

# Conformance Relations between Business Processes

Ragnhild Van Der Straeten  
System and Software Engineering Lab  
Vrije Universiteit Brussel  
Pleinlaan 2, 1050 Brussel, Belgium  
`rvdstrae@vub.ac.be`

**Abstract.** *This document reports on possible relationships between business processes. Different notations can be used to express business processes. The Business Process Modelling Notation (BPMN) is an emergent notation standard providing businesses with the capability of understanding their internal business procedures. The possible usage of BPMN is to represent business processes that are mapped to (abstract) Business Process Execution Language for Web Services (BPEL4WS) processes. Also UML 2.0 activities can be used to represent such business processes. These business process models in BPMN or UML abstract away from implementation or execution details. Different versions of the same business process model can reside on different levels of abstractions similar to models used in the traditional software development life-cycle. The question arises which kinds of relations can be defined between these different versions of a business process model. Also in the BPEL4WS community current issues of discussion are: what are the different kinds of relationships that can exist between BPEL4WS (abstract) processes and how can these relationships be represented. In this report we first present a well-defined representation of business processes that is independent of any notation or process language. Based on this representation, two semantic conformance relationships between business processes are defined. Next, we show how Description Logics can be used to represent business processes and how violations to the conformance relationships can be detected. To validate our approach, a prototype plug-in initially developed for inconsistency management in a UML CASE tool is used to check the conformance relations.*



# 1 Introduction

Due to the service oriented architecture there is an increasing interest in business process modelling techniques. This interest is even more boosted by the specification of BPEL4WS [11], shortly BPEL, which provides a means to specify business processes and interaction protocols and by the development of BPEL execution engines. Business processes can be modelled using the BPMN notation [4] or by UML 2.0 activities [12]. Using these high-level notations business process models can be expressed at different levels of abstraction similar to the traditional software development process.

BPEL defines *abstract* processes. There are two ways to describe processes in BPEL. On the one hand, *executable business processes* model actual behaviour of a participant in a business interaction. *Business protocols*, on the other hand, use process descriptions that specify the mutually visible message exchange behaviour of each of the parties involved in the protocol, without revealing their internal behaviour. The process descriptions for business protocols are called *abstract processes* (see also BPEL Specification [11]). In this report an *abstract process* denotes *a model of a business process representing the behaviour of a participant or the process of message exchange between different parties*. These processes are specified on a certain level of abstraction and in a business process notation such as BPMN or UML 2.0. These abstract processes can be reused in different contexts. The goal is to refine these processes in different service development steps and tailor these processes towards a certain process language (for example, BPEL) or execution environment or context.

The question now arises which kinds of relationships are valid between different versions of a process on different levels of abstraction. Remark that the same question arises for BPEL abstract processes. BPEL issues can be posted on the WS BPEL issue list [6]. Several issues on this list are on the specification of BPEL abstract processes and the compatibility between abstract processes on the one hand and abstract and executable processes on the other hand. These issues recognise the need to be more explicit about the relationship and compatibility of abstract and executable processes. The proposed resolution is to define abstract BPEL processes by common base and a usage profile. The common base specifies the syntax to which all BPEL abstract processes must conform to. “*Profiles are provided to define classes of abstract processes with a common (shared) semantic interpretation of the members of the class*” [5]. A profile must define the set of permitted executable completions for abstract processes that belong to the class of abstract processes. We will not focus on the definition of classes of abstract processes through the identification of permitted executable completions, but we will focus on the relationship that must hold between a source and target process. The relationships defined in this report can be interpreted as refinements, specifying how the protocol defined by the source process

conforms to the protocol defined by the target process.

Our approach taken can be summarised as follows. First, we will present a lightweight formalisation of business processes. Consequently, business processes are well-defined in our work. Our formalisation will also define some semantic concepts. These concepts let us easily and precisely define two different kinds of conformance relations between two business processes. We will also show how business processes can be expressed in a logic formalism, Description Logics (DLs) [3]. As a result of this representation, the meaning of the business processes and especially the sequence flows are determined and well-defined. Based on the reasoning tasks of the DL formalism, these relationships can be automatically checked between different processes.

This research is also inspired by our previous research done in the context of inconsistency management in software engineering. In [15] and [16] we defined certain relationships between the behaviour of a superclass and the behaviour of its subclasses. We also observed that similar relationships can be defined between a class and its evolved version. These relationships are similar to the relationships presented and defined in this report. We also used DLs to detect and handle violations against the defined relationships between a superclass and its subclasses or between a class and its evolved version.

The remainder of this report is structured as follows. The next section introduces a running example that will be used throughout this report. Section 3 presents our formalisation of business processes and semantic business process concepts. Based on this formalisation, precise definitions of conformance relations are given in Section 4. Section 5 introduces DLs and the translation of business processes to DLs. Section 7 summarises related work. A discussion of our approach and future work is given in Section 8. Section 9 concludes this report.

## 2 Running Example

The running example used throughout this report illustrates a real life business process of reserving and booking tickets for a trip. It is based on the example used in [19].

In Figure 1 the process of ordering, booking and reserving tickets is shown from the viewpoint of the Travel Agent. On the left-hand side of this figure, the process is shown in a BPMN business process diagram, while on the right-hand side the process is expressed in a UML 2.0 activity diagram<sup>1</sup>. The *Order Trip* action starts the process. Next the best itinerary is evaluated. The availability of the journey included in this best itinerary is checked. Using this information, the travel agent can constitute the ideal

---

<sup>1</sup>For an extensive overview of the different modelling elements of BPMN and UML 2.0 activities, we refer to the respective specification documents.

travel plan and can propose an itinerary to a traveller. The traveller can accept or reject the proposed itinerary. As a result the sequence flow enters a decision point (called an event-based gateway in BPMN). At this point, the process waits for an event to arrive and this will cause the sequence flow to follow the branch that leads to the intermediate event that was received. If an accept message is received, the tickets are reserved, booked and a confirmation message is sent to the traveller. In case the traveller rejects the itinerary, modifications are submitted and the travel agent re-computes an itinerary.

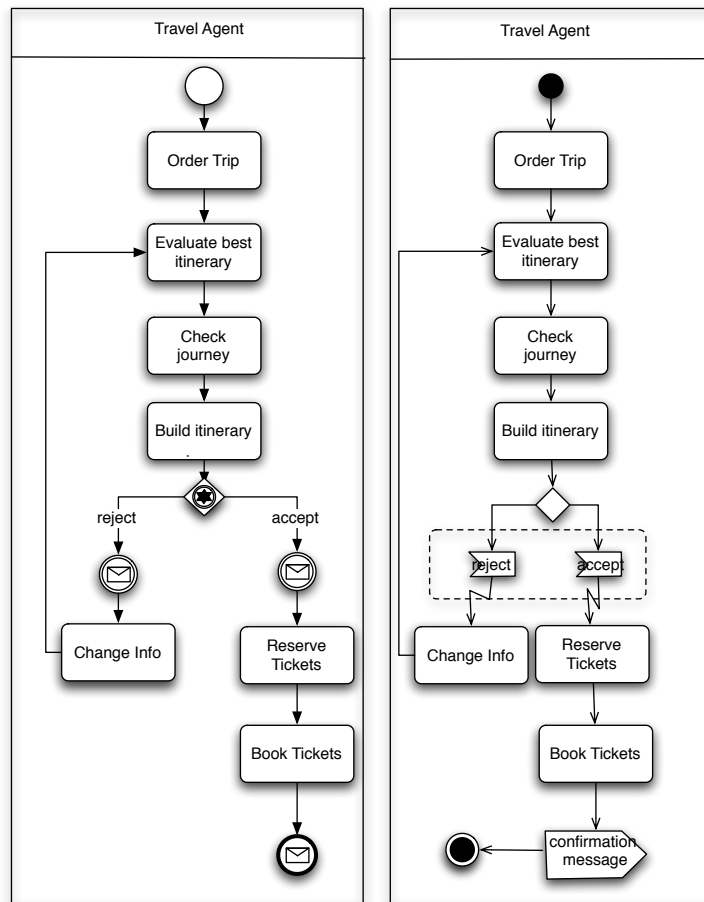


Figure 1: Business process for reserving and booking tickets.

This process is on a fairly high-level of abstraction and is not yet executable by, e.g., BPEL. We can further refine the process by, e.g., adding more detail through the addition of subprocesses, tasks, gateways, transactions, etc. .

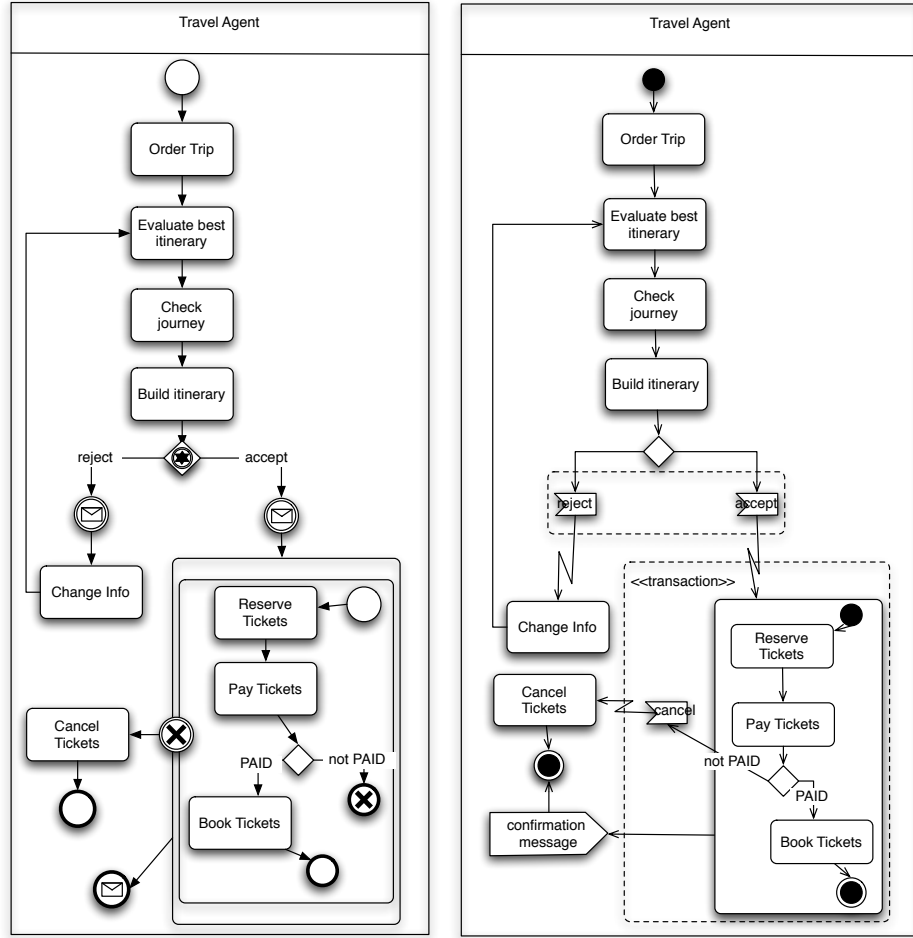


Figure 2: Business process of Figure 1 refined.

Figure 2 shows the same process as in Figure 1, but the behaviour of this process has been extended. Again, at the left-hand side the process is modelled using BPMN, while at the right-hand side the process is modelled using UML 2.0 activities. The reservation and booking of tickets actions belong to a transaction process that can be cancelled. This cancellation results in a *Cancel Tickets* action and the end of the process. Also a *Pay Tickets* action has been added. If the payment succeeds the tickets are booked, if the tickets are not paid, the payment action triggers a cancel action. This cancellation action also results in the *Cancel Tickets* action and the process ends.

One way to interpret these diagrams is that they specify all the action sequences that are *observable* to the party using this service, e.g., in our case

the traveller. In this interpretation, a possible conformance relationship is to demand that each sequence of actions observable with respect to a refined process must result under projection to the actions known in an observable sequence of the original process. This relationship is valid between the processes described in Figure 1 and Figure 2, respectively.

Another way to interpret these diagrams is that they specify all the flow sequences that are *invocable* by a certain part, e.g., in our case the traveller. In this interpretation, a conformance relation can be defined that demands that each sequence that is invocable on a certain process is also invocable on its refined version.

To be able to precisely define these conformance relationships between processes, we need to formalise business processes.

### 3 Lightweight Formalisation of Business Processes

The BPMN is just a notation and not a language. To turn the BPMN into a language, an abstract syntax and semantics must be added to it. UML 2.0 activity diagrams describe work and object flow and can be used to describe workflows. The UML 2.0 is a huge language that is full of compromises. It also lacks a formal semantics. To be able to precisely define relationships between different business processes we first must provide an abstract syntax and semantics for these processes.

The formalisation presented in this section is independent of any notation. As a result it can be used as a formalisation of the UML 2.0 activity diagram or of the BPMN business process diagrams. This formalisation must satisfy the following criteria: (1) it must be powerful enough to capture the essence of business processes; (2) we want a general formalisation independent of any specific run-time semantics.

In this section, we will restrict ourselves to the definition of the concepts necessary to define the observation and invocation conformance relationships that are intuitively introduced in the previous section.

**Notation 1** *The set of all business processes is denoted by  $\Omega$ .*

*The set of all actions is denoted by  $\mathbf{A}$ .*

*The set of all labels is denoted by  $\mathbf{L}$ .*

*The set of all conditions is denoted by  $\mathbf{G}$ .*

*The set of all operations is denoted by  $\mathbf{Op}$ .*

*The set of all triggers is denoted by  $\mathbf{T}$ .*

A business process can be defined as follows:

**Definition 1** A business process  $\omega = (A, F, L, \rho, \Lambda) \in \Omega$  consists of a set of Actions  $A \subseteq \mathbf{A}$  and a labelled flow set  $F \subseteq \mathcal{P}(A) \times L \times \mathcal{P}(A)$  containing labelled relations  $(AS, \tau, AS')$  such that  $AS, AS' \subseteq A$ ,  $\tau \in L$  where  $L \subseteq \mathbf{L}$  is a set of labels.

A label is a couple  $\tau = (t, g)$ , where  $t \in \mathbf{T}$  is the trigger triggering this flow and  $g \in \mathbf{G}$  is the guard that can be specified on the flow. Remark that as well the trigger as the guard are optional. A label can also have the following formats:  $(\epsilon_t, g)$ ,  $(t, \epsilon_g)$  or  $(\epsilon_t, \epsilon_g)$ .  $\epsilon_t$  represents a dummy trigger and  $\epsilon_g$  a dummy guard.

A trigger can be associated to an action:

The partial function **Generatedtrigger** :  $\mathbf{A} \rightarrow \mathbf{T}$ .

An operation can be associated to a trigger:

The partial function **Operationtrigger** :  $\mathbf{T} \rightarrow \mathbf{Op}$ .

$\rho$  denotes the top-most initial node of the business process, and  $\nexists AS \subseteq A$ ,  $\nexists AS' \subseteq A$ :  $\rho \in AS' \wedge (AS, \tau, AS') \in T$ .

$\Lambda$  denotes the set of final nodes of the business process, for which  $\nexists AS \subseteq \Lambda$ ,  $\nexists AS' \subseteq A$ :  $(AS, \tau, AS') \in T$ .

An action in our definition corresponds to a UML action and to a BPMN task, respectively. Subprocesses can be defined similar to our notion of process. Our definition also supports advanced BPMN gateways and UML control nodes. We first define the notion of (active) action configuration and next, we explain the different kinds of flows and show how these are supported by Definition 1.

**Definition 2** An action configuration  $\alpha$  in a business process  $\omega = (A, F, L, \rho, \Lambda) \in \Omega$  is defined as  $\alpha \subseteq A$ .

We now show how Definition 1 supports the different BPMN gateways. In Appendix A an overview of the support of these BPMN gateways by our model can be found. Remark that these gateways correspond to certain UML 2.0 activity diagram control nodes and activity nodes.

- A **start event** indicates where a particular process will start. In our definition a business process has always an initial node. Start events can have triggers that define the cause of the event. Depending on the kind of trigger specified on the start event different flows will originate from this initial node.
  - A start event can be **triggered by a message**. The message is generally sent from an outside source and triggers the start of the process.
  - In case of a **time trigger** on a start event, the process is started whenever the timer reaches a specific time or date.



- A **rule trigger** indicates that a signal, sent from another process based on the satisfaction of a rule, should start the current process.
- A **link trigger** indicates that a signal is sent from another process should start the current process.

In all cases a certain trigger must be received or condition must be fulfilled before the process can start. In our model, these events are translated into a flow relation  $(\rho, \tau, A)$ , where  $\rho$  denotes the start event and  $\tau = (t, g)$  where  $t$  represents the message trigger, time trigger, rule trigger or link trigger, respectively.  $g$  is an optional guard that can occur, e.g., in case of a time trigger or rule trigger specifying the time condition, rule condition, respectively.

- An **intermediate event** occurs between a start and an end event. It will influence the flow of the process. Intermediate events also have triggers defining the cause of the event. Also in the case of intermediate events there are various ways that these events can be triggered.

- The **message intermediate event** is the receipt of a message from an outside source. It can occur within a normal flow, or it can be attached to the boundary of action or it can occur within an event-based decision. In all cases this event is a place where a message is expected from an outside source. If the event is attached to the boundary of an action, the reception of the message will interrupt the action and the flow exiting the intermediate event will be followed. In case the message intermediate event occurs in an event-based decision the first message that arrives will determine the flow that is taken.

In our model a message intermediate event generates a flow relation  $(A, \tau, A')$  where  $A$  is the action leading to the intermediate event and  $A'$  is the action resulting from this intermediate event. The label  $\tau = (t, g)$  consists of the message trigger  $t$  and an optional guard  $g$ .

- The **timer intermediate event** is triggered when time reaches the time specified in the event. Similar to the message trigger, this trigger can be used between two actions in a normal flow or can be attached to the boundary of an action or can be used in an event-based decision. In all cases a timer intermediate event is in our model represented by a flow relation that has an appropriate trigger in its label.
- The **exception intermediate event** reacts to or generates an exception. This event can be used between two actions in a normal flow. When used in the normal flow, the event acts as a

place where an exception should be triggered. The process will trigger the exception and then continue with the next action. In our model this event is modelled as an explicit action that has a trigger associated to it.

This exception intermediate event can also be attached to the boundary of an action or it can be used in an event-based decision pattern. In the latter case, the first event that arrives will be executed. In both cases in our model a flow relation is generated with a label that contains the appropriate trigger.

- A **cancel trigger** can be attached to an intermediate event. The resulting event type is always attached to the boundary of a transaction action. It acts as a mechanism to interrupt this transaction action. The representation of this intermediate event in our model is similar to the representation of an exception intermediate event attached to the boundary of an action, i.e., a flow relation with a label containing this cancel trigger.
- A **compensation intermediate event** reacts or generates a compensation event. This trigger can be used in the normal business flow or it can be attached to the boundary of an action. The representations of this event are similar to the corresponding ones of the exception intermediate event.
- A **rule intermediate event** is triggered when a business rule becomes true during the execution of the process. This trigger can occur in the normal flow of the process or it can be attached to the boundary of an action or it can also be used in an event-based decision pattern. This event is in all three cases represented in our model by a trigger associated to the flow relation.
- A **terminate event** determines where the process will end. A terminate event trigger determines how the process will end. Possible terminate event triggers are: message, exception, compensate, cancel and link. In all cases, the corresponding events are represented in our model by an action and its associated trigger and/or condition.

We can conclude that the representation of these events in our model depends on whether these events are triggered in the process or received by the process. The UML 2.0 activity diagrams has different notations for the reception or the sending of signals. In Figure 3 the notation and its corresponding representation in our model is shown.

BPMN also defines different kinds of gateway control types. Using these different kinds of gateways different decision patterns or workflow patterns can be created [14]. The different types of control include: XOR decision/merging, OR decision/merging, complex decisions and forking/joining. We now explain how our Definition 1 copes with these types of control.

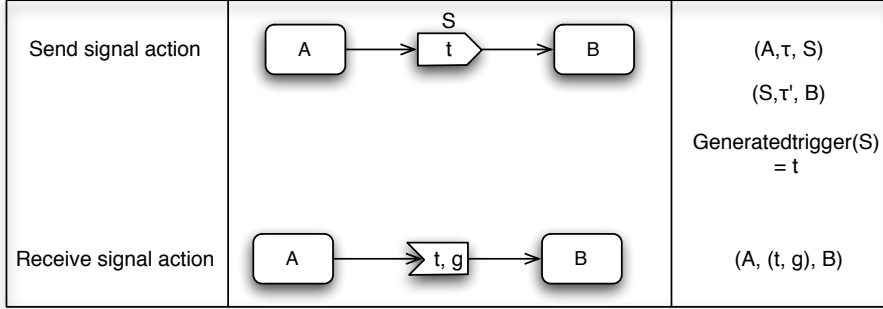


Figure 3: Accept and send event actions in UML.

- A **fork** is used to refer to the dividing of a flow into two or more parallel flows. As a result, actions are performed concurrently. The incoming and outgoing flows on a fork are represented in our model by one labelled flow  $(A, \tau, A')$  where  $A$  denotes the set of source actions of the incoming flow of the fork.  $A'$  is the set of the target actions of the outgoing flows of the fork.
- A **join** refers to the combining of two or more parallel flows into one flow. In case of a join, the incoming and outgoing flows on a join state are represented as one labelled flow  $(A, \tau, A')$ .  $A$  denotes the source actions of the incoming flows on the join.  $A'$  denotes the target action of the outgoing flow of the join.
- An **exclusive decision** (XOR) restricts the flow such that only one of the set of alternatives may be chosen during run-time. Exclusive decisions can be **data-based** or **event-based**.

In case of a data-based exclusive decision, the decision of alternatives is based on conditional expressions expressed on the different flows outgoing this decision. In our model as many labelled flows as there are outgoing flows are generated. These flows have the same source action. Each flow has an associated guard. The guards are mutually exclusive.

In case of an event-based exclusive decision, the decision of alternatives is based on an event that occurs at that point in the process. The specific event determines which flow that will be taken. Different intermediate event types can be modelled at an event-based exclusive decision. We already discussed these different event types above.

- An **exclusive merge** can be used for alternative flow. In our model as many labelled flows are created as there are incoming flows on the

merge.

- An **inclusive decision** represents a branching point where the decision of alternatives is based on guards expressed on the outgoing sequence flows. All combinations of flows can be taken. In our model as many labelled flows as there are outgoing flows of the inclusive decision point are generated. All of these flows have the same source action. Each flow has an associated guard and/or trigger. These guards and triggers do not have to be mutually exclusive.
- If the **inclusive gateway** is used as a **merge**, it will wait for all flows that were actually generated in a previous decision. Different labelled flows are created in our model. These flows all have the form  $(A, \tau, A')$ , where  $A'$  is the target action of the flow outgoing the merge and  $A$  is a possible combination of source actions of the incoming flows on the merge.
- In case of a **complex decision** an expression determines which of the outgoing flows will be chosen for the process to continue. In our model, there will be as many labelled flows created as there are outgoing flows on the decision. The flows can have associated guards.
- In case of a **complex merge**, there will be an expression that will determine which of the incoming flow will be required for the process to continue. Labelled flow(s) will be created in our model depending on this expression.

An action execution is created when all its control flow prerequisites have been satisfied.

**Definition 3** An **action trace**  $\kappa$  in a business process  $\omega = (A, F, L, \rho, \Lambda) \in \Omega$  is a  $n$ -tuple of action configurations, denoted  $\langle \alpha_1, \dots, \alpha_n \rangle$ , such that  $\alpha_1 = \{\rho\} \wedge \alpha_n \subseteq \Lambda \wedge \forall i \in \{1 \dots n - 1\} \exists \tau \in L : ((\alpha_i, \tau, \alpha_{i+1}) \in F \vee (\alpha_i, \tau, \alpha_{i+1}) \in F)$ .

Each  $\alpha_i \in \langle \alpha_1, \dots, \alpha_n \rangle$  is called an **active action configuration**.

**Example 1** The following action sequence is an action trace in the business process specified by the diagrams in Figure 1  $\langle \{\text{OrderTrip}\}, \{\text{EvaluateBestItinerary}\}, \{\text{CheckJourney}\}, \{\text{ReserveTickets}\}, \{\text{BookTickets}\}, \{\text{Change-Info}\}, \{\text{EvaluateBestItinerary}\}, \{\text{CheckJourney}\}, \{\text{ReserveTickets}\}, \{\text{BookTickets}\}, \{\text{Confirmation}\} \rangle$ . Each of the action singletons in this action trace, is an active action configuration.

**Definition 4** A **flow sequence**  $\mu$  in a business process  $\omega = (A, F, L, \rho, \Lambda) \in \Omega$  is a  $n$ -tuple of labels, denoted  $\langle \tau_1, \dots, \tau_n \rangle$  ( $n \geq 1$ ), such that  $\forall i \in \{1, \dots, n\} : \tau_i \in L$ .

**Definition 5** Given a flow sequence  $\mu = \langle \tau_1, \dots, \tau_n \rangle$  and a business process  $\omega = (A, F, L, \rho, \Lambda)$ :

**valid** is a binary relation such that **valid** $(\mu, \omega)$  iff  $\exists \kappa = \langle \alpha_1, \dots, \alpha_n \rangle$  where  $\kappa$  is an action trace in  $\omega$  and  $\alpha_1 = \rho$  and  $\forall i \in \{1, \dots, n\}$ :  $(\alpha_i, \tau_i, \alpha_{i+1}) \in F$ .

## 4 Conformance Relationships

The basic definitions given in the previous section enable the precise definition of conformance relations between different versions of the same business process. In order to define these relationships, we need an auxiliary definition.

**Definition 6** The **restriction  $\kappa_L$  of a sequence  $\kappa = \langle \alpha_1, \dots, \alpha_n \rangle$  to a set  $L$** , denoted  $\kappa_L$  is the sequence  $\kappa_L = \langle \alpha_1/L, \dots, \alpha_n/L \rangle$  where  $\alpha_i/L = \alpha_i \cap L, \forall i \in \{1, \dots, n\}$ .

The relations defined in the remainder of this section are specified between a business process and its new version. Before doing so, however, we need to be clear about what it means to be “a new version of a business process”. We will adopt a very broad view here. It includes changes to the business process (renaming, adding, removing or modifying actions and flows), but even more sophisticated changes can be envisioned, such as splitting a business process (each of the new business processes is then considered to be a new version of the original one), or combining two or more business processes into a single merged version.

The process described by business process models can be interpreted as a description of all the invocable actions that a client can use. Given this interpretation, we get a first conformance relationship between business processes:

*Each flow sequence which is invocable with respect to a given business process must also be invocable on the new version of this business process.*

**Definition 7** A business process  $\omega' = (A', F', L', \rho', \Lambda')$  is **invocation conform** to a business process  $\omega = (A, F, L, \rho, \Lambda)$  if and only if,

$\forall \mu : (\text{valid}(\mu, \omega) \Rightarrow \text{valid}(\mu, \omega'))$  and for the action traces  $\kappa$  corresponding to  $\mu$  in  $\omega$  and  $\kappa'$  corresponding to  $\mu$  in  $\omega'$ , it holds that  $\kappa = \kappa'_A$ .

**Example 2** Assume that the business process diagram in Figure 1 specifies the possible flow sequences that can be invoked on this process. The business process diagram in Figure 2 specifies the possible flow sequences that can be invoked on a refined version of the business process. The question can now be asked if the processes are invocable conform. This implies that every

flow sequence valid on the business process in Figure 1 is also valid on the business process in Figure 2. This is not the case in this example because the Pay Tickets action must always be executed before the tickets are booked or a cancel action is activated.

Another interpretation of the business process diagrams is to view them as a description of all observable action traces. The observable action traces are traces that might occur on this business process, however it is not guaranteed that all the resulting flow sequences are invocable. Given this interpretation, we get another conformance relation between business processes: *each action trace that is observable with respect to a new version of a business process must result under projection to the actions known by the original business process, in an observable action trace of the original business process.*

**Definition 8** A business process  $\omega' = (A', F', L', \rho', \Lambda')$  is **observation conform** to a business process  $\omega = (A, F, L, \rho, \Lambda)$  if and only if,  
 $\forall \kappa'$  action trace in  $\omega' \Rightarrow \kappa'_A$  is an action trace in  $\omega$ .

**Example 3** Consider again the business process diagrams shown in Figure 1 and in Figure 2. The question is whether each action trace with respect to the business process diagram in Figure 2 results under the projection to the actions known, in an observable action trace with respect to the business process in Figure 1. This relation is valid between both processes because the actions related to the payment of the tickets are not known in the original process shown in Figure 1.

## 5 Encoding into Description Logics

### 5.1 Introduction to DLs

DLs are a family of knowledge representation formalisms. These formalisms allow us to represent the knowledge of the world by defining the *concepts* of the application domain and then use these concepts and *roles* to specify properties of individuals occurring in the domain. The basic syntactic building blocks are atomic concepts (unary predicates), atomic roles (binary predicates) and individuals (constants). The expressive power of the language is restricted. It uses a small set of constructors to construct complex concepts and roles. Different combinations of constructors generate DL languages with different expressiveness. State-of-the-art DL systems such as, e.g., Racer [9], use the logic (or a logic based on) *SHIQ* [10]. Table 1 shows the syntax and semantics of this language. We will use this syntax to illustrate the encoding of our business process concepts to DLs.

Constructor	Syntax	Semantics	
atomic concept	$A$	$A^I \subseteq \Delta^I$	$\mathcal{S}$
universal concept	$\top$	$\top^I = \Delta^I$	
atomic role	$R$	$R^I \subseteq \Delta^I \times \Delta^I$	
transitive role	$R \in R_+$	$R^I = (R^I)^+$	
conjunction	$C \sqcap D$	$C^I \cap D^I$	
disjunction	$C \sqcup D$	$C^I \cup D^I$	
negation	$\neg C$	$\Delta^I \setminus C^I$	
value restriction	$\forall R.C$	$\{d_1 \mid \forall d_2 \in \Delta^I. ((d_1, d_2) \in R^I \rightarrow d_2 \in C^I)\}$	
exists restriction	$\exists R.C$	$\{d_1 \mid \exists d_2 \in \Delta^I. ((d_1, d_2) \in R^I \wedge d_2 \in C^I)\}$	
role hierarchy	$R \sqsubseteq S$	$R^I \subseteq S^I$	$\mathcal{H}$
inverse role	$R^-$	$\{(x, y) \mid (y, x) \in R^I\}$	$\mathcal{I}$
qualified	$(\geq n \ R.C)$	$\{d_1 \mid  \{d_2 \mid (d_1, d_2) \in R^I \wedge d_2 \in C^I\}  \geq n\}$	$\mathcal{Q}$
number restriction	$(\leq n \ R.C)$	$\{d_1 \mid  \{d_2 \mid (d_1, d_2) \in R^I \wedge d_2 \in C^I\}  \leq n\}$	

Table 1: Syntax and semantics of  $\mathcal{SHIQ}$ .

**Definition 9** [10] Let  $\mathbf{C}$  be a set of concept names and  $\mathbf{R}$  a set of role names with transitive roles  $\mathbf{R}_+ \subseteq \mathbf{R}$ . The set of  $\mathcal{SI}$ -roles is  $\mathbf{R} \cup \{R^- \mid R \in \mathbf{R}\}$ . The set of  $\mathcal{SI}$ -concepts is the smallest set such that every concept name is a concept and, if  $C$  and  $D$  are concepts and  $R$  is a  $\mathcal{SI}$ -role, then  $C \sqcap D$ ,  $C \sqcup D$ ,  $\neg C$ ,  $\forall R.C$  and  $\exists R.C$  are also concepts.

$\mathcal{SHI}$  is obtained from  $\mathcal{SI}$  by allowing for a set of role inclusion axioms  $R \sqsubseteq S$  where  $R$  and  $S$  are two roles.

$\mathcal{SHIQ}$  is obtained from  $\mathcal{SHI}$  by allowing, additionally, for qualifying number restrictions, i.e., concepts of the form  $\geq n \ R.C$  and  $\leq n \ R.C$ , where  $R$  is a simple (possibly inverse) role and  $n$  is a non-negative integer.

A simple role  $r$  iff it is neither transitive nor has transitive subroles.

The most important feature of DLs is their reasoning ability. One of the most important reasoning tasks is to check *subsumption* between concepts.

**Definition 10** A **general concept inclusion axiom (GCI)** is of the form  $C \sqsubseteq D$  for  $C, D$   $\mathcal{SHIQ}$ -concepts.  $C$  is called a *subconcept* or *child* of  $D$  and  $D$  is called a *superconcept* or *parent* of  $C$ .

A Tbox  $\mathcal{T}$  is a finite, possibly empty set of GCI's.

Another reasoning task is *Tbox coherence*. This check implies checking the satisfiability of all concept names mentioned in a Tbox without computing parent- and child-concepts. We will use this reasoning task to check our conformance relationships (see Section 5.3).

## 5.2 Encoding of Business Processes into DLs

In this section we present a possible encoding of business process into DLs. In previous work [15], we have shown how UML class diagrams, sequence diagrams and state machine diagrams can be translated to DLs and how

several inconsistencies in and between these diagrams can be detected using this formalism.

Because business processes focus on the sequence of flows, the translation presented here focusses on the encoding of the different flows defined by a business process.

**Action** Actions are translated into atomic DL concepts.

**Trigger on an action** A trigger can be associated to an action that represents the sending of the trigger. If an action  $B$  has a trigger associated,  $t = \mathbf{Generatedtrigger}(B)$ , this trigger is encoded by a concept. If this trigger has an operation  $op = \mathbf{Operationtrigger}(t)$  associated to it, it is encoded by the following concept description:  $\forall \mathbf{trigger.op} \sqcap (= 1 \mathbf{trigger})$ . A role  $\mathbf{trigger}$  is introduced that is used to define the operations triggered by an action.  $op$  is the concept representing the operation  $op$ .

**Label**  $\tau = (t, g)$  A label represents the triggering of a flow sequence and optional guard. A trigger can be explicitly specified. The trigger and optional associated operation are translated as specified in the previous item. Guards are also translated to DL concepts. How expressive such guards can be depends on the expressiveness of the DL used. In [15] (chapter 6, section 7) we explain how guards can be translated in DLs and how expressive these guards can be.

**Action Traces** Recall Definition 3 defining an action trace of a business process  $\omega = (A, F, L, \rho, \Lambda)$ . An action trace  $\kappa = (\alpha_1, \dots, \alpha_n)$  can be encoded in DLs using a binary role  $\mathbf{r}$  and by exploiting the subsumption relation of DLs. DL expressions of the form  $\mathbf{A} \sqsubseteq \exists \mathbf{r.A'}$  express a normal flow between an action  $A$  and an action  $A'$ . The concepts  $\mathbf{A}$  and  $\mathbf{A'}$  represent the actions  $A$  and  $A'$ , respectively.

Consider a sequence flow between an action  $A$  and an action  $B$  with a trigger associated to  $B$ . This is encoded by the DL expressions:  $\mathbf{A} \sqsubseteq \exists \mathbf{r.B}$  and  $\mathbf{B} \sqsubseteq \exists \mathbf{associatedTrigger.t}$ , where  $\mathbf{t}$  is the trigger concept as defined in the item **Trigger on an action**.

**Inserting Flow Information** Consider a sequence flow between an action  $A$  and an action  $B$  where the sequence flow has a label  $(t, g)$ . This flow is translated into a DL expression:  $\mathbf{A} \sqcap \mathbf{t} \sqcap \mathbf{g} \sqsubseteq \exists \mathbf{r.B}$ , where the concepts  $\mathbf{A}$ ,  $\mathbf{t}$ ,  $\mathbf{g}$  and  $\mathbf{B}$  represent the action  $A$ , the trigger  $t$ , the guard  $g$  and the action  $B$ , respectively.

If several action traces and flow sequences are taken into consideration, a certain action can be followed by several other actions depending on a certain decision pattern. As an example, consider the decision and



merging patterns shown in Table 5.2. We now show the translation of each pattern into DL expressions.

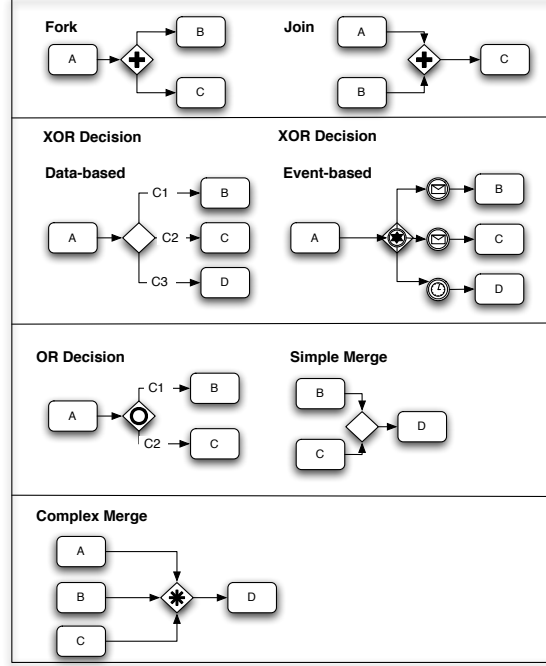


Table 2: Examples of decision or merging patterns using BPMN gateways.

**Fork** The fork example shown in the first row of Table 5.2 can be represented by the following DL expression:  $A \sqsubseteq \exists r.(B \sqcap C)$ . This expression expresses that the sequence flow outgoing the action  $A$  goes to both the actions  $B$  and  $C$ .

**Join** The join shown in the first row of the Table 5.2 can be encoded as the following DL expression:  $A \sqcap B \sqsubseteq \exists r.C$ . Both actions  $A$  and  $B$  must be reached to be able to transit to action  $C$ .

**Data-based XOR decision** The data-based exclusive decision example shown in the second row of Table 5.2 can be expressed by the following three DL expressions:  $A \sqcap C1 \sqsubseteq \exists r.B$ ,  $A \sqcap C2 \sqsubseteq \exists r.C$ ,  $A \sqcap C3 \sqsubseteq \exists r.D$ . The concepts  $C1$ ,  $C2$  and  $C3$  represent the guards  $C1$ ,  $C2$  and  $C3$ , respectively, written down on the different branches exiting the gateway. Because the decisions are exclusive the different conditions must be disjoint. This is expressed by the disjointness constraint  $C1 \sqcap C2 \sqcap C3 \sqsubseteq \perp$ .

**Event-based XOR decision** The event-based XOR decision in the second row of Table 5.2 is represented by the DL expression:

$A \sqcap M1 \sqsubseteq \exists r.B$ ,  $A \sqcap M2 \sqsubseteq \exists r.C$ ,  $A \sqcap T \sqsubseteq \exists r.D$ . The concepts  $M1$  and  $M2$  represent the message triggers specified by the message intermediate events. The concept  $T$  represents the time trigger and/or the time condition specified on the timer intermediate event. Because the decisions are exclusive the different triggers and conditions must be disjoint. This is expressed by the disjointness constraint  $M1 \sqcap M2 \sqcap T \sqsubseteq \perp$ .

**Inclusive decision** The inclusive or OR decision example shown in the third row of the Table 5.2 is encoded in the following two DL expression:  $A \sqcap C1 \sqsubseteq \exists r.B$ ,  $A \sqcap C2 \sqsubseteq \exists r.C$ . No further restriction is necessary because several of the flows can be taken.

**Simple merge** A simple merge gateway is a gateway that is used to bring together several multiple alternate flows. It is not used to synchronise concurrent flows, but to accept one among several alternate flows. The simple merge example shown in the third row of Figure 5.2 can be encoded in DLs as follows:  $B \sqsubseteq \exists r.D$ ,  $C \sqsubseteq \exists r.D$ . No further restrictions are specified for this pattern.

**Other merges** Several more complex merge patterns exist (see, e.g., [14]). As shown in the example in the fourth row of Table 5.2, a complex gateway can be used to define conditions required at the merge. We will not go into detail on all possible merge patterns but we refer to Section 8 for an in-depth discussion on the encoding of these patterns into DLs.

**Completeness of business processes** In case a business process represents the complete set of possible flow sequences and action traces, the different  $\exists r.X$  concepts, where  $X$  is a concept variable, are replaced by  $\forall r.X$ . If we also want to express that only one flow is possible starting from a certain action, we can add the restriction  $\sqcap ( = 1 \ r )$  to each  $\forall r.X$ .

**Completeness and disjointness of actions and triggers** If necessary completeness of the set of actions can also be enforced. Another restriction is the disjointness of the different actions and/or triggers of a business process. If the set of actions is complete and disjoint, it is guaranteed that two actions cannot be active at the same time.

### 5.3 Checking Conformance Relations using DLs

*Tbox coherence* can be used to check our conformance relationships. The different business processes are translated to GCI's using the translation introduced above.

For *invocation conformance* a *Tbox* can be constructed for the different flow sequences and action traces of both business processes. The original

version of a business process is considered to be complete. In this report, a business process or a set of business process elements is considered to be complete if it is assumed that the process or the set contains all relevant elements. The *Tbox* represented in the RACER Fragment 5.1 shows the translation of the business process modelled in Figure 1 and the one modelled in Figure 2. Statement (1) until (9) represent the original business process. This process is considered to be complete because we will check whether both processes are invocation conform. The statements (10) until (22) represent the refined business process. The resulting *Tbox* is incoherent which means that both process are not invocation conform.

- (1) (implies ordertrip (all r evaluatebestitinerary))
- (2) (implies evaluatebestitinerary (all r checkjourney))
- (3) (implies checkjourney (all r builditinerary))
- (4) (implies (and reject builditinerary) (all r changeinfo))
- (5) (implies changeinfo (all r evaluatebestitinerary))
- (6) (implies (and accept builditinerary) (all r reservetickets))
- (7) (implies reservetickets (all r booktickets))
- (8) (disjoint ordertrip evaluatebestitinerary checkjourney builditinerary
- (9) changeinfo accept reservetickets booktickets)
- (10) (implies ordertrip (some r evaluatebestitinerary))
- (11) (implies evaluatebestitinerary (some r checkjourney))
- (12) (implies checkjourney (some r builditinerary))
- (13) (implies (and reject builditinerary) (some r changeinfo))
- (14) (implies changeinfo (some r evaluatebestitinerary))
- (15) (implies (and accept builditinerary) (some r reservetickets))
- (16) (implies reservetickets (some r paytickets))
- (17) (implies (and paytickets paid) (some r booktickets))
- (18) (implies (and (not paid) paytickets) (some r canceltickets))
- (19) (equivalent transaction (or reservetickets paytickets booktickets))
- (20) (implies (and transaction cancel) (some r canceltickets))
- (21) (disjoint ordertrip evaluatebestitinerary checkjourney builditinerary
- (22) changeinfo accept reservetickets booktickets paytickets paid canceltickets cancel)

RACER Fragment 5.1: *Racer* expressions representing the business processes of Figure 1 and Figure 2 where the first process is complete.

For *observation conformance*, a *Tbox* can be constructed for the different action traces of both business processes. In this case the new version of the business process is assumed to be complete. The resulting *Tboxes* can be checked for coherence using a state-of-the-art DL system. If the *Tboxes* are coherent the processes are invocation, observation conform, respectively. If the *Tboxes* are not coherent the conformance relations are violated.

## 6 Tool Support

In previous work [15] and [16], we introduced a Poseidon [8] – a state-of-the-art UML CASE tool – plug-in called *RACOOoN*. Racer is used as underlying

reasoning system. This plug-in enables the checking and resolution of various inconsistencies that can arise between UML diagrams. Another functionality of this tool is to check whether certain inheritance relations between a superclass and its subclasses are preserved or whether certain behaviour preserving relations are preserved between a class and its evolved version. These relations are similar to the above defined conformance relations between business processes.

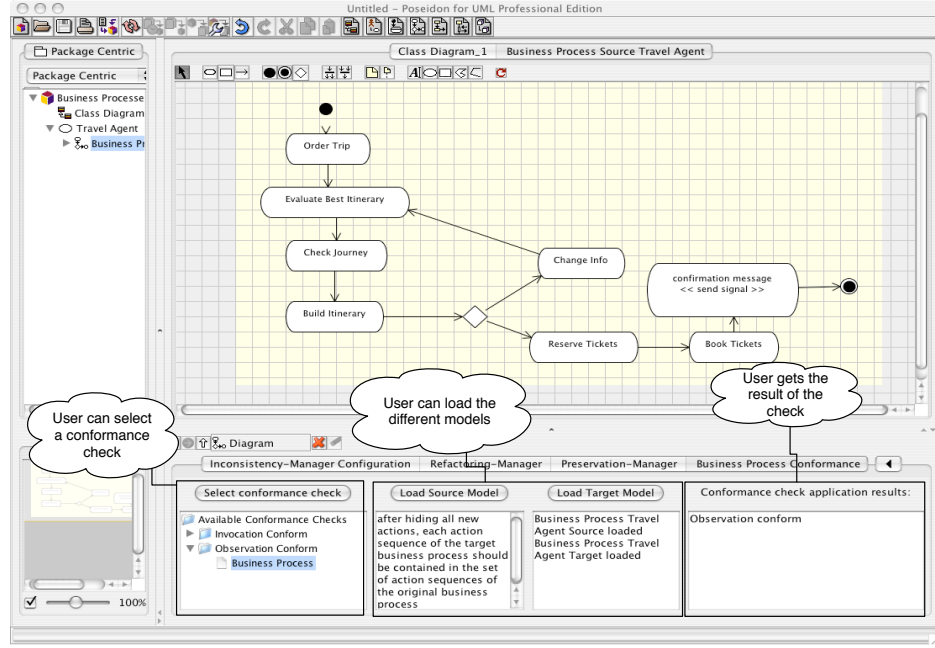


Figure 4: Screenshot of *RACoon*'s pane for conformance checking.

Figure 4 shows a screenshot of the business processes' conformance checking pane of *RACoon* incorporated in Poseidon. The user selects a conformance check, e.g., observation conformance. The user is asked for the source and target business process model. As a result these user-defined models (modelled using UML activity diagrams) are translated into a *Tbox*. This *Tbox* is loaded into Racer and the *Tbox coherence* reasoning task is started in Racer. Depending on the result returned by Racer, the user is informed whether the conformance relationship holds.

## 7 Related Work

As already mentioned in previous sections, the work presented in this report is related to our earlier research done in the context of inconsistency

management [15], [16]. Similar relations as the ones presented in this article can be defined between a subclass and its superclass or between an evolved class and its original class. All these relations are inspired by the notions of observable and invocation consistency between a superclass and its subclass using homomorphisms on state diagrams by Engels *et al.* [7]. These relationships have been further investigated by Schrefl *et al.* [13] using Petri nets.

Several semantics for UML 2.0 activity diagrams and for workflows have been presented. Examples are semantics based on Petri nets [17], on state machines [20] or semantics defined by means of a virtual machine [18]. All these approaches adopt a token-based semantics. We stress the formalisation of the *description* of a business process to be able to check *static conformance* relationships.

The work of Andersson [1] translates BPEL processes into finite state machines. The aim is to check whether web services can be composed. This work can be used in conjunction with our work. Using Andersson's approach the web service composition can be checked and our relationships can be used to check the conformance between composed services.

## 8 Discussion

The current disadvantage of conformance detection using *Tbox* coherence checking is the lack of feedback given to the user. If a *Tbox* is not coherent, the DL reasoning engine returns the set of unsatisfiable concepts. From this information only, we are not able to deduce, e.g., in case of an invocation conformance, which flow sequences conform and which not. To be able to inform the user correctly, i.e., to back-annotate the business process model with information concerning the cause of the non-conformance, two items must be further investigated. First, current DL tools, such as Racer, should give more and proper feedback on the cause of the satisfiability problem in case we check for *Tbox* coherence. Secondly, the necessary information for reconstructing a business process model from a DL translation must be stored (including lay-out information of the diagrams).

By using DLs for representing business processes, other properties and conformance checks besides our defined inconsistencies, can be checked. It is possible to check constraint equivalence, this boils down to equivalence of logical formula's. Furthermore, the constraints used in guards on flows can be checked for mutual exclusiveness.

In [14] a classification of workflow patterns is given into six categories. We will not discuss the content of these patterns in this report. For such a discussion we refer to [14]. However, we will discuss whether DLs are expressive enough to express these patterns and to verify their properties.

Table 3 describes which patterns can be supported by DLs (first column),

Pattern Category	Supported	Supported but ...	Not Supported
<b>Basic Control Flow</b>	Sequence Parallell Split Synchronization Exclusive Choice Simple Merge		
<b>Advanced Branching and Synchronization</b>	Multi-choice	Synchronizing Merge Multi-merge Discriminator	
<b>Structural</b>	Arbitrary Cycles	Implicit Termination	
<b>Multiple Instances</b>	without Synchronization	with a Priori Design Time Knowledge	with a Priori Runtime Knowledge without a Priori Runtime Knowledge
<b>State-based</b>	Milestone	Deferred choice Interleaved Parallell Routing	
<b>Cancellation</b>		Cancel Activity	Cancel Case

Table 3: Workflow Patterns and DLs.

which patterns can be supported to a certain extent (second column) and which patterns cannot be supported at all (third column).

All *Basic Control Flow* patterns described by van der Aalst *et al.* can be supported. Remark that in this report a *parallell split* is called a fork, a *synchronization* is called a join and an *exclusive choice* is called an exclusive decision. The encoding of these patterns into DLs is defined in Section 5.2.

The *Advanced Branching and Synchronization* pattern that is supported by DLs is the *multi-choice* pattern, called inclusive decision in this report. The other patterns of this category can also be supported by DLs but we need to make some remarks here. DLs have the open-world semantics which allows the specification of incomplete knowledge. This is very useful if we want to model business processes that are not yet complete. The constraints that are implicitly imposed by the *multi-merge* and *discriminator* pattern, i.e., the allowed sequences by this pattern, must be explicitly encoded in DL expressions. The same remark can be made for the *synchronizing merge* pattern. In this case we also need to be able to express that a certain concept representing an action (or, using the terminology of van der Aalst *et al.*, an activity) only can be instantiated once. Qualified number restrictions

can be used, but these express only how many times a certain flow may be instantiated. In case of actions we need to express so-called *cardinality restrictions* on concepts restricting the number of instances of a given concept. Such cardinality restrictions have been proposed in [2] and they show that the important reasoning tasks of such DLs stay decidable. However, this has never been implemented in any DL system.

The *Structural* pattern *Arbitrary Cycles* is supported by the translation into DL because a cycle results from a flow from an action  $A$  to  $B$  where there is already a flow going (indirectly) from  $B$  to  $A$ . The *Implicit Termination* pattern can be supported as long as there are DL expressions specified that express which activities can be terminated.

The *Multiple Instances* patterns involve several active instances of a certain action at the same time. The *Multiple Instances Without Synchronization* pattern implies that there is a facility to spawn off new threads of control, each independent of the other. There is no need to synchronize these threads [14]. With the translation presented in this report, several instances of the same action are allowed and without extra constraints, no synchronization is done. In case of the *Multiple Instances with a Priori Design Time Knowledge* pattern, the number of action instances is known at design time. To be able to support this pattern, we again need to express cardinality restrictions on concepts. The *Multiple Instances with a Priori Runtime Knowledge* pattern can in general not be supported because the number of instances is only known at run-time and the DL expressions represent a workflow or business process at design time. The *Multiple Instances without a Priori Runtime Knowledge* differs from the previous pattern because even while some of the instances are being executed or already completed, new ones can be created [14]. This cannot be supported by our design-time description of a workflow or business process.

In a *Deferred Choice* pattern, at the activation of one of the branches the other alternative branches are withdrawn. The choice is *delayed until the processing in one of the alternative branches is actually started, i.e., the moment of choice is as late as possible* [14]. This pattern can only be supported if certain states are explicitly modelled. However, we are not able to specify the withdrawal of activities in DLs. In case of the *Interleaved Parallel Routing* pattern the set of activities is executed in an arbitrary order. The order is decided at run-time and no two activities are active at the same moment. The only way to express this in DLs is to specify no sequence at all and to encode only the constraint expressing that only two activities are active at the same moment. What is not possible, for example, is to specify that if a certain action is already executed, it cannot be executed a second time. In a *Milestone* pattern the *enabling of an action depends on the case being in a specified state* [14]. This pattern is supported by our DL representation as long as the state on which an action depends can be represented by DL expressions.

In a *Cancel Activity* an enabled action is disabled. This can only be represented in our representation of it is explicitly modelled that some activities can be cancelled leading to the end of that thread. A *Cancel Case* removes all instances of a certain action. This cannot be expressed in DLs because it is not possible to quantify over particular sets of instances of an action.

From the analysis on how to represent the workflow patterns specified by van der Aalst *et al.* in DLs we conclude that nine of the twenty patterns can be fully supported by state-of-the-art DLs and DL systems. Seventeen of the twenty patterns can also be supported but need expressive DLs or the explicit modelling of certain constraints. Three of the twenty patterns cannot be represented by our DL translation at all.

As an issue of future work, the question arises which other kinds of conformance relationships can be defined between business processes. We should also investigate which kinds of relations are important in the context of process composition. How approaches based on Petri nets and our approach relate to each other and how both formalisms might be integrated remains an open question too.

## 9 Conclusion

Nowadays there is an increasing interest in business process modelling. Different notations and modelling languages can be used to describe business processes. Different versions of business processes can reside on different levels of abstraction and can be related to each other. The question arises what the relation is between the different versions of a business process.

In this report *we presented a lightweight formalisation for business processes*. Consequently, business processes are well-defined in our work. Based on this formalisation, we defined *two different conformance relationships* between different versions of a business process.

Tool support is needed to manage the business process models and for checking whether the conformance relations hold. In our opinion, this tool support must rely on a powerful formalism enabling the representation of business processes and the checking of the static conformance relations. We propose the declarative formalism DLs to support static conformance checking. In this report, we introduced DLs and its basic reasoning tasks. In a first step, we *showed how DLs can be used to express business processes*. Consequently, the meaning of the business processes and especially the sequence flows are determined and well-defined. Next, we showed *that the standard DL reasoning tasks are sufficient for the detection of our defined conformance relations*.

Finally, we implemented *the conformance relations in a tool which is a plug-in for the state-of-the-art CASE tool Poseidon relying on a state-of-*



*the-art DL system, Racer, and acting as a proof-of-concept of our ideas.*

This report is concluded by an overview of related work and a discussion on the representation in DLs of the workflow patterns presented in [14].

## References

- [1] Tobias Andersson. Conformance of behaviour protocols for web services. In Welf Löwe, editor, *Proceedings second Nordic Conference on Web Services*, pages 45–62. Växjö University Press, November 2003. 21
- [2] Franz Baader, Martin Buchheit, and Bernhard Hollunder. Cardinality restrictions on concepts. DFKI Research Report RR-93-48, Deutsches Forschungszentrum für Künstliche Intelligenz, 1993. Kaiserslautern, Germany. 23
- [3] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003. 4
- [4] BPML. Specification, business process modeling notation, version 1.0. <http://www.bpmn.org/>, September 2005. 3
- [5] Choreology. Description of abstract processes in spec. [http://www.choreology.com/external/WS\\_BPEL\\_issues\\_list.html-#Issue82](http://www.choreology.com/external/WS_BPEL_issues_list.html-#Issue82), September 2005. 3
- [6] Choreology. WSBPEL issues list. [http://www.choreology.com/-external/WS\\_BPEL\\_issues\\_list.html](http://www.choreology.com/-external/WS_BPEL_issues_list.html), September 2005. 3
- [7] Jürgen Ebert and Gregor Engels. Specialization of object life cycle definitions. Fachbericht Informatik 19/95, Universität Koblenz-Landau, Fachbereich Informatik, 1995. Koblenz. 21
- [8] gentleware.com. Poseidon. <http://www.gentleware.com/>, April 8 2005. 19
- [9] Volker Haarslev and Ralf Möller. RACER system description. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings first Int’l Joint Conf. Automated Reasoning (IJCAR2001)*, volume 2083 of *Lecture Notes in Computer Science*, pages 701–706. Springer, June 2001. Siena, Italy. 14
- [10] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive Description Logics. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Proceedings sixth Int’l Conf.*

- on *Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705 in Lecture Notes in Artificial Intelligence, pages 161–180. Springer, September 1999. 14, 15
- [11] IBM. Specification, business process execution language for web services, version 1.1. <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>, September 2005. 3
  - [12] Object Management Group. Unified Modeling Language 2.0 Superstructure Final Adopted Specification. <http://www.omg.org/cgi-bin/doc?ptc/03-08-02>, February 2004. 3
  - [13] M. Stumptner and M. Schrefl. Behavior consistent inheritance in UML. In Alberto H. F. Laender et al. editor, *Proceedings nineteenth Int'l Conf. Conceptual Modeling (ER 2000)*, volume 1920 of *LNCIS*, pages 527–542. Springer, 2000. 21
  - [14] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003. 10, 18, 21, 23, 25
  - [15] Ragnhild Van Der Straeten. *Inconsistency Management in Model-driven Engineering. An Approach using Description Logics*. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, September 2005. 4, 15, 16, 19, 21
  - [16] Ragnhild Van Der Straeten, Tom Mens, and Viviane Jonckers. A formal approach to model refactoring and model refinement. *Journal Software and Systems Modeling*, 2006. To be published. 4, 19, 21
  - [17] H.M.W. Verbeek and W.M.P. van der Aalst. Analyzing BPEL processes using petri nets. In D. Marinescu, editor, *Proceedings second Int'l Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 59–78. Florida International University, 2005. 21
  - [18] Valdis Vitolins and Audris Kalnins. Semantics of uml 2.0 activity diagram for business modeling by means of virtual machine. In Markus Aleksy and Patrick C. K. Hung, editors, *Ninth IEEE Int'l EDOC Enterprise Computing Conference*, pages 181–192. IEEE, September 2005. 21
  - [19] w3. Web Service Choreography Interface. <http://www.w3.org/TR/2002/NOTE-wsci-20020808/>, September 2005. 4

- [20] D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In F. N. Afrati and P. Kolaitis, editors, *Proceedings sixth Int'l Conf. Database Theory (ICDT)*, volume 1186 of *Lecture Notes in Computer Science*, pages 230–246. Springer, 1997. 21



## Appendix A

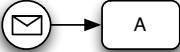




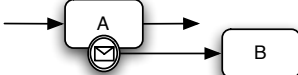
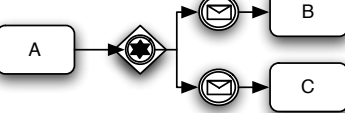


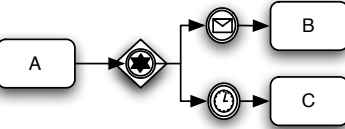
message start event		$(\rho, (t, g), A)$
timer start event		$(\rho, (t, g), A)$
rule start event		$(\rho, (t, g), A)$
link start event		$(\rho, (t, g), A)$
message intermediate event		$(A, (t, g), B)$
message intermediate event		$(A, (t, g), B)$
message intermediate event		$(A, (t, g), B)$ $(A, (t', g'), C)$
timer intermediate event		$(A, (t, g), B)$
timer intermediate event		$(A, (t, g), B)$
timer intermediate event		$(A, (t, g), B)$ $(A, (t, g), C)$

Table 4: Representation of start and intermediate events in our model.

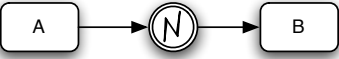
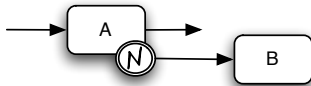
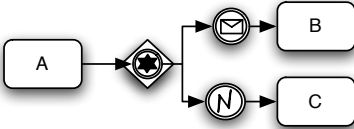
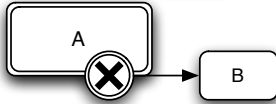
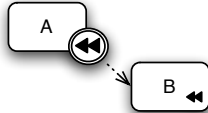



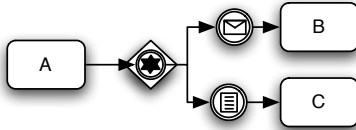
exception intermediate event		$(A, \tau, B),$ $\text{Generatedtrigger}(A) = e$
exception intermediate event		$(A, (t, g), B)$
exception intermediate event		$(A, (t, g), B)$ $(A, (t', g'), C)$
cancel intermediate event		$(A, (t, g), B)$
compensation intermediate event		$(A, (t, g), B)$
compensation intermediate event		$(A, \tau, B),$ $\text{Generatedtrigger}(A) = c$
rule intermediate event		$(A, (t, g), B)$
rule intermediate event		$(A, (t, g), B)$
rule intermediate event		$(A, (t, g), B)$ $(A, (t', g'), C)$

Table 5: Representation of intermediate events in our model.






message end event		$(A, \tau, \lambda),$ $\text{Generatedtrigger}(A)$ $= m$
exception end event		$(A, \tau, \lambda),$ $\text{Generatedtrigger}(A)$ $= e$
cancel end event		$(A, \tau, \lambda),$ $\text{Generatedtrigger}(A)$ $= c$
compensation end event		$(A, \tau, \lambda),$ $\text{Generatedtrigger}(A)$ $= co$
link end event		$(A, \tau, \lambda),$ $\text{Generatedtrigger}(A)$ $= l$

Table 6: Representation of end events in our model.