demonstrate how to specify the behavioural property in *PL*. We conclude this paper in Section 4.

2 Patterns of Behaviour

The language of CSP is expressive enough as both a specification language and a method of describing systems as implemented. This expressiveness has allowed us to use BPMN to describe more abstract behaviour as well as to model workflow processes as implemented. Since it is not necessary to employ all of the CSP operators to define the semantics of BPMN, BPMN alone is not enough to specify all types of behavioural properties. In particular it is not possible to describe nondeterminism using BPMN. 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:

$$P \in \mathit{SPL} ::= P \sqcap P \mid P \sqcap\!\!\sqcap P \mid a \rightarrow P \mid \mathit{End} \quad \textbf{where}\ a \in \mathit{Atom}$$
$$\mathit{Atom} \quad ::= t \mid \texttt{available}\ t \mid \texttt{live} \quad \textbf{where}\ t \in \mathit{Task}$$

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 $t \rightarrow P$ 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 user's 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 the nondeterministic choice ($\sqcap$) and prefix ($\rightarrow$), 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 [7]: if $P = p \rightarrow P'$ and $Q = q \rightarrow Q'$ then

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

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

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

Note the operator $\sqcap\!\sqcap$ is both commutative and associative and is defined in terms of the nondeterministic choice operator $\sqcap$ and the prefix operator $\rightarrow$. 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.

$$
\begin{array}{l}
pattern : SPL \rightarrow BTL^* \\
atom : SPL \rightarrow BTL \\
\hline
\forall\, a : Atom;\ t : Task;\ P, Q : SPL \bullet \\
\quad pattern(End) = * \\
\quad pattern(a \rightarrow P) = atom(a) \wedge \bigcirc(pattern\ P) \\
\quad pattern(P \sqcap Q) = (pattern\ P) \vee (pattern\ Q)) \\
\quad pattern(P \sqcap\!\sqcap Q) = pattern(npar(P, Q)) \\
\\
\quad atom(\texttt{available}\ t) = \texttt{available}\,(event(t)) \\
\quad atom(\texttt{live}) = \texttt{live} \\
\quad atom(t) = event(t)
\end{array}
$$

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

$$
\begin{array}{l}
\phi \vee * \equiv \phi \\
* \wedge \phi \equiv \phi \\
\phi \wedge \bigcirc * \equiv \phi
\end{array}
$$

Note 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 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.

$$
\begin{array}{l}
\mathcal{RT}_{SPL}[\![\, * \,]\!] = \emptyset \\
\mathcal{RT}_{SPL}[\![t \rightarrow P]\!] = \{\, \langle\rangle \,\} \cup \{\, a : \Sigma;\ X : \mathbb{P}\,\Sigma;\ tr : RT \mid \\
\qquad\qquad a = event(t) \wedge a \notin X \wedge tr \in \mathcal{RT}_{SPL}[\![P]\!] \bullet \langle X, a \rangle \frown tr \,\}
\end{array}
$$

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.

$$
\begin{array}{l}
\mathcal{RT}_{BTL}[\![a]\!] = \{\, \langle\rangle \,\} \cup \{\, a : \Sigma;\ X : \mathbb{P}\,\Sigma;\ tr : RT \cup IRT \mid a \notin X \bullet \langle X, a \rangle \frown tr \,\} \\
\mathcal{RT}_{BTL}[\![\bigcirc\phi]\!] = \{\, \langle\rangle, \langle\Sigma\rangle \,\} \cup \{\, a : \Sigma;\ X : \mathbb{P}\,\Sigma;\ tr : RT \cup IRT \mid \\
\qquad\qquad a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![\phi]\!] \bullet \langle X, a \rangle \frown tr \,\} \\
\mathcal{RT}_{BTL}[\![\psi \wedge \phi]\!] = \mathcal{RT}_{BTL}[\![\psi]\!] \cap \mathcal{RT}_{BTL}[\![\phi]\!]
\end{array}
$$

According to our translation function $\mathit{pattern}\ t \rightarrow P = \mathit{event}(t) \wedge \bigcirc(\mathit{pattern}\ P)$,

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

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 $P, Q \in \mathit{SPL}$.

$$
\begin{array}{|l}
\mathit{npar} : (\mathit{SPL} \times \mathit{SPL}) \rightarrow \mathit{SPL} \\
\hline
\forall P, Q : \mathit{SPL} \bullet \\
\quad \mathit{npar}(\mathit{End}, \mathit{End}) = \mathit{End} \\
\quad \mathit{npar}(\mathit{End}, Q) = Q \\
\quad \mathit{npar}(P, \mathit{End}) = P \\
\quad \mathit{npar}(P, Q) = \\
\qquad (\sqcap(a, X) : \mathit{initials}(P) \bullet a \rightarrow \mathit{npar}(X, Q)) \\
\qquad \sqcap (\sqcap(a, X) : \mathit{initials}(Q) \bullet a \rightarrow \mathit{npar}(X, P))
\end{array}
$$

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 finite set of pairs, each pair contains a possible initial task enactment and the model after enacting that task. For example hp takes $a \rightarrow A \sqcap b \rightarrow B$ and returns the sequence $\{(a, A), (b, B)\}$.

$$
\begin{array}{|l}
\mathit{hp} : \mathit{SPL} \nrightarrow \mathbb{F}(\mathit{Atom} \times \mathit{SPL}) \\
\hline
\forall a : \mathit{Atom};\ P, Q : \mathit{SPL} \bullet \\
\quad \mathit{hp}(P \sqcap Q) = \mathit{hp}(P) \cup \mathit{hp}(Q) \\
\quad \mathit{hp}(P \sqcap\!\sqcap Q) = \mathit{hp}(\mathit{npar}(P, Q)) \\
\quad \mathit{hp}(a \rightarrow P) = \{(a, P)\} \\
\quad \mathit{hp}(\mathit{End}) = \langle\rangle
\end{array}
$$

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

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

and *Spec* is the corresponding CSP process of ϕ. We use the event *starts.a* to associate with some task A in accordance with our BPMN's semantics [8].

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

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.

$$Spec \sqsubseteq_{\mathcal{RT}} starts.a \rightarrow starts.b \rightarrow Stop$$
$$Spec \sqsubseteq_{\mathcal{RT}} starts.a \rightarrow Stop \,|||\, starts.b \rightarrow Stop$$

Note 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 an atomic formula.

Definition 1 *An BTL expression is in* ***restricted disjunctive normal form*** *(rDNF) if it has the form,*

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

where each a^j_i *is a atomic formula and* $nexts_i\, a$ *is defined by* i $\bigcirc$ *operators over some formula* a.

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.

$$\bigcirc(a \wedge b) \equiv \bigcirc a \wedge \bigcirc b \qquad [\bigcirc\text{-}\wedge\text{-dist}]$$

$$a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c) \qquad [\wedge\text{-}\vee\text{-dist}]$$

Here we show these two laws are valid under $\mathcal{RT}$.

$$
\begin{aligned}
&\mathcal{RT}_{BTL}[\![\bigcirc\phi \wedge \bigcirc\psi]\!] \\
&\quad = \mathcal{RT}_{BTL}[\![\bigcirc\phi]\!] \cap \mathcal{RT}_{BTL}[\![\bigcirc\psi]\!] && [\text{def of } \wedge] \\
&\quad = \{\, a : \Sigma;\ X : \mathbb{P}\,\Sigma;\ tr : RT \cup IRT \mid a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![\phi]\!] \bullet \langle X, a\rangle \frown tr \,\} \\
&\qquad \cup \{\, \langle\rangle, \langle\Sigma\rangle \,\} \cap \mathcal{RT}_{BTL}[\![\bigcirc\psi]\!] && [\text{def of } \bigcirc] \\
&\quad = \{\, a : \Sigma;\ X : \mathbb{P}\,\Sigma;\ tr : RT \cup IRT \mid a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![\phi]\!] \bullet \langle X, a\rangle \frown tr \,\} \\
&\qquad \cap \{\, a : \Sigma;\ X : \mathbb{P}\,\Sigma;\ tr : RT \cup IRT \mid a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![\phi]\!] \bullet \langle X, a\rangle \frown tr \,\} \\
&\qquad \cup \{\, \langle\rangle, \langle\Sigma\rangle \,\} && [\text{def of } \bigcirc] \\
&\quad = \{\, a : \Sigma;\ X : \mathbb{P}\,\Sigma;\ tr : RT \cup IRT \mid a \notin X \wedge tr \in \mathcal{RT}_{BTL}[\![\phi \wedge \psi]\!] \bullet \langle X, a\rangle \frown tr \,\} \\
&\qquad \cup \{\, \langle\rangle, \langle\Sigma\rangle \,\} && [\text{def of } \cap] \\
&\quad = \mathcal{RT}_{BTL}[\![\bigcirc(\phi \wedge \psi)]\!]
\end{aligned}
$$

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

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

3 Property Patterns

To assist the specification of behavioural properties in terms of the generalised property patterns, we have defined a property specification language *PL* and it is defined by the following grammar:

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

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. Note 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 is captured by our sub-language *SL*. Here we describe each one briefly.

- $\texttt{always}$ (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 execution;
- $\texttt{before}\,(p, n)$ (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 execution. 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 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 shortcomings as these patterns, belonging the set of *order* patterns, may be expressed in the *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 to which ϕ asserts. For example the furthest state of the expression $a \vee b$ is 1 and both expressions $\bigcirc b$ and $a \wedge \bigcirc \texttt{available}\ c$ are 2.

$$
\begin{array}{|l}
next : (BTL \times BTL) \nrightarrow BTL \\
nexts : (\mathbb{N} \times BTL) \rightarrow BTL \\
\hline
\forall \phi, \psi : BTL;\ n : \mathbb{N} \bullet \\
\quad next(\phi, \psi) = nexts(states(\phi), \psi) \\
\quad nexts(0, \psi) = \psi \\
\quad nexts(n, \psi) = \bigcirc (nexts(n-1, \psi))
\end{array}
$$

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 $max(i, j)$ returns the maximum between i and j. The function *nexts* is defined such that $next(n, \phi)$ returns a composition of ϕ with n next operators. For presentation purpose we write $next_\phi(\psi)$ and $nexts_n(\psi)$ to denote $next(\phi, \psi)$ and $nexts(n, \psi)$ respectively.

$$
\begin{array}{|l}
states : BTL \nrightarrow \mathbb{N} \\
\hline
\forall \phi, \psi : BTL \bullet \\
\quad states(\phi \vee \psi) = max(states(\phi), states(\psi)) \\
\quad states(\phi \wedge \psi) = max(states(\phi), states(\psi)) \\
\quad states(\bigcirc \phi) = 1 + states(\psi) \\
\quad states(\phi) = 1
\end{array}
$$

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 $\Diamond$ and $\widetilde{\mathcal{U}}$ to express *bounded eventuality* and until and they are defined such that for any CSP process P, formulae ψ and ϕ,

$$P \models \Diamond_n\ \phi \equiv \forall\, tr : \mathcal{RT}[\![P]\!] \bullet \exists\, i : 0 \mathinner{..} n \bullet tr^i \in \mathcal{RT}_{BLT}[\![\phi]\!]$$

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

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 define the function *derive* to derive both bounded operators using the standard syntax of *BTL*.

$$
\begin{array}{l}
derive : BTL^{\delta} \rightarrow BTL \\
derive' : (BTL^{\delta} \times \mathbb{N}) \nrightarrow BTL^{\delta} \\
\hline
\forall n : \mathbb{N};\ \phi, \psi : BTL \bullet \\
\quad derive(\lhd\!\rhd_n\ \phi) = derive(derive1(\mathtt{true}, \phi, n)) \\
\quad derive(\psi\ \widetilde{\mathcal{U}}_n\ \phi) = derive(derive'(\psi, \phi, n)) \\
\quad derive(\psi \Rightarrow \phi) = derive(negate(\psi) \vee (\psi \wedge \phi)) \\
\quad derive(\bigcirc\phi) = \bigcirc(derive(\phi)) \\
\quad derive(\phi \wedge \psi) = derive(\phi) \wedge derive(\psi) \\
\quad derive(\phi \wedge \psi) = derive(\phi) \wedge derive(\psi) \\
\quad derive(\phi \mathcal{R} \psi) = derive(\phi)\, \mathcal{R}\, derive(\psi) \\
\quad derive(\phi) = \phi \\
\\
\quad derive'(\psi, \phi, 1) = \phi \\
\quad derive'(\psi, \phi, n) = \mathbf{if}\ single(\psi)\ \mathbf{then}\ \phi \vee (\psi \wedge \bigcirc(derive'(\psi, \phi, n-1))) \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \phi \vee (\psi \wedge next_{\psi}(derive'(\exists, \phi, n-1)))
\end{array}
$$

where we write $\phi \Rightarrow \psi$ as a shorthand for $\neg\phi \vee (\phi \wedge \psi)$ where ϕ and *psi* are expressions in BTL^{δ} and ϕ does not include operators $\Box$ and $\mathcal{R}$. 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 $(a \vee b) \vee (\mathtt{true} \wedge \bigcirc(a \vee b))$.

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 2 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*, omitting the more trivial part of the definition, where $\phi, \psi \in BTL^{\delta}$ and $n \in \mathbb{N}$.

$$
\begin{array}{l}
negate : BTL^{\delta} \nrightarrow BTL^{\delta} \\
\hline
\forall a : \mathrm{ran}\ event;\ \phi, \psi : BTL \bullet \\
\quad negate(\phi \vee \psi) = negate(\phi) \wedge negate(\psi) \\
\quad negate(\phi \wedge \psi) = negate(\phi) \vee negate(\psi) \\
\quad negate(\phi \Rightarrow \psi) = \phi \wedge (negate(\phi) \vee negate(\psi)) \\
\quad negate(\lhd\!\rhd_n\ \phi) = (negate \circ derive)(\lhd\!\rhd_n\ \phi) \\
\quad negate(\psi\ \widetilde{\mathcal{U}}_n\ \phi) = (negate \circ derive)(\psi\ \widetilde{\mathcal{U}}_n\ \phi) \\
\quad negate(\bigcirc\phi) = \bigcirc(negate(\phi)) \\
\quad negate(\mathtt{available}\ a) = \neg a \\
\quad negate(\mathtt{live}) = \mathtt{deadlock} \\
\quad negate(\mathtt{deadlock}) = \mathtt{live} \\
\quad negate(a) = \neg a \\
\quad negate(\neg a) = a
\end{array}
$$

We define a translation function $makeTL$ which takes a property specification in PL and returns its corresponding temporal logic expression in BTL.

$$
\begin{array}{l}
makeTL : PL \rightarrow BTL \\
makeTL' : PL \rightarrow BTL^{\delta} \\
\hline
\forall\, n : \mathbb{N};\ \mu, \nu : SPL;\ b : Bound;\ s : SL;\ \sigma, \rho : PL \bullet \\
\quad makeTL = (derive \circ makeTL') \\
\quad makeTL'(\sigma \wedge \rho) = makeTL'(\sigma) \wedge makeTL'(\rho) \\
\quad makeTL'(\sigma \vee \rho) = makeTL'(\sigma) \vee makeTL'(\rho) \\
\quad makeTL'(\mathtt{Abs}(\mu, s)) = absence(\mu, s) \\
\quad makeTL'(\mathtt{Ex}(\mu, n, s)) = exist(\mu, n, s) \\
\quad makeTL'(\mathtt{BEx}(\mu, b, s)) = boundexist(\mu, b, s) \\
\quad makeTL'(\mathtt{Un}(\mu, s)) = universal(\mu, s)
\end{array}
$$

Table 4 shows BTL^{δ} mappings of functions *absence*, *exist*, *universal* and *boundexist*, the rest of this section provides explanations on each of the mapping defined for all of these functions

3.1 Absence

The function *absence* takes a pattern of behaviour μ and a scope s and returns the corresponding expression in BTL^{δ} stating μ must not occur within s.

$$
\begin{array}{l}
absence : (SPL \times SL) \rightarrow BTL^{\delta} \\
\hline
\forall\, p, q, r : BTL;\ \mu, \nu, \upsilon : SPL;\ n \in \mathbb{N} \mid \\
\quad p = pattern(\mu) \wedge q = pattern(\nu) \wedge r = pattern(\upsilon) \bullet \\
\quad\quad absence(\mu, \mathtt{always}) = \Box\, negate(p) \\
\quad\quad \wedge\ absence(\mu, (\mathtt{before}\,(\nu, n))) = (\Box\, negate(q)) \vee (negate(p)\, \widetilde{\mathcal{U}}_n\ q) \\
\quad\quad \wedge\ absence(\mu, (\mathtt{after}\,\nu)) = \Box(q \Rightarrow (next_q\,(\Box\, negate(p)))) \\
\quad\quad \wedge\ absence(\mu, (\mathtt{between}\,\nu\,\mathtt{and}\,(\upsilon, n))) = \\
\quad\quad\quad \Box(q \Rightarrow (next_q(\lhd\!\rhd_n\ r \Rightarrow (negate(p)\, \widetilde{\mathcal{U}}_n\ r)))) \\
\quad\quad \wedge\ absence(\mu, (\mathtt{from}\,\nu\,\mathtt{until}\,(\upsilon, n))) = \\
\quad\quad\quad \Box(q \Rightarrow (next_q\,(\Box\, negate(p) \vee (negate(p)\, \widetilde{\mathcal{U}}_n\ r))))
\end{array}
$$

Here we provide a description of our formalisation, assuming $p = pattern(\mu)$, $q = pattern(\nu)$ and $r = pattern(\upsilon)$. We write $\neg p$ for some pattern of behaviour of p to represent the negation of p by distributing $\neg$ as described by the function *negate*.

- 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 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 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 the following.

> if some event x is ever to occur then some other event y must occur after but without specifying x ever being performed.

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

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

  which states 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 if the behaviour ν occurs then either μ must be refused there after or μ must be refused until υ occurs in the subsequent n state 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 and it may be expressed in PL as $\texttt{Abs}(a \to End \sqcap b \to End, \texttt{after}\ c \to End)$ and the following is the CSP specification translated from the corresponding BTL expression $\Box((a \vee b) \Rightarrow \bigcirc(\Box\neg c))$.

$$
\begin{array}{l}
Spec = \\
\quad \textbf{let} \\
\quad\quad Spec0 = \\
\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad poss = \Sigma \setminus \{\, starts.c \,\} \\
\quad\quad\quad\quad proceed = \bigsqcap x : poss \bullet x \to (Spec0 \sqcap Spec1) \\
\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad Skip \sqcap Stop \sqcap (\textbf{if}\ poss = \emptyset\ \textbf{then}\ \mathsf{div}\ \textbf{else}\ proceed) \\
\quad\quad Spec1 = starts.c \to (Spec2 \sqcap Spec3) \\
\quad\quad Spec2 = \\
\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad poss = \Sigma \setminus \{\, starts.a, starts.b, starts.c \,\} \\
\quad\quad\quad\quad proceed = \bigsqcap x : poss \bullet x \to (Spec2 \sqcap Spec3) \\
\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad Skip \sqcap Stop \sqcap (\textbf{if}\ poss = \emptyset\ \textbf{then}\ \mathsf{div}\ \textbf{else}\ proceed) \\
\quad\quad Spec3 = starts.c \to (Spec2 \sqcap Spec3) \\
\quad \textbf{in} \\
\quad\quad Spec0 \sqcap Spec1
\end{array}
$$

3.2 Universality

The function *univsersal* takes a pattern of behaviour μ and a scope s and returns the corresponding expression in BTL^{δ} stating only μ may occur throughout s. Note 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 [2] for such specification. Our definition of *universal* reflects this.

$$
\begin{array}{l}
\mathit{universal} : (\mathit{SPL} \times \mathit{SL}) \rightarrow \mathit{BTL}^{\delta} \\
\hline
\forall\, p, q, r : \mathit{BTL};\ \mu, \nu, \upsilon : \mathit{SPL};\ n \in \mathbb{N} \mid \\
\quad p = \mathit{pattern}(\mu) \wedge q = \mathit{pattern}(\nu) \wedge r = \mathit{pattern}(\upsilon) \bullet \\
\qquad \mathit{universal}(\mu, \mathtt{always}) = \textbf{if } \mathit{single}\ p \textbf{ then } \Box p \textbf{ else } p \\
\qquad \wedge\ \mathit{universal}(\mu, (\mathtt{before}\,(\nu, n))) = \\
\qquad\qquad \textbf{if } \mathit{single}(p) \textbf{ then } (\Box\, \mathit{negate}(q)) \vee p\, \widetilde{\mathcal{U}}_n\ q) \\
\qquad\qquad \textbf{else if } n > \mathit{states}(p) \textbf{ then } (\Box\, \mathit{negate}(q)) \vee (p \wedge \mathit{nexts}_n(q)) \\
\qquad\qquad\quad \textbf{else } (\Box\, \mathit{negate}(q)) \vee (p \wedge \mathit{next}_p(q)) \\
\qquad \wedge\ \mathit{universal}(\mu, (\mathtt{after}\,\nu)) = \\
\qquad\qquad \textbf{if } \mathit{single}(p) \textbf{ then } \Box(q \Rightarrow (\mathit{next}_q\,(\Box\, p))) \textbf{ else } \Box(q \Rightarrow (\mathit{next}_q\, p)) \\
\qquad \wedge\ \mathit{universal}(\mu, (\mathtt{between}\,\nu\,\mathtt{and}\,(\upsilon, n))) = \\
\qquad\qquad \textbf{if } \mathit{single}(p) \textbf{ then } \Box(q \Rightarrow \mathit{next}_q\,(\Diamond_n\ r \Rightarrow (p\, \widetilde{\mathcal{U}}_n\ r))) \\
\qquad\qquad \textbf{else if } n > \mathit{states}(p) \textbf{ then } \Box(q \Rightarrow \mathit{next}_q\,(\Diamond_n\ r \Rightarrow (p \wedge \mathit{nexts}_n r))) \\
\qquad\qquad\quad \textbf{else } \Box(q \Rightarrow \mathit{next}_q\,(\Diamond_{\mathit{states}(p)+1}\ r \Rightarrow (p \wedge \mathit{next}_p r))) \\
\qquad \wedge\ \mathit{universal}(\mu, (\mathtt{from}\,\nu\,\mathtt{until}\,(\upsilon, n))) = \\
\qquad\qquad \textbf{if } \mathit{single}(p) \textbf{ then } \Box(q \Rightarrow (\mathit{next}_q\,(\Box\, p \vee (p\, \widetilde{\mathcal{U}}_n\ r)))) \\
\qquad\qquad \textbf{else if } n > \mathit{states}(p) \textbf{ then } \Box(q \Rightarrow (\mathit{next}_q\,(p \vee \mathit{nexts}_n r))) \\
\qquad\qquad\quad \textbf{else } \Box(q \Rightarrow (\mathit{next}_q\,(p \vee \mathit{next}_p r)))
\end{array}
$$

Here we provide a description of our formalisation similar to the format when describing the absence pattern, assuming $p = \mathit{pattern}(\mu)$, $q = \mathit{pattern}(\nu)$ and $r = \mathit{pattern}(\upsilon)$.

- 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 either the behaviour ν does not exist or μ occurs repeatedly until ν occurs in the subsequent n states where $n > \mathit{states}(p)$. Otherwise, this is modelled as $(\Box\, \mathit{negate}(q)) \vee (p \wedge \mathit{nexts}_n(q))$, which states 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 $n > \mathit{states}(p)$;
- The occurrence of μ after some behaviour ν is modelled as $\Box(q \Rightarrow (\mathit{next}_q\,(\Box\, p)))$ if μ is a single state specification and it states 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 if ν occurs then μ occurs in the next immediate states;

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

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

  if μ is a single state specification. This expression states 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 $\Box(q \Rightarrow next_q\,(\Diamond_n\ r \Rightarrow (p \wedge nexts_n r)))$ and this states 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 if the behaviour ν occurs then either μ must occur repeatedly there after or μ must occur repeatedly until υ occurs in the subsequent n state after ν has occurred. Otherwise this pattern is modelled as $\Box(q \Rightarrow (next_q\,(p \vee nexts_n r)))$ and this states if the behaviour ν occurs then either one instance of μ must occur immediately after ν and υ might occur in the subsequent n state 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 only task A or B can be performed between tasks D and C. This may be expressed in PL as $\texttt{Un}(a \rightarrow End \sqcap b \rightarrow End, \texttt{between}\, c \rightarrow End\, \texttt{and}\, (d \rightarrow End, 2))$ and the following is the CSP specification translated from the corresponding BTL expression.

$$
\begin{array}{l}
Spec = \\
\quad \textbf{let} \\
\quad\quad Spec0 = \\
\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad poss = \Sigma \setminus \{\, starts.d \,\} \\
\quad\quad\quad\quad proceed = \sqcap\, x : poss \bullet x\ \textbf{then}\ (Spec0 \sqcap Spec1) \\
\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad Skip \sqcap Stop \sqcap (\textbf{if}\ poss == \emptyset\ \textbf{then}\ \mathsf{div}\ \textbf{else}\ proceed) \\
\quad\quad Spec1 = starts.d \rightarrow (Spec2 \sqcap Spec3 \sqcap Spec4 \sqcap Spec5 \sqcap Spec6) \\
\quad\quad Spec2 = \\
\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad poss = \Sigma \setminus \{\, starts.c, starts.d \,\}) \\
\quad\quad\quad\quad proceed = \sqcap\, x : poss \bullet x \rightarrow (Spec7 \sqcap Spec8) \\
\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad Skip \sqcap Stop \sqcap (\textbf{if}\ poss == \emptyset\ \textbf{then}\ \mathsf{div}\ \textbf{else}\ proceed)
\end{array}
$$

$$
\begin{aligned}
&Spec3 = starts.d \rightarrow (Spec2 \sqcap Spec3 \sqcap Spec9 \sqcap Spec10)\\
&Spec4 = starts.c \rightarrow (Spec0 \sqcap Spec1)\\
&Spec5 = starts.b \rightarrow Spec4\\
&Spec6 = starts.a \rightarrow Spec4
\end{aligned}
$$

$$
\begin{aligned}
&Spec7 = \\
&\quad \mathbf{let}\\
&\qquad poss = \Sigma \setminus \{ starts.c, starts.d \})\\
&\qquad proceed = \sqcap x : poss \bullet x \rightarrow (Spec0 \sqcap Spec1)\\
&\quad \mathbf{in}\\
&\qquad Skip \sqcap Stop \sqcap (\mathbf{if}\ poss == \emptyset\ \mathbf{then}\ \mathsf{div}\ \mathbf{else}\ proceed)
\end{aligned}
$$

$$
\begin{aligned}
&Spec8 = starts.d \rightarrow (Spec2 \sqcap Spec3 \sqcap Spec4 \sqcap Spec5 \sqcap Spec6)\\
&Spec9 = starts.b \rightarrow Spec4\\
&Spec10 = starts.a \rightarrow Spec4\\
\mathbf{in}&\\
&Spec0 \sqcap Spec1
\end{aligned}
$$

3.3 Existence

The function *exist* takes a pattern of behaviour μ and a scope s and returns the corresponding expression in BTL^δ stating one instance of μ must occur within s and other behaviours may also within s. Again our definition reflects the impossibility of expressing unbounded eventually operator under the refusal traces model.

$$
\begin{aligned}
&exist : (SPL \times \mathbb{N} \times SL) \rightarrow BTL^\delta\\
\hline
&\forall\, p, q, r : BTL;\ \mu, \nu, \upsilon : SPL;\ m, n \in \mathbb{N} \mid\\
&\quad p = pattern(\mu) \wedge q = pattern(\nu) \wedge r = pattern(\upsilon) \bullet\\
&\qquad exist(\mu, m, \mathtt{always}) = \diamond_m\ p\\
&\qquad \wedge\ exist(\mu, m, (\mathtt{before}\,(\nu, n))) =\\
&\qquad\qquad \mathbf{if}\ n \geq m + states(p)\ \mathbf{then}\ \diamond_n\ q \Rightarrow (negate(q)\,\widetilde{\mathcal{U}}_m\ p)\\
&\qquad\qquad \mathbf{else}\ \diamond_{m+states(p)}\ q \Rightarrow (negate(q)\,\widetilde{\mathcal{U}}_m\ p)\\
&\qquad \wedge\ exist(\mu, m, (\mathtt{after}\,\nu)) = \Box(q \Rightarrow (next_q\,(\diamond_m\ p)))\\
&\qquad \wedge\ exist(\mu, m, (\mathtt{between}\,\nu\,\mathtt{and}\,(\upsilon, n))) =\\
&\qquad\qquad \mathbf{if}\ n \geq m + states(p)\\
&\qquad\qquad \mathbf{then}\ \Box(q \Rightarrow next_q\,(\diamond_n\ r \Rightarrow ((negate(r)\,\widetilde{\mathcal{U}}_m\ p) \wedge r\,\mathcal{R}\,negate(q))))\\
&\qquad\qquad \mathbf{else}\ \Box(q \Rightarrow next_q\,(\diamond_{m+states(p)}\ r \Rightarrow ((negate(r)\,\widetilde{\mathcal{U}}_m\ p) \wedge r\,\mathcal{R}\,negate(q))))\\
&\qquad \wedge\ exist(\mu, m, (\mathtt{from}\,\nu\,\mathtt{until}\,(\upsilon, n))) = \Box(q \Rightarrow next_q\,(negate(r)\,\widetilde{\mathcal{U}}_m\ p))
\end{aligned}
$$

Here we provide a description of our formalisation similar to the format when describing the absence pattern, assuming $p = pattern(\mu)$, $q = pattern(\nu)$ and $r = pattern(\upsilon)$.

- The global existence μ is modelled trivially as $\diamond_n\ p$ which simply states 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

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

, 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

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

and it 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 μ occurs in one of the subsequent m state 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

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

and it states if the behaviour ν occurs then either μ must occur in one of the subsequent m state 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 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 $\texttt{Ex}(a \rightarrow b \rightarrow End, 2, \texttt{after}\ c \rightarrow End \sqcap d \rightarrow End)$ and the following is the CSP specification translated from the corresponding BTL expression.

$$
\begin{array}{l}
Spec = \\
\quad \textbf{let} \\
\quad\quad Spec0 = \\
\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad poss = \Sigma \setminus \{\, starts.c, starts.d \,\} \\
\quad\quad\quad\quad proceed = \bigsqcap x : poss \bullet x \rightarrow (Spec0 \sqcap Spec1 \sqcap Spec2) \\
\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad Skip \sqcap Stop \sqcap (\textbf{if}\ poss = \emptyset\ \textbf{then}\ \mathsf{div}\ \textbf{else}\ proceed) \\
\quad\quad Spec1 = starts.d \rightarrow (Spec3 \sqcap Spec4 \sqcap Spec5 \sqcap Spec6) \\
\quad\quad Spec2 = starts.c \rightarrow (Spec3 \sqcap Spec4 \sqcap Spec5 \sqcap Spec6) \\
\quad\quad Spec3 = starts.a \rightarrow Spec7
\end{array}
$$

$$
\begin{array}{l}
\quad \mathit{Spec}4 = \\
\qquad \mathbf{let} \\
\qquad\quad \mathit{poss} = \Sigma \setminus \{\, \mathit{starts.c}, \mathit{starts.d} \,\} \\
\qquad\quad \mathit{proceed} = \sqcap x : \mathit{poss} \bullet (x \rightarrow (\mathit{Spec}3)) \\
\qquad \mathbf{in} \\
\qquad\quad \mathit{Skip} \sqcap \mathit{Stop} \sqcap (\mathbf{if}\ \mathit{poss} = \emptyset\ \mathbf{then}\ \mathsf{div}\ \mathbf{else}\ \mathit{proceed}) \\
\quad \mathit{Spec}5 = \mathit{starts.d} \rightarrow \mathit{Spec}3 \\
\quad \mathit{Spec}6 = \mathit{starts.c} \rightarrow \mathit{Spec}3 \\
\quad \mathit{Spec}7 = \mathit{starts.b} \rightarrow (\mathit{Spec}0 \sqcap \mathit{Spec}1 \sqcap \mathit{Spec}2) \\
\mathbf{in} \\
\quad \mathit{Spec}0 \sqcap \mathit{Spec}1 \sqcap \mathit{Spec}2
\end{array}
$$

3.4 Bounded Existence

While it only requires the *maximum* number of states of the patterns of behaviour when specifying properties in the patterns addressed so far, it is necessary to calculate *all possible number of states* of the pattern of behaviour for specifying properties in the bounded existence pattern. This is because to express a context over bounded number of occurrences of some pattern of behaviour μ, we need to know *exactly* the number of states all occurrences of μ spans. 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. To accommodate this we define the function *combine* such that $\mathit{combine}(\mu, n)$ returns μ a set of patterns of behaviour, each defines a disjunction of possible n occurrences of possible behaviour by μ such that each disjunct covers equal number of states. Here we assume μ is in *rDNF*.

$$
\begin{array}{|l}
\mathit{combine} : \mathit{BTL} \rightarrow\!\!\!\!\!\!+ \ \mathbb{N} \rightarrow\!\!\!\!\!\!+ \ \mathbb{F}_1\, \mathit{BTL} \\
\mathit{joins} : \mathrm{seq}_1\, \mathit{BTL} \rightarrow\!\!\!\!\!\!+ \ \mathit{BTL} \\
\mathit{repeat} : \mathit{BTL} \rightarrow\!\!\!\!\!\!+ \ \mathbb{F}\, \mathit{BTL} \rightarrow\!\!\!\!\!\!+ \ \mathbb{N} \rightarrow\!\!\!\!\!\!+ \ \mathbb{F}_1(\mathrm{seq}_1\, \mathit{BTL}) \\
\mathit{disjunct} : \mathit{BTL} \rightarrow\!\!\!\!\!\!+ \ \mathbb{F}_1\, \mathit{BTL} \\
\hline
\forall\, p, q : \mathit{BTL};\ \mathit{bs} : \mathrm{seq}_1\, \mathit{BTL};\ n : \mathbb{N} \bullet \\
\quad \mathit{combine}(\mathit{ps}, 1) = \mathit{disjunct}(\mathit{ps}) \\
\quad \wedge\ \mathit{combine}(\mathit{ps}, n) = \bigcup \{\, p : \mathit{disjunct}(\mathit{ps}) \bullet \{\, q : \mathit{repeat}(\mathit{ps}, \{\, p \,\}, n-1) \bullet \mathit{joins}(q) \,\} \,\} \\
\\
\quad \mathit{joins}(\langle p \rangle) = p \\
\quad \wedge\ \mathit{joins}(\langle p \rangle \frown \mathit{bs}) = p \wedge \mathit{next}_p \mathit{joins}(\mathit{bs}) \\
\\
\quad \mathit{repeat}(\mathit{ps}, \mathit{fs}, 1) = \{\, p : \mathit{ps} \bullet \mathit{fs} \frown \langle\, p \,\rangle \,\} \\
\quad \wedge\ \mathit{repeat}(\mathit{ps}, \mathit{fs}, n) = \{\, p : \mathit{ps} \bullet \bigcup (\mathit{repeat}(\mathit{ps}, \mathit{fs} \frown \langle\, p \,\rangle, n-1)) \,\} \\
\\
\quad \mathit{disjunct}(p \vee q) = \mathit{disjunct}(p) \cap \mathit{disjunct}(q) \\
\quad \wedge\ \mathit{disjunct}(p) = \{\, p \,\}
\end{array}
$$

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

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

Consequently we are able to define the function *boundexists*, which 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.

$$\begin{array}{l}
boundexists : (SPL \times Bound \times SL) \rightarrow\!\!\!\!\!+ BTL^\delta \\
\hline
\forall\, ps : \mathbb{F}_1\, BTL;\ \mu : SPL;\ b : Bound;\ n \in \mathbb{N}_1;\ s : SL \mid \\
\quad ps = combine(patternDNF(\mu), getbound(b)) \bullet \\
\quad\quad boundexists(\mu, b, s) = \bigvee\{\, p : ps \bullet boundexist(p, b, s) \,\}
\end{array}$$

where the function *boundexist* considers individual partitions of possible alternative behaviour such that each partition contains a set of behaviour, each of which covers equal number of states. Our definition *boundexist* reflects the impossibility of expressing unbounded eventually operator under the refusal traces model. We write $getbound(b)$ for some bound b to denote the number part of the value.

$$\begin{array}{l}
boundexist : (BTL \times Bound \times SL) \rightarrow\!\!\!\!\!+ BTL^\delta \\
\hline
\forall\, p, q, r : BTL;\ \mu, \nu, \upsilon : SPL;\ b : Bound;\ n \in \mathbb{N}_1 \mid \\
\quad q = pattern(\nu) \wedge r = pattern(\upsilon) \bullet \\
\quad\quad boundexist(p, b, \mathtt{always}) = bound(p, \mathtt{false}, b) \\
\quad\quad \wedge\ boundexist(p, b, (\mathtt{before}\,(\nu, n))) = \\
\quad\quad\quad\quad \lozenge_n\ q \Rightarrow negate(q)\, \widetilde{\mathcal{U}}_{n - getbound(b) * states(p)}\ bound(p, q, b) \\
\quad\quad \wedge\ boundexist(p, b, (\mathtt{after}\,\nu)) = \Box(q \Rightarrow next_q(bound(p, q, b))) \\
\quad\quad \wedge\ boundexist(p, b, (\mathtt{between}\,\nu\,\mathtt{and}\,(\upsilon, n))) = \\
\quad\quad\quad\quad \mathbf{if}\ n > getbound(b) * states(p) \\
\quad\quad\quad\quad \mathbf{then}\ \Box(q \Rightarrow (next_q\ \lozenge_n\ r \Rightarrow \\
\quad\quad\quad\quad\quad\quad (bound(p, r, b) \wedge bound(p, r, b)\, \mathcal{R}\, negate(r) \wedge r\, \mathcal{R}\, negate(q)))) \\
\quad\quad\quad\quad \mathbf{else}\ \Box(q \Rightarrow (next_q\ \lozenge_{getbound(b) * states(p) + 1}\ r1 \Rightarrow \\
\quad\quad\quad\quad\quad\quad (bound(p, r, b) \wedge bound(p, r, b)\, \mathcal{R}\, negate(r) \wedge r\, \mathcal{R}\, negate(q)))) \\
\quad\quad \wedge\ boundexist(p, m, (\mathtt{from}\,\nu\,\mathtt{until}\,(\upsilon, n))) = \\
\quad\quad\quad\quad \Box(q \Rightarrow (next_q negate(r)\, \widetilde{\mathcal{U}}_1\ bound(p, r \vee q, b)))
\end{array}$$

The function *bound* is defined such that $bound(p, q, b)$ returns the corresponding expression in BTL^δ stating a bounded existence of behaviour p with no scope, with an ending assertion $(\neg p)\, \mathcal{R}\, q$ for bounds $= n$ and $\leq n$ where $n \in \mathbb{N}_1$, this ending condition is to ensure behaviour p is refused after its bound until some behaviour q as the beginning of its scope occurs again.

$$
\begin{array}{l}
bound : (BTL \times BTL \times Bound) \nrightarrow BTL^{\delta} \\
\hline
\forall\, p, q : BTL;\ m \in \mathbb{N}_1 \bullet \\
\quad bound(p, q, = 1) = p \wedge next_p\ (q\, \mathcal{R}\, negate(p)) \\
\quad \wedge\ bound(p, q, = m) = p \wedge next_p\ bound(p, q, = (m-1)) \\
\quad \wedge\ bound(p, q, \geq 1) = p \\
\quad \wedge\ bound(p, q, \geq m) = p \wedge next_p\ bound(p, q, \geq (m-1)) \\
\quad \wedge\ bound(p, q, \leq 1) = (p \vee negate(p)) \wedge next_p\ (q\, \mathcal{R}\, negate(p)) \\
\quad \wedge\ bound(p, q, \leq m) = (p \vee negate(p)) \wedge next_p\ bound(p, q, \leq (m-1))
\end{array}
$$

Here we describe the formalisation for each type of bounds.

- The expressions to model exactly n $(= n)$ existences of behaviour p may be written as $\bigwedge_{i \in n-1} (nexts_{i * states(p)}\ p) \wedge nexts_{n * states(p)}\ (q\, \mathcal{R}\, \neg p)$. Note since it is not possible to model unbounded eventually, and hence unbounded until operator, we restrict this pattern with all the 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 $(\geq n)$ existences of behaviour p may be written as $\bigwedge_{i \in n-1} (nexts_{i * states(p)}\ p)$. 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 $(\leq n)$ existences 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.

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

  $$\Diamond\!\!\rhd_n\ q \Rightarrow negate(q)\ \widetilde{\mathcal{U}}_{n - getbound(b) * states(p)}\ bound(p, q, b)$$

  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 behaviours ν and υ is modelled as

$$\Box(q \Rightarrow (next_q \mathbin{\lhd\!\rhd}_n\ r \Rightarrow (bound(p, r, b) \wedge bound(p, r, b)\,\mathcal{R}\,\neg r \wedge r\,\mathcal{R}\,\neg q)))$$

  and it 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 behaviours ν until υ is modelled as

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

  and it 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 $\mathtt{BEx}(a \rightarrow End \sqcap c \rightarrow End, = 2, \mathtt{after}\ b \rightarrow End)$ and the following is the CSP specification translated from the corresponding BTL expression.

$$
\begin{array}{l}
Spec = \\
\quad \textbf{let} \\
\quad\quad Spec0 = \\
\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad poss = \Sigma \setminus \{\, starts.b \,\} \\
\quad\quad\quad\quad proceed = \bigsqcap x : poss \bullet x \rightarrow (Spec0 \sqcap Spec1) \\
\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad Stop \sqcap Skip \sqcap (\textbf{if}\ poss = \emptyset\ \textbf{then}\ \mathsf{div}\ \textbf{else}\ proceed) \\
\quad\quad Spec1 = starts.b \rightarrow (Spec2 \sqcap Spec3) \\
\quad\quad Spec2 = starts.c \rightarrow (Spec4 \sqcap Spec5) \\
\quad\quad Spec3 = starts.a \rightarrow (Spec4 \sqcap Spec5) \\
\quad\quad Spec4 = starts.c \rightarrow (Spec6 \sqcap Spec7) \\
\quad\quad Spec5 = starts.a \rightarrow (Spec6 \sqcap Spec7) \\
\quad\quad Spec6 = \\
\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad poss = \Sigma \setminus \{\, starts.a, starts.b, starts.c \,\} \\
\quad\quad\quad\quad proceed = \bigsqcap x : poss \bullet x \rightarrow (Spec6 \sqcap Spec7) \\
\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad Stop \sqcap Skip \sqcap (\textbf{if}\ poss = \emptyset\ \textbf{then}\ \mathsf{div}\ \textbf{else}\ proceed) \\
\quad\quad Spec7 = starts.b \rightarrow (Spec2 \sqcap Spec3) \\
\quad \textbf{in} \\
\quad\quad Spec0 \sqcap Spec1
\end{array}
$$

3.5 Revisiting the Example

Going back to our running example, recall the property of the travel agent diagram is that **agent must not allow any kind of cancellation after the traveller has booked his tickets, if invoice is to be sent to the traveller**. We may use the absence pattern "the absence of μ between some behaviours ν and υ" to specify this property. Let's assume *bookseat*, *requestcancel*, *reservetimeout* and *sendinvoice* denote the tasks *Book_Seat*, *Request_Cancel*, *Request_Timeout* and *Send_Invoice* respectively; the following is the corresponding *PL* expression specifying this property.

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

where the behaviour *Cancel* is defined as follows:

$$Cancel = requestcancel \rightarrow End \sqcap reservetimeout \rightarrow End$$

Here is the corresponding CSP process.

$$\begin{array}{l}
Spec = \textbf{let} \\
\qquad Spec0 = \ \textbf{let}\ poss = \Sigma \setminus \{\, bookseat \,\} \\
\qquad\qquad\qquad Proceed = \bigsqcap x : poss \bullet x \rightarrow (Spec0 \sqcap Spec1) \\
\qquad\qquad\quad \textbf{in}\ \ Skip \sqcap Stop \sqcap Proceed \\
\qquad Spec1 = bookseat \rightarrow (Spec2 \sqcap Spec3 \sqcap Spec4 \sqcap Spec5 \sqcap Spec6) \\
\qquad Spec2 = \ \textbf{let}\ poss = \Sigma \setminus \{\, bookseat, sendinvoice \,\} \\
\qquad\qquad\qquad Proceed = \bigsqcap x : poss \bullet x \rightarrow (Spec7 \sqcap Spec1) \\
\qquad\qquad\quad \textbf{in}\ \ Skip \sqcap Stop \sqcap Proceed \\
\qquad Spec3 = sendinvoice \rightarrow (Spec0 \sqcap Spec1) \\
\qquad Spec4 = bookseat \rightarrow (Spec2 \sqcap Spec4 \sqcap Spec8 \sqcap Spec9) \\
\qquad Spec5 = \ \textbf{let}\ poss = \Sigma \setminus \{\, bookseat, requestcancel, reservetimeout \,\} \\
\qquad\qquad\qquad Proceed = \bigsqcap x : poss \bullet x \rightarrow Spec3) \\
\qquad\qquad\quad \textbf{in}\ \ Skip \sqcap Stop \sqcap Proceed \\
\qquad Spec6 = bookseat \rightarrow (Spec3) \\
\qquad Spec7 = \ \textbf{let}\ poss = \Sigma \setminus \{\, bookseat, sendinvoice \,\} \\
\qquad\qquad\qquad Proceed = \bigsqcap x : poss \bullet x \rightarrow (Spec0 \sqcap Spec1) \\
\qquad\qquad\quad \textbf{in}\ \ Skip \sqcap Stop \sqcap Proceed \\
\qquad Spec8 = \ \textbf{let}\ poss = \Sigma \setminus \{\, bookseat, requestcancel, \\
\qquad\qquad\qquad\qquad\qquad reservetimeout, sendinvoice \,\} \\
\qquad\qquad\qquad Proceed = \bigsqcap x : poss \bullet x \rightarrow Spec3 \\
\qquad\qquad\quad \textbf{in}\ \ Skip \sqcap Stop \sqcap Proceed \\
\qquad Spec9 = bookseat \rightarrow (Spec3) \\
\quad \textbf{in}\ \ Spec0 \sqcap Spec1
\end{array}$$

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.

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

4 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, using Lowe's implementation[3]. Our intention is to implement tool support allowing developers to build property specifications without the knowledge of *PL*, *LTL* or *CSP*.

This work is supported by a grant from Microsoft Research.

References

[1] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[2] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27(3), 1983.

[3] G. Lowe. Specification of communicating processes: temporal logic versus refusals-based refinement. *Formal Aspects of Computing*, 20(3), 2008.

[4] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[5] A. Mukarram. *A Refusal Testing Model for CSP*. D.Phil thesis, University of Oxford, 1992.

[6] Object Management Group. *BPMN Specification*, Feb. 2006. `www.bpmn.org`.

[7] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[8] P. Y. H. Wong and J. Gibbons. A Process Semantics for BPMN. In *Proceedings of 10th International Conference on Formal Engineering Methods*, 2008. To appear. Extended version available at `http://www.comlab.ox.ac.uk/peter.wong/pub/bpmnsem.pdf`.

[9] P. Y. H. Wong and J. Gibbons. A Relative-Timed Semantics for BPMN. In *Proceedings of 7th International Workshop on the Foundations of Coordination Languages and Software Architectures*, July 2008. Available at `http://www.comlab.ox.ac.uk/peter.wong/pub/foclasa08.pdf`.

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

	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`$\,\nu\,$`and`$\,(\upsilon, n)$	$\Box(q \Rightarrow next_q\,(\lhd\!\rhd_n\ r \Rightarrow (p\,\widetilde{\mathcal{U}}_n\ r)))$ 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`$\,\nu\,$`until`$\,(\upsilon, n)$	$\Box(q \Rightarrow (next_q\,(\Box\, p \vee (p\,\widetilde{\mathcal{U}}_n\ r))))$ 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`$\,\nu\,$`and`$\,(\upsilon, n)$	$\Box(q \Rightarrow (next_q\ \lhd\!\rhd_n\ r \Rightarrow (bound(p, r, b) \wedge bound(p, r, b)\,\mathcal{R}\,\neg r \wedge r\,\mathcal{R}\,\neg q)))$	$\Box(q \Rightarrow next_q\,(\lhd\!\rhd_n\ r \Rightarrow (\neg r\,\widetilde{\mathcal{U}}_m\ p)))$
`from`$\,\nu\,$`until`$\,(\upsilon, n)$	$\Box(q \Rightarrow (next_q \neg r\,\widetilde{\mathcal{U}}_1\ bound(p, r \vee q, b)))$	$\Box(q \Rightarrow next_q\,(\neg r\,\widetilde{\mathcal{U}}_m\ p))$

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

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