Towards a WPSL: A Critical Analysis of the 20 Classical Workflow Control-flow Patterns

Nataliya Mulyar[1], Wil M.P. van der Aalst[1,2], Arthur H.M. ter Hofstede[2], Nick Russell[2]

[1] Department of Technology Management, Eindhoven University of Technology
GPO Box 513, NL5600 MB Eindhoven, The Netherlands
{n.mulyar, w.m.p.v.d.aalst}@tm.tue.nl

[2] Faculty of Information Technology, Queensland University of Technology
GPO Box 2434, Brisbane QLD 4001, Australia
{a.terhofstede, n.russell}@qut.edu.au

Abstract. In 2000, after a comprehensive survey of tools and techniques for workflow management, 20 control-flow patterns were identified [5] and made these available through www.workflowpatterns.com. Since then, many commercial and academic workflow management systems have been evaluated using these patterns. Moreover, standards such as BPEL and XPDL have been evaluated and these evaluations have triggered improvements in them. Although the 20 workflow patterns have proven to be useful, the selection of these patterns was done in an ad-hoc manner and the description of the patterns in natural language has been rather ambiguous. Therefore, we propose a more analytical approach using a new Workflow Pattern Specification Language (WPSL). WPSL is independent of any implementation language utilized by contemporary workflow management systems. In this paper, we analyze the 20 original workflow patterns using WPSL, discuss the different variants of the patterns, and use WPSL to capture the detailed semantics of existing workflow management systems.

1 Introduction

A multitude of commercial workflow systems is available on the market today. These systems can be characterized by the wide range of unique features and capabilities of modelling languages they implement. The lack of the appropriate standards and limited adoptance by commercial vendors of standards proposed by the Workflow Management Coalition [27, 28] and the Object Management Group [13] explains why comparison of contemporary workflow systems, based on different concepts and paradigms, is a non-trivial task.

When automating business processes there is often a need to express specific process steps using the functionality available in workflow systems. Since the functionality of modeling languages employed by workflow systems differs significantly, the selection of an appropriate workflow system is hard to make. This problem has been addressed in [5, 1, 23, 22, 4] by formulating the requirements for workflow languages in form of patterns. The initial set of control-flow patterns has been inspired by capabilities of WFMS available for analysis. Note however, that the classical set of the control-flow patterns [5] turned out to be incomplete. This was no surprise as the original set of patterns was inspired by workflow systems available in the late nineties rather than obtained by systematic analysis. Moreover, current pattern definitions are defined informally and hence can be interpreted in various ways. In this paper, we address this issue using a systematic approach based on a comprehensive conceptual foundation.

To illustrate different capabilities of the modelling languages adopted by workflow systems, we briefly review the workflow systems YAWL, COSA, Staffware and Oracle BPEL PM. We show that even basic constructs such as the XOR-join and AND-join (the Petri-net representations of which are given in Figure 1), which are supported by the majority of workflow systems, are not interpreted in a uniform way.

In Petri-nets, activities are modelled by transitions and causal dependencies are modelled by places, transitions, and arcs. The XOR-join specifies that several distinct paths come together without synchronization (see a place $p1$ with ingoing arcs from transitions A and B in Figure 1(a)). The AND-join specifies the synchronization of multiple paths (see a transition C with incoming arcs from places $p1$ and $p2$ in Figure 1(b)).

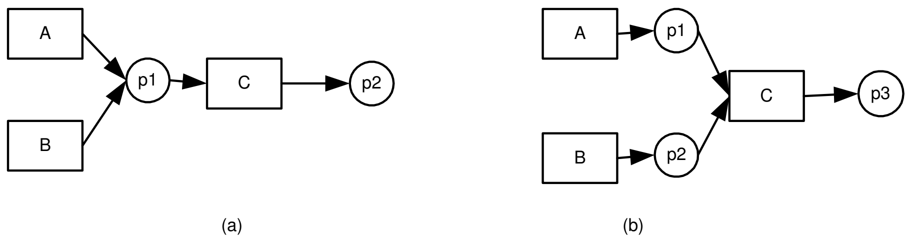


Fig. 1. XOR-join and AND-join in Petri Nets

The Petri-net based workflow system YAWL offers a direct support for the XOR-join and AND-join constructs (see Figure 2(a) and (b) respectively). Places in YAWL have unbounded capacity and to enable a task, each input place must contain at least one token to enable a task. In contrast to Petri nets, YAWL also allows the modelling of an OR-join (Figure 2(c)), which is a construct that may behave like an XOR-join or an AND-join (or a mixture of the two) depending on the context in which it is used. I.e. the OR-join waits untiil no additional tokens can arrive.

COSA is a Petri-net based workflow management system. The main building blocks of COSA are states, activities, and transitions, which are mapped directly on the concepts of Petri-nets as places, transitions, and arcs respectively. Figure 3 depicts the XOR-join and the AND-join on the nets (a) and (b) respectively. COSA can be considered as a safe Petri-net, which is characterized by at most one token being stored in a place at any given time. Hence, activities block when the output states are not empty. For example, activity A blocks when there is a

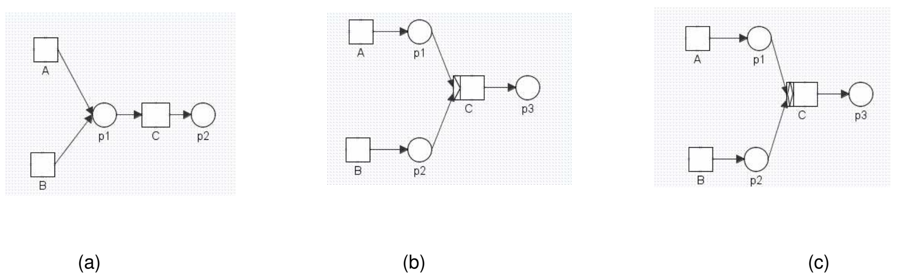


Fig. 2. Notation of XOR/AND/OR-joins in YAWL

token in place $s1$. As a result, COSA behaves differently from YAWL and ordinary Petri nets.

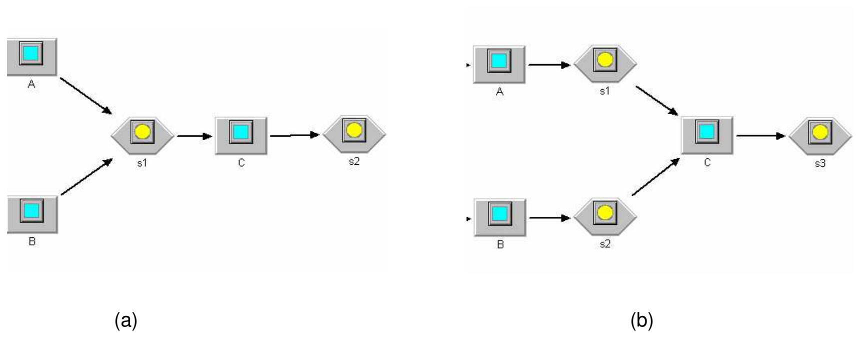


Fig. 3. Notations of XOR/AND-joins in COSA: activities block if not all output states are empty

The XOR-join and AND-join constructs in Staffware are denoted by means of *Step* and *Wait* objects (see models (a) and (b) of Figure 4 respectively). The *Step C* behaves as an XOR-join, i.e. it is triggered when *Step A* or *Step B* has completed. Only one instance of C can be active at a time. For instance, if *Step C* is still active and a new trigger arrived, this trigger will be ignored and all information associated with it will be lost. Note that this way a "race condition" is created. Just like COSA, Staffware forces intermediate states to be "safe". However, COSA enforces safeness by blocking activities, while Staffware simply removes excess triggers. The *Wait* object synchronizes left and top input arcs. This object may have only one left and up to 16 top arcs. However, the *Wait* object can be triggered only by a signal arriving at the left arc. When the object has been triggered, it starts evaluating the status of the top arcs. When all input arcs provided input, the *Wait* object executes. Note that when used in a loop, the *Wait* object behaves differently. For instance, if *Step B* is in the loop, but *Step A* is not, then for the repeated enabling of *Step C*, it is sufficient for *Step*

B to complete. However, if both steps *A* and *B* are in the loop, they both must complete in order to trigger *Step C* again.

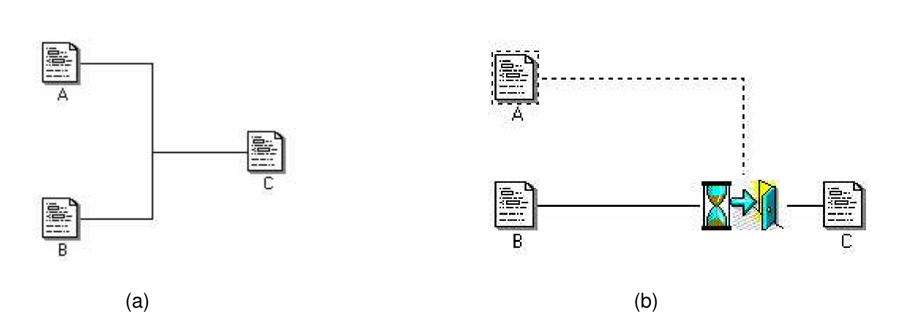


Fig. 4. Constructs for (a) an XOR-join and (b) an AND-join in Staffware. The default semantics of a step (e.g. *C*) is an XOR-join. A wait step (sand-timer symbol) needs to be inserted to synchronize flows (AND-join))

Currently there are many systems supporting BPEL [9]. In this paper, we selected Oracle BPEL as a representative of this class. Oracle BPEL PM implements the XOR-join and the AND-join by means of BPEL activities *switch* and *flow* respectively. In contrast to YAWL, Staffware and COSA, these constructs are applied within the structured workflow, i.e. every join is preceded by the corresponding split-construct. This way the corresponding processes are safe and the exceptional situations mentioned for COSA and Staffware cannot occur.

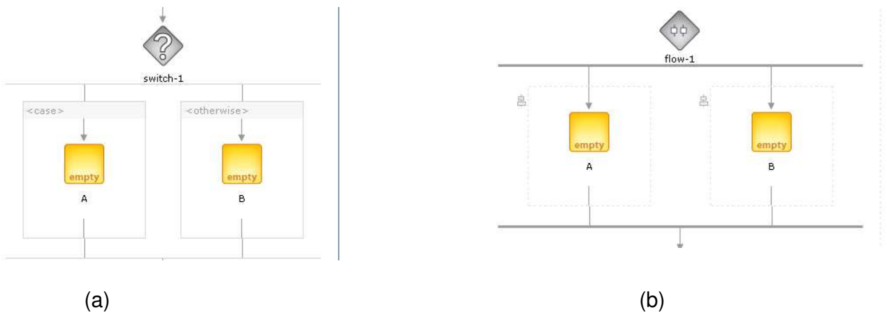


Fig. 5. Constructs for an XOR-join and an AND-joins in Oracle BPEL PM. Although BPEL is a textual language, a graphical interface is provided which directly reflects the BPEL code

We showed that implementation of XOR- and AND-joins in COSA, YAWL, Staffware and Oracle BPEL PM differ by the capacity of places, blocking behavior of activities, etc. Thus, *even simple constructs such as the XOR-join and AND-join are not interpreted in a uniform way in different WFM systems.* In order to distinguish between the differences identified in the modelling languages we propose a *Workflow Pattern Specification Language* (WPSL) that is formally defined and has a graphical notation that can be used as a means of analyzing and reasoning about the differences in the modelling languages. In addition, we apply WPSL to increase the degree of precision and completeness of the classical set of the control-flow patterns. Note that the proposed language can be applied to describe a wide variety of control-flow constructs.

The rest of this paper is organized as follows. Section 2 describes the goal, scope, basic premises and the structure of our WPSL. The syntax and semantics of WPSL are formally described in Section 2.2. Next, an analysis of the classical control-flow patterns is done by means of WPSL in Section 3. In Section 4 the paper concludes with a discussion of the lessons learned. Related work is outlined in Section 5. Finally, the conclusions and future work are discussed in Section 6.

2 Workflow Pattern Specification Language (WPSL)

In this section we introduce the main concepts of WPSL, followed by the formal definition of the language and an example illustrating its applicability.

2.1 Introduction to the language

The graphical Workflow Pattern Specification Language (WPSL) aims to capture the requirements of Process-Aware Information Systems (PAIS) expressed in the form of patterns. The current specification covers only the control-flow perspective and does not consider the data and organizational aspects. The main building block in WPSL is a *Task*, which is a basic modelling construct encountered in the process definition language of any Workflow Management System (WFMS). WPSL offers a means for visualizing all kinds of task variants that can be used for modelling a business process and provides different ways of combining these tasks for routing purposes.

The scope of the language is limited to a single case, covering the details of the case routing, while leaving the external relationships with other cases, processes, and external environments out of consideration.

There are two fundamental premises in regard to the WPSL semantics. First of all, all behavior in the modelled process is associated with active tasks, the execution of which changes the state of the modelled system. Secondly, the modelled process behaviors are message-driven and discrete. Discrete means that a modelled system is characterized by a certain state at every moment of time.

WPSL defines fundamental language constructs for representing the control-flow patterns. Figure 6 illustrates the semantics areas of WPSL and the hierarchical relationship between them.

On the bottom layer, there are three structural language constructs: a task, a channel, and a message. These are the main entities of which a generic workflow net (GWF-net) is composed. In terms of Petri nets, a task, a channel, and a message correspond to a transition, a place, and a token respectively. A *task* is an

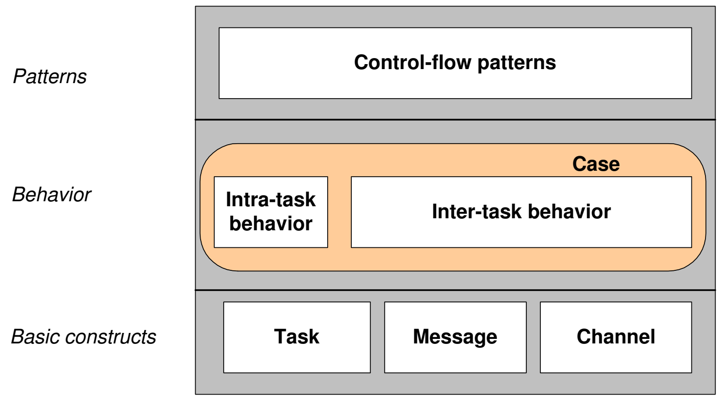


Fig. 6. Schematic semantics of WPSL

abstraction of an activity, characterized by a set of inputs and outputs, assigned to a certain resource. A *message* refers to the task input/output expressed in terms of a basic or complex data structure. A message is an abstract wrapper of the control data, used for routing purposes only, and/or the production data, i.e. any information (excluding control data) that can be manipulated as a discrete entity for the purpose of executing a certain activity. Note that by using messages we abstract from the actual data contained in the messages. A *channel* connects tasks and is used to convey messages.

The next layer is behavioral; it addresses the details of inter-task and intra-task behaviors. The intra-task area defines the variants of the task behavior based on the task properties, while the inter-task area addresses different ways of combining the structural entities together.

At the highest level of abstraction there is a set of extensible control flow patterns, which is obtained as a result of the interplay between the inter- and intra-task behaviors. A *control-flow pattern* is a three-part rule, which expresses a relation between a certain context (the lifecycle of a single case), a problem (addressing the behavioral aspect of the task routing), and a solution (expressed in terms of the structural entities).

In the WPSL structure depicted in Figure 6 we have introduced only the main concepts. Now we will present a detailed description of WPSL, all elements of which are graphically depicted in Figure 7.

Tasks send and receive messages to/from channels via *ports*, which play the role of message gates. Ports producing messages are called *output ports*. Ports consuming messages are called *input ports*. Every input port is mapped to a channel, which stores messages. We will refer to the combination of an input port and a channel to which this port is mapped as a *task input*, whilst we understand a *task output* to be the combination of an output port and a channel to which this port sends messages. We denote input and output channels as squares residing on the front and back edges of the task block respectively.

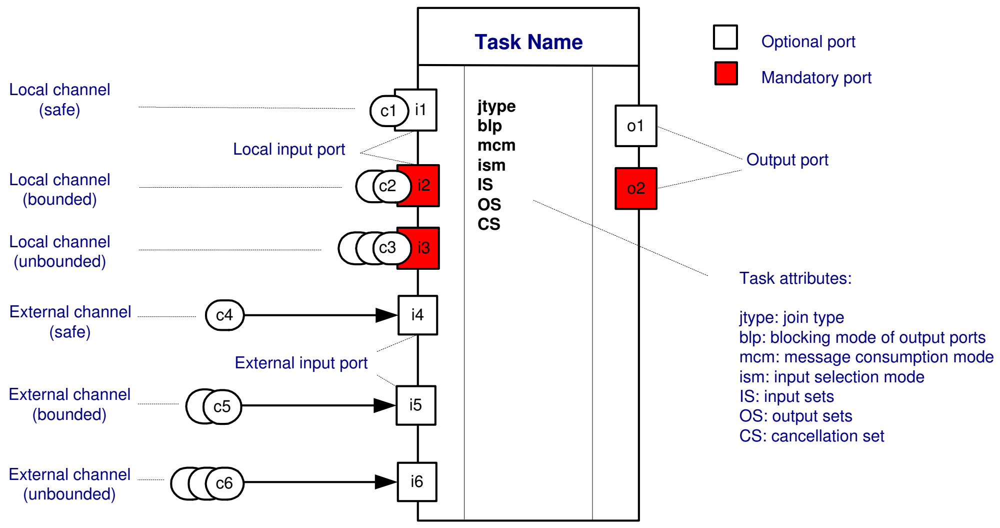


Fig. 7. The various notations for tasks, channels, and ports

Every GWF-net may have multiple input channels and output channels, however exactly one input channel and one output channels are involved in the initiation and termination of the process instance.

Every channel is characterized by a set of parameters, such as the maximal capacity, the minimal capacity, and the enabling status. The *maximal capacity* parameter defines how many messages the channel may hold at once. A channel with unlimited maximal capacity is called an *unbounded channel*, while a channel with limited maximal capacity is called a *bounded channel*. We will refer to bounded channels that are able to hold at most one message at a time, as *safe channels*. We denote safe, bounded and unbounded channels as a single, double and triple circles respectively. The direction of the arrows represents the message flow.

The *minimal capacity* parameter of a channel defines at least how many messages the channel must contain in order to make a port, consuming messages from this channel, enabled. A channel is *enabled* if its minimal capacity has been reached, otherwise the channel is said to be *disabled*.

Depending on the enabling status of a channel, an input port mapped to it can be either enabled or disabled. An input port is *enabled* if the channel to which this port is mapped is also enabled, otherwise the port is considered to be *disabled*.

Depending on the level of the visibility of the transferred messages and accessibility to them two types of channels can be distinguished. The *local channel* (relative to the task input) is used for the dedicated message transfer, i.e. when messages sent by a task-producer are to be received by a single dedicated task-consumer, and no other tasks may access the messages stored in this channel. The *external channel* (relative to the task input) is used for non-dedicated message transfer, i.e. when the message sent by a task-producer to the channel is to be consumed by one of several task-consumers which share access to the messages stored in this channel. To distinguish local and external channels graphically, we merge local channels with input ports.

Input ports which are mapped to local channels are called *local input ports*, while input ports mapped to external channels are called *external input ports*. The availability of messages in input channels is a property associated with input ports. A *mandatory input port* is a port, which must be enabled before the task may commence. An *optional input port* is a port, the enabling of which is not compulsory for the task commencement. The output ports can also be mandatory or optional. An output port produces one message upon task termination. A *mandatory output port* always produces a message upon task termination. An *optional output port* produces one message upon the task termination if and only if a data-based condition associated with this port has been satisfied (we do not elaborate on the data conditions, since in the context of this work we abstract from the data perspective). We denote optional and mandatory ports as white and dark squares respectively.

Every task has a set of properties that define the input and output logic of the task and the behavior of the task in an active state. A task is in the *active* state after it has commenced but before it has terminated. The *input sets (IS)* of a task define all possible sets of input ports, enabling of which is required for task commencement. The *input selection mode (ism)* of a task defines which set of enabled input ports is to be selected from the input sets.

The *message consumption mode (mcm)* of a task defines how many messages are to be consumed from the channels attached to the ports selected for consumption. In *minimal message consumption mode*, the minimal channel capacity is consumed from each of the channels, attached to the enabled ports selected for the message consumption. The non-consumed messages remain in the channels unless these channels are explicitly included in the task *cancellation set (CS)*, which specifies locations from which all messages are to be removed upon task termination. In *maximal message consumption mode*, all messages available in the channels attached to the enabled ports selected for the message consumption, are consumed at once.

The *output sets (OS)* of a task defines a set of output ports each of which will produce one message at the moment of task termination.

The *blocking mode (blp)* of output ports is a property defined for each task. In *blocked mode*, output ports may send messages to the output channels if and only if the maximal capacity of the corresponding channels has not been reached. If the maximal capacity of the channel has been reached, the output ports are blocked and wait until the required channel capacity becomes available. In *open mode*, output ports may send messages to the channels the maximal capacity of which has been reached, however these messages will be lost and will not modify the state of the channels.

The *cancellation set (CS)* defines which parts of the net should be emptied at the time of task termination. Emptying part of a GWF-net corresponds to removing messages from specified locations. Removing messages from a task corresponds to aborting execution of that task. We denote the cancellation set as a dashed-line attached to a task (for an example see an example in Figure 10).

Every task has a data-based *guard*, the status of which influences the enabling status of the task. Furthermore, the join logic of a given task is dependent on the *jtype* parameter, which specifies whether the processing of the task inputs is *local*, i.e. based on the messages currently available in the input channels, or *future*, i.e. postponed until no more new messages may arrive at the task inputs.

2.2 Formal Definition

In this subsection we will formalize the notions just introduced. First, we define a GWF-net.

Definition 1 (GWF-net). *A generic workflow net (GWF-net) N is a tuple (C, LC, EC, i, o, T, P, IP, OP, ManP, OptP, ptoc, psend, mincap, maxcap, blp, IS, ism, OS, mcm, CS, guard, jtype, F) where:*

- *C is a set of channels.*
- *$LC \subseteq C$ is a set of local channels.*
- *$EC \subseteq C$ is a set of external channels, such that LC and EC partition C, i.e. $LC \bigcap EC = \emptyset \wedge LC \bigcup EC = C$.*
- *$i \subset C$ is a set of input channels.*
- *$o \subset C$ is a set of output channels, such that $i \bigcap o = \emptyset$.*
- *T is a set of tasks.*
- *P is a set of ports.*
- *$IP : T \rightarrow \mathcal{P}(P)$ defines a set of input ports for each task.*
- *$OP : T \rightarrow \mathcal{P}(P)$ defines a set of output ports of a task $t \in T$, such that $\forall_{t_1,t_2 \in T} : (IP(t_1) \bigcup OP(t_1)) \bigcap (IP(t_2) \bigcup OP(t_2)) \neq \emptyset) \Rightarrow t_1 = t_2$ and $IP(t) \bigcap OP(t) = \emptyset$ for any task $t \in T$.*
- *$ManP : T \rightarrow \mathcal{P}(P)$ defines a set of mandatory ports for each task, such that $\forall_{t \in T} : ManP(t) \subseteq (IP(t) \bigcup OP(t))$*
- *$OptP : T \rightarrow \mathcal{P}(P)$ defines a set of optional ports for each task, such that $\forall_{t \in T} : ManP(t) \bigcap OptP(t) = \emptyset \wedge (ManP(t) \bigcup OptP(t) = IP(t) \bigcup OP(t))$*
- *$ptoc : P \rightarrow C$ maps every port to a channel, such that a single input/output port is mapped to a local channel, while multiple ports can be mapped to the same external channel.*

  $$\forall_{p_1,p_2 \in P} \forall_{c \in C} : ptoc(p_1) = ptoc(p_2) = c \Rightarrow (p_1 = p_2 \vee c \in EC)$$

  Let $\overline{p} = ptoc(p)$ for any $p \in P$, and generalize it for sets: $\overline{X} = \{ptoc(x) | x \in X\}$.
- *$mincap : C \rightarrow \mathbb{N}$ defines the minimal channel capacity. (Note that a channel with mincap(c)=0 behaves like a reset arc.)*

- *$maxcap : C \rightarrow (\mathbb{N}\backslash\{0\}) \bigcup \{\infty\}$ defines the maximal channel capacity.*

  if $maxcap(c) = \infty$, then the channel c is unbounded.

  if $maxcap(c) = k$, where $k \in \mathbb{N}\backslash\{0\}$, then the channel c is bounded.

  if $maxcap(c) = 1$ then the channel c is safe.

- *$blp : P \nrightarrow \{blocked, open\}$ defines the blocking mode of all output ports. Note that $dom(blp) = \bigcup_{t\in T} OP(t)$.*
- *$IS : T \rightarrow \mathcal{P}(\mathcal{P}(P))$ defines input sets for each task, specifying input ports the enabling of which is sufficient for the task commencement, such that $(\forall_{t\in T}\forall_{Q\in IS(t)} : (Q \subseteq IP(t) \wedge (ManP(t) \bigcap IP(t) \subseteq Q)))$*
- *$ism : T \rightarrow \{max, min, ran\}$ defines the input selection mode of a task.*

  max: select a "maximal" set of $IS(t)$, i.e. there is no a larger set Q of enabled input ports in IS(t) with respect to set inclusion.

  min: select a "minimal" set of $IS(t)$, i.e. there is no a smaller set Q of enabled input ports in IS(t) with respect to set inclusion.

  ran: select any set of enabled input ports in $IS(t)$.

- *$OS : T \rightarrow \mathcal{P}(\mathcal{P}(P))$ defines output sets of a task specifying what output ports are to produce messages upon the task termination, such that $(\forall_{t\in T}\forall_{Q\in OS(t)} : (Q \subseteq OP(t) \wedge (ManP(t) \bigcap OP(t) \subseteq Q)))$*
- *$mcm : T \rightarrow \{min, max\}$ defines the message consumption mode, i.e. how many messages are to be consumed from the enabled inputs selected according to $ism(t)$ for the given task $t \in T$.*

  min: consume the number of messages specified by the minimal capacity parameter of the channel.

  max: consume all messages available in the channel.

- *$CS : T \rightarrow \mathcal{P}((C \bigcup T) \setminus \{i, o\})$ specifies the task cancellation set, i.e. what additional messages are to be removed by emptying a part of the workflow.*
- *$guard : T \rightarrow Bool$ defines the status of the data-based task guard, which influences the enabling status of the task. (Note that the signature of this function might be misleading, since the dependency on data elements is missing due to the abstraction from the data perspective. Given a task t, $guard(t)$ may evaluate to $true$ or $false$ depending on the data values at the moment of evaluation.)*
- *$jtype : T \rightarrow \{local, future\}$ specifies whether the processing of the task inputs is local, i.e. based on the messages currently available in the input channels, or future, i.e. postponed until no more new messages may arrive at the task inputs.*
- *$F = \{(c, t) \in C \times T | c \in \overline{IP(t)} \bigcup \{(t, c) \in T \times C | c \in \overline{OP(t)}\}$ is the flow relation.*
- *every node in the graph $(C \bigcup T, F)$ is on the directed path from some $c_1 \in i$ to some $c_2 \in o$, i.e.*

$$(\forall_{x \in C \bigcup T} \exists_{c_1 \in i} \exists_{c_2 \in o} : (c_1, x) \in F^* \wedge (x, c_2) \in F^*)$$

where F^* is the transitive closure of F.

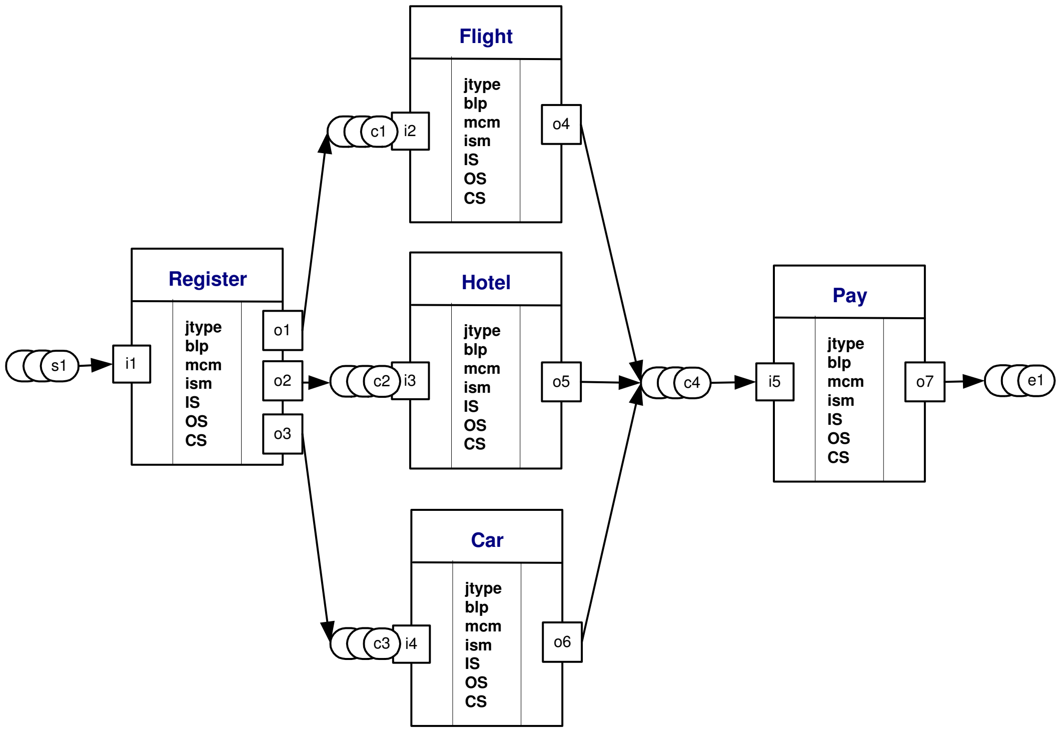
The Travel Agency example Let's consider an example of booking a business trip to illustrate the WPSL notation. The first WPSL specification presented in Figure 8 starts with task *Register* which enables tasks *Flight*, *Hotel* and/or *Car*. Task *Pay* is executed each time one of the three tasks (*Flight*, *Hotel* and *Car*) completes. In this graphical notation the choice is made to use unbounded channels, i.e. channels which are able to store an unlimited number of messages at any moment of time. The minimal capacity *mincap* of all channels is set to 1, specifying that exactly one message is required from each channel to enable a port attached to the given channel. Although channels are able to store multiple messages, the message consumption mode *mcm* of all tasks is set to minimal, meaning that exactly one message (as specified by the minimal channel capacity) will be consumed by the corresponding ports, while the rest of the messages will be ignored and thus kept in the channels for the subsequent task enabling.

In order to show that as the result of the execution of task *Register* a message is sent to a single task (*Flight*, *Hotel* and *Car*) or their combination, the output set of task *Register*, i.e. $OS(Register)$, lists all set variants that may be enabled upon the termination of the considered task.

The enabling of all tasks in Figure 8 is based on the messages currently available in the channels. This is reflected by the *jtype* parameter, which is set to *local*.

The second WPSL specification shown in Figure 9 combines individual payments into one payment. Task *Pay* waits until each of the tasks enabled by *Register* completes. Note that task *Pay* does not synchronize incoming channels if and only if a flight, a hotel or a car is booked. However, if the trip contains two or three elements, task *Pay* is delayed until all have completed. This mechanism is reflected by the parameter *jtype* of task *Pay* which is set to *future*. Moreover, to indicate that the maximal set of input ports from the ones specified in the input sets $IS(Pay)$ is to be selected for the message consumption, the input selection mode $ism(Pay)$ is set to maximal. For instance, if tasks *Hotel* and *Car* were executed, i.e. the messages were placed in channels $c4$ and $c6$, then both ports $i5$ and $i7$ (attached to these channels respectively) will be enabled.

The third WPSL specification shown in Figure 10 enables all three tasks (*Flight*, *Hotel* and *Car*) but executes task *Pay* after the first task has completed. After the payment all running tasks are cancelled. In contrast to the two earlier specifications, this WPSL specification associates a non-empty cancellation set to task *Pay*. The cancellation set of task *Pay* contains all channels and tasks which will be emptied the moment the task completes. Graphically the cancellation set of a task is visualized as a dashed rectangle attached to the given task, the scope of which is also indicated by the attribute CS.



jtype(Register)=local
mcm(Register)=min
ism(Register)=min
IS(Register)={{i1}}
OS(Register)={{o1},{o2},{o3},{o1,o2},{o2,o3},{o1,o3},{o1,o2,o3}}
CS(Register)={}

jtype(Pay)=local
mcm(Pay)=min
ism(Pay)=min
IS(Pay)={{i5}}
OS(Pay)={{o7}}
CS(Pay)={}

mincap(s1)=1
mincap(c1)=1
mincap(c2)=1
mincap(c3)=1
mincap(c4)=1
mincap(e1)=1

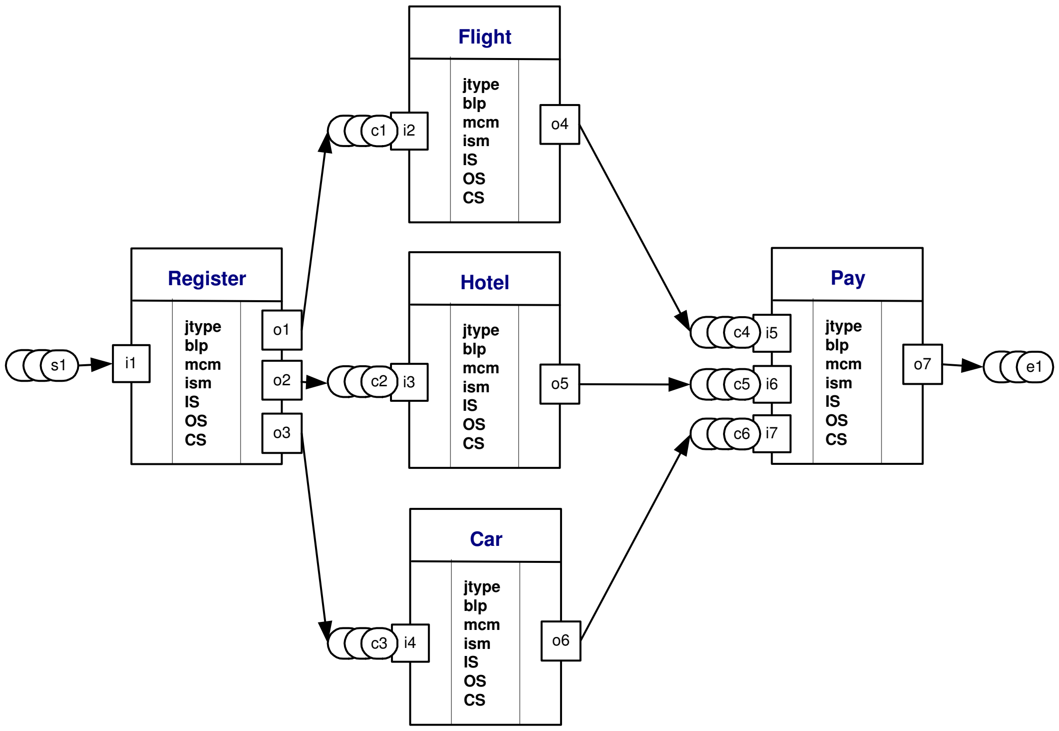
jtype(Flight)=local
mcm(Flight)=min
ism(Flight)=min
IS(Flight)={{i2}}
OS(Flight)={{o4}}
CS(Flight)={}

jtype(Hotel)=local
mcm(Hotel)=min
ism(Hotel)=min
IS(Hotel)={{i3}}
OS(Hotel)={{o5}}
CS(Hotel)={}

jtype(Car)=local
mcm(Car)=min
ism(Car)=min
IS(Car)={{i4}}
OS(Car)={{o6}}
CS(Car)={}

for all output ports p: blp(p)=open

Fig. 8. Task Pay executes each time one of the three preceding tasks completes



jtype(Register)=local
mcm(Register)=min
ism(Register)=min
IS(Register)={{i1}}
OS(Register)={{o1},{o2},{o3},{o1,o2},{o2,o3},{o1,o3}, {o1,o2,o3}}
CS(Register)={}

jtype(Pay)=future
mcm(Pay)=min
ism(Pay)=max
IS(Pay)={{i5},{i6},{i7},{i5,i6}, {i5,i7}, {i6,i7},{i5,i6,i7}}
OS(Pay)={{o7}}
CS(Pay)={}

jtype(Flight)=local
mcm(Flight)=min
ism(Flight)=min
IS(Flight)={{i2}}
OS(Flight)={{o4}}
CS(Flight)={}

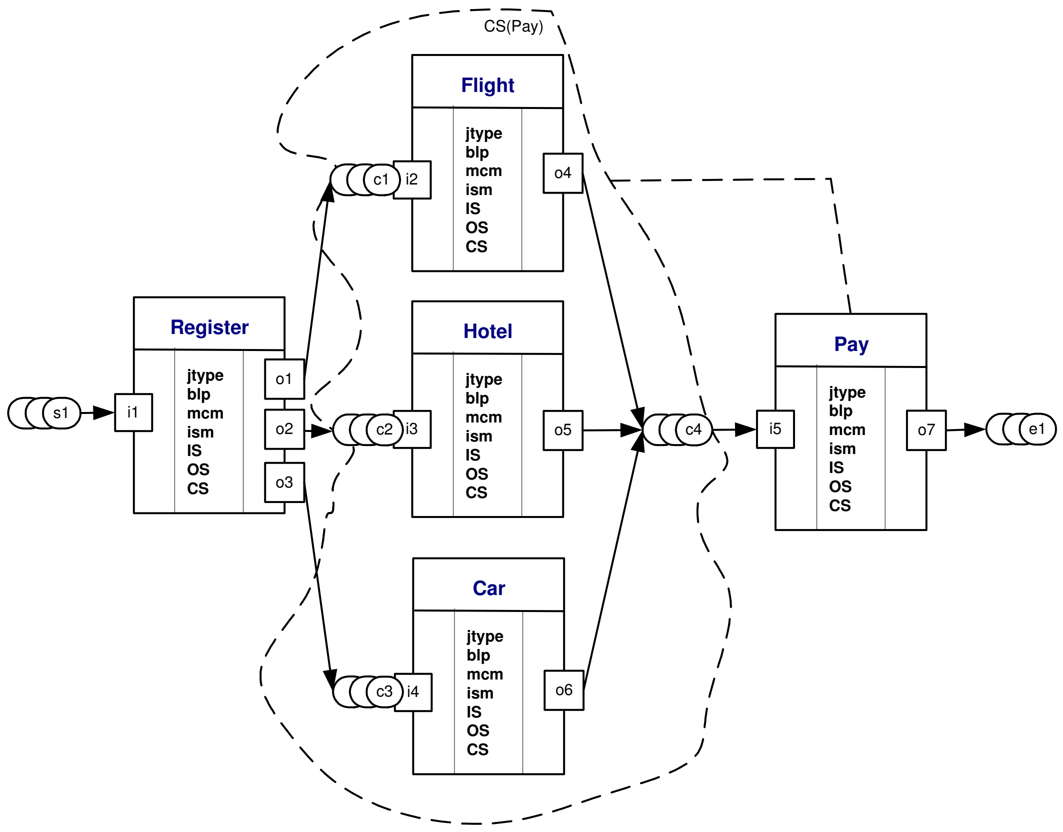
jtype(Hotel)=local
mcm(Hotel)=min
ism(Hotel)=min
IS(Hotel)={{i3}}
OS(Hotel)={{o5}}
CS(Hotel)={}

jtype(Car)=local
mcm(Car)=min
ism(Car)=min
IS(Car)={{i4}}
OS(Car)={{o6}}
CS(Car)={}

mincap(s1)=1
mincap(c1)=1
mincap(c2)=1
mincap(c3)=1
mincap(c4)=1
mincap(c5)=1
mincap(c6)=1
mincap(e1)=1

for all output ports p: blp(p)=open

Fig. 9. Task Pay executes only once, i.e. after all started tasks have completed



jtype(Register)=local
mcm(Register)=min
ism(Register)=min
IS(Register)={{i1}}
OS(Register)={{o1},{o2},{o3},{o1,o2},{o2,o3},{o1,o3}, {o1,o2,o3}}
CS(Register)={}

jtype(Pay)=local
mcm(Pay)=min
ism(Pay)=min
IS(Pay)={{i5}}
OS(Pay)={{o7}}
CS(Pay)={c1,c2,c3,c4, Flight, Hotel, Car}

jtype(Flight)=local
mcm(Flight)=min
ism(Flight)=min
IS(Flight)={{i2}}
OS(Flight)={{o4}}
CS(Flight)={}

jtype(Hotel)=local
mcm(Hotel)=min
ism(Hotel)=min
IS(Hotel)={{i3}}
OS(Hotel)={{o5}}
CS(Hotel)={}

jtype(Car)=local
mcm(Car)=min
ism(Car)=min
IS(Car)={{i4}}
OS(Car)={{o6}}
CS(Car)={}

mincap(s1)=1
mincap(c1)=1
mincap(c2)=1
mincap(c3)=1
mincap(c4)=1
mincap(e1)=1

for all output ports p: blp(p)=open

Fig. 10. Task Pay executes only once, i.e. after the first task has completed

2.3 Semantics

Definition 1 specifies the syntax of the GWF-net in mathematical terms, however it does not give any semantics. For this purpose, we define state space and state transitions.

The state space of GWN-net consists of a collection of messages, which serve as wrappers for data [1]. In order to deal with identical messages which may accumulate in channels we use bags also known as multi-sets. The state of a channel is represented by a multi-set of messages. In order to define the state space, we first introduce some notations.

Notation Let's denote input ports and output ports of a task $t \in T$ as $\bullet t$ and $t\bullet$, and input channels and output channels of the task as $\overline{\bullet t}$ and $\overline{t\bullet}$ respectively, such that:

$\bullet t = IP(t)$
$t\bullet = OP(t)$
$\overline{\bullet} t = \overline{\bullet t}$
$t\overline{\bullet} = \overline{t\bullet}$

A bag over alphabet A is a function from A to the natural numbers $\mathbb{N}$. For some bag X over alphabet A and $a \in A$, $X(a)$ denotes the number of occurrences of a in X, and is referred to as the cardinality of a in X. $[]$ denotes the empty bag, $[a, a, b]$ and $[a^2, b]$ denote the bag containing two a's and one b. Let $\mathcal{B}(A)$ denote the set of all bags over A. The sum of two bags X and Y, denoted $X \biguplus Y$, is defined as $[a^n | a \in A \wedge n = X(a) + Y(a)]$. The difference of X and Y, denoted as $X - Y$, is defined as $[a^n | a \in A \wedge n = max((X(a) - Y(a)), 0)]$. The size of the bag is denoted as $size(X) = \sum_{a \in A} X(a)$. The restriction of X to some domain $D \subseteq A$, denoted as $X \upharpoonright D$, is defined as $[a^{X(a)} | a \in D]$. Restriction binds more strongly than sum and difference (note that the binding of sum and difference is left-associative). Bag X is a sub-bag of Y, denoted as $X \subseteq Y$, iff for all $a \in A, X(a) \leq Y(a)$. $X \subset Y$ iff $X \subseteq Y$ and for some $a \in A, X(a) < Y(a)$. Note that any finite set of elements from A also denotes a unique bag over A, namely the function yielding 1 for every element in the set and 0 otherwise. Therefore, finite sets can be also used as bags. If X is a bag over A and Y is a finite subset of A, then $X - Y, X \biguplus Y, Y - X, Y \biguplus X$ yield bags over A. Let $set(x)$ denote a function which transforms a bag $x \in \mathcal{B}(A)$ into a set, such that $set(x) = \{a \in A | x(a) \geq 1\}$.

State space

Definition 2 (State space). *Let N=(C, LC, EC, i, o, T, P, IP, OP, ManP, OptP, ptoc, psend, mincap, maxcap, blp, IS, ism, OS, mcm, CS, guard, jtype, F) be a GWF-net. A workflow state s is a multi-set over the channels and tasks, i.e. $s \in S$, where $S = \mathcal{B}(C \bigcup T)$ is the state space of N.*

[1] Note that in this specification we do not consider the data perspective.

Whenever there is a need to refer to a set of locations (i.e. either channels or internal task states) marked in state s by messages, we will use the function $set(s) = \{x \in C \bigcup T | x \in s\}$.

Let's consider the task lifecycle visualized in Figure 11. This figure shows the internal structure of a task t. Note that in a state $s \in S$, there is a token in $Active_t$ if and only if $t \in S$. A task is considered to be active if its internal state is marked by a message, i.e. after the task has commenced but before it has terminated.

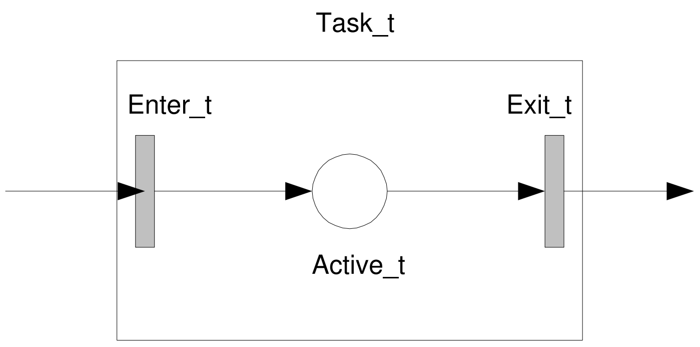


Fig. 11. The internal task states

Definition 3 (Task enabling). *Let N=(C, LC, EC, i, o, T, P, IP, OP, ManP, OptP, ptoc, psend, mincap, maxcap, blp, IS, ism, OS, mcm, CS, guard, jtype, F) be a GWF-net. The boolean function $enable(t, s)$ evaluates to true if and only if for a task $t \in T$ in state $s \in S$ the following set of conditions is satisfied:*

- *The task guard is satisfied:*
$$guard(t) = true$$
- *One of the input sets is enabled:*
$$(\exists_{Q \in IS(t)} : Q \subseteq \{p \in \bullet t | s(\overline{p}) \geq mincap(\overline{p})\})$$
- *Let S' be a set of states reachable from s (assuming some reachability relation). If $jtype(t) = future$, then $\nexists_{s' \in S'} s \restriction (\overline{\bullet} t) \subset s' \restriction (\overline{\bullet} t)$.*

Let's clarify the semantics of the message consumption/production using a task with multiple input and output ports connected to single external input and output channel respectively. This situation is depicted in Figure 12.
To illustrate the semantics, we consider two cases:

- If $IS(Task1) = \{\{i1\}, \{i2\}\}$ and $OS(Task1) = \{\{o1\}, \{o2\}\}$, then a single message is consumed from the channel $c1$ and a single message is produced to channel $c2$.
- However, if $IS(Task1) = \{\{i1, i2\}\}$ and $OS(Task1) = \{\{o1, o2\}\}$, then there is an asymmetry in the message consumption/production, i.e. a single message

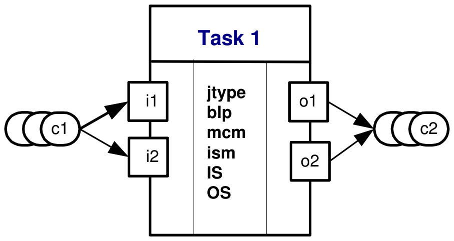


Fig. 12. Message consumption/production

is consumed from channel $c1$ and two messages are produced to channel $c2$. Note that such an asymmetry has to do with rules for task enabling, message consumption and message production. If the minimal capacity of the channel $c1$ has been reached, both input ports $i1$ and $i2$ of *Task 1* become enabled. At a time of the task activation, the minimal capacity of the channel $c1$ is consumed once by input ports (either $i1$ or $i2$), while upon the task termination both output ports produce a message to the outgoing channel $c2$.

Enabling of a task t with $jtype(t) = future$ and multiple inputs which need to be synchronized (we will refer to such tasks as $OR - joins$) needs to be postponed until no more messages can arrive resulting in enabling of a larger number of input ports of the OR-join. Since enabling of the OR-join depends on the possible future states, its semantics is non-local. Non-locality of the semantics of the OR-join has been a subject for a debate, and as a consequence several approaches to handle non-locality semantics have been proposed. In [2] Kindler, Desel and van der Aalst address the problem of non-local semantics in the context of EPCs demonstrating that there is no sound formal semantics for EPCs that is fully compliant with the informal semantics of EPCs. In [15, 16] Kindler defines a non-local semantics of EPCs using techniques from fixed point theory and a pair of two corresponding transition relations. The proposed technique is claimed to be applicable for formalizing all kinds of non-local semantics. In [29] Wynn, Edmond, van der Aalst and ter Hofstede propose a general and formal approach to OR-joins in workflow using Reset-nets. The authors examine the concept of the OR-join in the context of the workflow language YAWL and propose an algorithmic approach towards determining OR-join enablement. Because the issue of non-local semantics of OR-joins is a subject for an investigation on its own, we consider this issue to be outside of the scope of this work. Thus, we assume that a suitable approach for dealing with non-local semantics of OR-joins is known. Therefore, we assume some S' in Definition 3, i.e. S' is the set of reachable states and if $jtype(t) = future$, then the enabling of t depends on this set S'.

State transitions Let's formalize the transitions possible in a given state by means of binding functions $binding_{enter}$ and $binding_{exit}$ corresponding to the task commencement, which brings a task from the disabled state to the active state, and the task termination, which brings a task from an active state back to the disabled state, respectively.

Definition 4. *Let N=(C, LC, EC, i, o, T, P, IP, OP, ManP, OptP, ptoc, psend, mincap, maxcap, blp, IS, ism, OS, mcm, CS, guard, jtype, F) be a GWF-net. The boolean function* $binding_{enter}(t, cons, prod, s)$ *evaluates to true if and only if the transition enter can occur for a task* $t \in T$ *in the state* $s \in S$*, while consuming the bag of messages cons and producing the bag of messages prod, and the following conditions are satisfied:*

- *The task t is enabled in the given state s:*

$$enable(t, s) = true$$

- *Messages to be consumed are present in the state:*

$$cons \subseteq s$$

- *There exists a set* $Q \in IS(t)$ *such that:*
  - *Messages are consumed from inputs of the task:*

$$set(cons) = \overline{Q}$$

  - *If the input selection mode is set to maximal, then a maximal set of enabled input ports of* $IS(t)$ *is selected, i.e. there is no a bigger set with respect to set inclusion:*

$$\textit{if } ism(t) = max, \ \textit{then} \, \forall_{Q' \in IS(t)}(Q \subseteq Q') \Rightarrow \exists_{p \in Q' \setminus Q} s(\overline{p}) < mincap(\overline{p})$$

  - *If the input selection mode is set to minimal, then a minimal set of enabled task inputs of* $IS(t)$ *is selected for the message consumption, i.e. there is no a smaller set with respect to the set inclusion:*

$$\textit{if } ism(t) = min, \ \textit{then} \, (\forall_{Q' \in IS(t)} : Q' \not\subset Q)$$

  - *If the input selection mode is set to random, then any set of enabled task inputs of* $IS(t)$ *can be selected for the message consumption.*

- *One message is created for the active task state:*

$$prod = [t]$$

- *The task is not active yet:*

$$t \notin s$$

- *The number of messages consumed from the selected task inputs is determined by the message consumption mode of the considered task. In the minimal message consumption mode, the number of messages required for enabling of the input is consumed, while the rest of the messages remain in the input channels. In the maximal message consumption mode all messages contained in the channels of the selected input ports are consumed. For all* $c \in set(cons)$*:*

$$(mcm(t) = min) \Rightarrow (cons(c) = mincap(c))$$

$$(mcm(t) = max) \Rightarrow (cons(c) = s(c))$$

Definition 5. *The boolean function* $binding_{exit}(t, cons, prod, s)$ *evaluates to true if and only if the transition exit can occur for a task* t *in state* s*, while consuming the bag of messages cons and producing the bag of messages prod, and the following conditions are satisfied:*

– *Task* t *is active in state* s*:*

$$t \in s$$

– *One message is consumed from the internal task state:*

$$cons = [t]$$

– *There exists a set* $Q \in OS(t)$ *such that potentially one message is produced for each of the selected output ports and the maximal channel capacity is respected (cf. blocking mode):*

$$prod' = [\overline{p}^1 | p \in Q]$$

$$\textit{if } blp(t) = blocked, \textit{ then } prod = prod' \textit{ and } \forall_{c \in t\overline{\bullet}}(s \biguplus prod)(c) \leq maxcap(c)$$

$$\textit{if } blp(t) = open, \textit{ then } \forall_{c \in t\overline{\bullet}} prod(c) = min(prod'(c), (maxcap(c) - s(c)))$$

Definition 6. *The boolean function* $binding^{CS}_{exit}(t, cons, prod, s)$ *yields true if and only if there exists a* $cons'$ *such that* $binding_{exit}(t, cons', prod, s)$ *yields true and for any* $x \in C \bigcup T$*:*

$$cons(x) = (s - cons') \bigcup prod(x) \textit{ if } x \in CS(t)$$

$$cons(x) = cons'(x) \textit{ if } x \notin CS(t)$$

I.e., messages are removed from all input channels of task t and from its cancellation set. This implies cancellation of tasks in the cancellation set which have not yet completed.

Definition 7. *Let N=(C, LC, EC, i, o, T, P, IP, OP, ManP, OptP, ptoc, psend, mincap, maxcap, blp, IS, ism, OS, mcm, CS, guard, jtype, F) be a GWF-net. The Boolean function* $binding(t, cons, prod, s)$ *yields true if and only if any of the following conditions holds:*

- *The enter part of a task is enabled:*

$$binding_{enter}(t, cons, prod, s)$$

- *The exit part of a task is enabled:*

$$binding^{CS}_{exit}(t, cons, prod, s)$$

Definition 8. *Let N=(C, LC, EC, i, o, T, P, IP, OP, ManP, OptP, ptoc, psend, mincap, maxcap, blp, IS, ism, OS, mcm, CS, guard, jtype, F) be a GWF-net, and* s_1 *and* s_2 *two workflow states of S.* $s_1 \rightarrowtail s_2$ *if and only if there are* $t \in T$*,* $cons, prod \in S$ *such that* $binding(t, cons, prod, s_1)$ *and* $s_2 = (s_1 - cons) \uplus prod$*.*

$\rightarrowtail$ defines a transition relation on the states of the given workflow. The reflexive transitive closure of $\rightarrowtail$ is denoted $\rightarrowtail^*$ and $R(s) = \{s' \in S | s \rightarrowtail^* s'\}$ is the set of states reachable from state s.

The state space S and transition relation $\rightarrowtail$ define a *transition system* $(S, \rightarrowtail)$ for a given GWF-net. This completes the formalization of GWF-nets. Using this formalization we can also reason about the correctness of GWF-nets. For example, we can generalize the well-known soundness property as shown in Definition 9.

Definition 9 (Soundness). *Let N=(C, LC, EC, i, o, T, P, IP, OP, ManP, OptP, ptoc, psend, mincap, maxcap, blp, IS, ism, OS, mcm, CS, guard, jtype, F) be a GWF-net.*

- *N has the option to complete iff for any state* $s \in \bigcup_{c \in i} R([c]) : \exists_{c \in o} [c] \in R(s)$.
- *N has no dead tasks iff for any* $t \in T$ *there is a state* $s \in \bigcup_{c \in i} R([c])$ *such that* $t \in set(s)$.
- *N has proper completion iff for any state* $s \in \bigcup_{c_1 \in i} R([c_1])$: $\forall c_2 \in o : (s \geq [c_2]) \Rightarrow (s = [c_2])$.

N is sound iff N has the option to complete, has no dead tasks, and has proper completion.

Note that three GWF-nets used in the Travel Agency example have option to complete and have no dead tasks. In contrast to the GWF-nets depicted in Figures 9 and 10, the GWF-net in Figure 8 has no proper completion because of multiple messages produced by task Pay to the end channel $e1$.

We could also formally define multiple instances of a task as it has been done in [3], however we chose not complicate things any further. In the next section, we consider control-flow patterns involving multiple instances of a task and for this we only introduce a graphical notation while abstracting from the formal definition.

3 Analysis of the classical Workflow Control Patterns

In this section, we analyze the classical set of control-flow patterns using the notation of WPSL. Each pattern can be addressed in several ways, depending on the context in which the given pattern is applied. Therefore, we offer the default WPSL notation, which captures the essence of the pattern; then we list variation points to illustrate alternative configurations of the considered pattern. Furthermore, for each pattern we show the precise notation adopted by Staffware and Oracle BPEL PM, and how those can be mapped to WPSL.
The list of variation points is shown below:

- **Channel characteristics**: *the channel capacity* defines how many messages the channel may store at once; *the channel positioning* specifies whether a local or external type of a channel can be used for the message transfer.
- **Blocking of output ports**: specifies whether output ports may produce messages in the open or blocked mode.

- **Message consumption mode**: defines whether all messages available in the channel are consumed at once (the maximal mode) or only the minimal channel capacity is consumed while leaving the rest of the messages in the channel (the minimal mode).
- **Input selection mode**: specifies whether the minimal, maximal or random set of enabled input ports from the perspective of the set inclusion can be selected for the message consumption.
- **Input sets**: specifies the logic over the input ports enabling of which is sufficient for task enabling.
- **Output sets**: specifies the logic over the output ports which will produce messages at a time of the task termination.
- **Cancellation set**: specifies tasks and channels from which messages must be removed at the time of the task termination.

3.1 WP1 -Sequence

Description An activity in a workflow process is enabled after the completion of another activity in the same process. [2]

Selected WPSL specification Figure 13 shows the Sequence pattern. In terms of WPSL task $t2$ is executed after the execution of task $t1$. Messages are transferred via a safe external channel $c1$. The blocking mode of an output port $o1$ of task $t1$ is chosen to be *open* to allow task $t1$ to complete if the channel $c1$ is non-empty. The enabling of task $t2$ is based on the messages currently available in the channel, i.e. *jtype* is set to *local*. The messages are consumed in the *minimal* message consumption mode. The choice of the input selection mode for this net is irrelevant, since the number of the input ports is limited to one, thus no option for the input selection is available. However, to make the specification complete, *ism* of $t2$ is set to minimal. For the sake of convenience input sets of task $t1$ and output sets of task $t2$ which may vary without influencing the behavior of a pattern are omitted in this net and in other nets considered further in this paper. This is one of many possible interpretations of the pattern.

Alternative WPSL specifications The variation points based on which alternative configurations of the Sequence pattern can be based are listed below:

- **Channel characteristics**. *Channel capacity*: the maximal channel capacity can be increased from safe to bounded or unbounded, depending on the number of messages the channel is able to store at once. If the minimal channel capacity $mincap(c1) = 1$ does not change, this notation is equivalent to the main notation. Note however, that by increasing the minimal channel capacity, the enabling of the input port connected to this

[2] The term 'activity' used in [5], from which the pattern definition is cited, should be interpreted as 'task'. We omit repeating this remark for the rest of the patterns, thus assuming that terms 'task' and 'activity' can be used interchangeably.

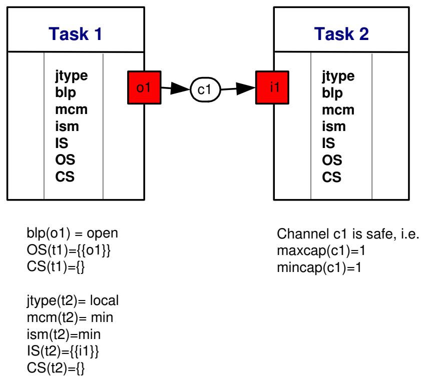


Fig. 13. WP1 - Sequence. Selected WPSL specification.

channel would be delayed until the specified minimal channel capacity is reached.
Channel positioning: external channel can be used with an additional requirement that no other tasks than the dedicated one are connected to this channel. However, for dedicated message transfer, the local channel type can be selected.

- **Blocking of output ports**. Output port $o1$ of task $t1$ can be set as *open* or *blocked*. If the safe channel $c1$ is not empty, the open output port $o1$ of task $t1$ is allowed to produce a message, however this message will be lost. If the output port is in the *blocked* mode, the completion of task $t1$ will be postponed until the capacity of the channel $c1$ is freed. Note that if the maximal capacity of the channel $c1$ is unbounded, the output port $o1$ of task $t1$ is always open and is never blocked since the maximal capacity of the channel cannot be reached.
- **Message consumption mode**. In the *minimal* message consumption mode the number of messages required to enable the channel (equivalent to the minimal channel capacity) is consumed, the rest of the non-consumed messages are stored in the channel for the subsequent task execution. To consume all messages available in the channel, the message consumption mode should be set to *maximal*.

Note that the initial pattern definition in [5] does not specify such attributes as the message consumption mode, the blocking of output ports, and the channel capacity, which makes the pattern subject to multiple interpretations.

Staffware implementation The Staffware model of the Sequence and its corresponding WPSL interpretation are presented in Figure 14 (a) and (b) respectively. The behavior of this pattern can be described by means of the WPSL attributes as follows. Messages in Staffware are transferred via a safe channel ($c1$ is chosen to be safe, i.e. $maxcap(c1) = 1$). The enabling of task B is based on the messages available locally ($jtype(B) = local$). The consumption of messages is performed in the minimal message consumption mode ($mcm(B) = min$). In Staffware messages can be sent to a channel even if the capacity of the channel has been reached. The second message will cancel the first one. Although Staffware has no notion of ports, the described behavior corresponds to the open mode of the output port ($blp(o1) = open$) in the WPSL terms.

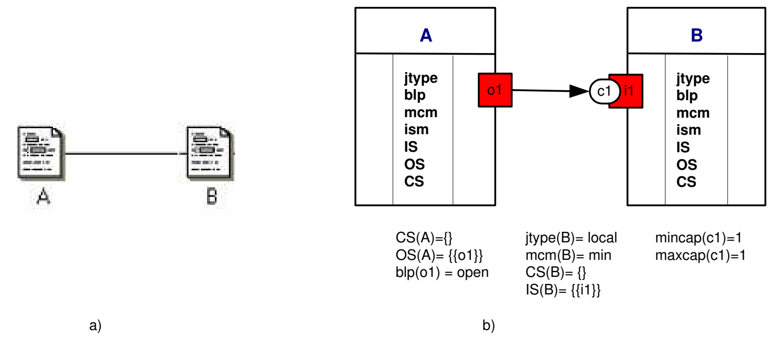


Fig. 14. Staffware implementation of WP1.

Oracle BPEL PM implementation The $\langle sequence\rangle$ construct presented in Figure 15, which also corresponds to the BPEL code listed below, allows the definition of the collection of tasks to be performed in lexical order.

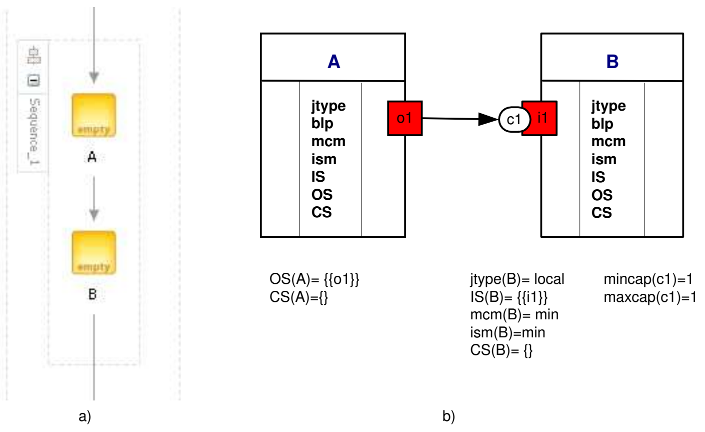


Fig. 15. Oracle implementation of WP1.

```
<sequence name="Sequence_1">
   <empty name="A"/>
```

```
    <empty name="B"/>
</sequence>
```

We will omit pieces of the XML-code further in this document, since they get larger for more complex patterns, and OracleBPEL diagrams adequately reflect the structure of the BPEL code.

The fact that BPEL is structured and acyclic, implies that it does not facilitate reasoning about attributes such as the input selection mode and the blocking mode of the ports. Activity B is enabled as soon at it is triggered by a message, therefore *jtype* of the corresponding WPSL task is set to local. It is sufficient to have a single trigger for enabling of a task, therefore the message consumption mode of task B is set to minimal.

3.2 WP2 -Parallel Split

Description A point in the workflow process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order.

Selected WPSL specification Figure 16 shows the Parallel Split pattern.

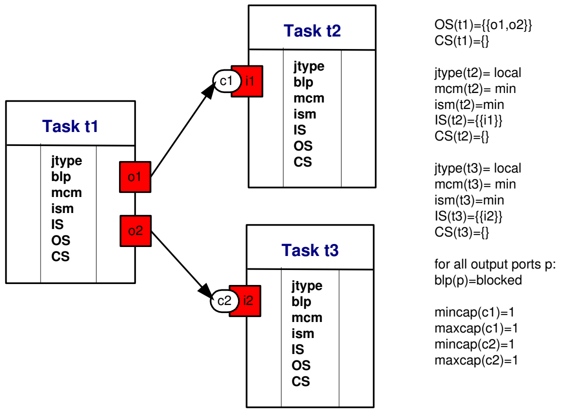


Fig. 16. WP2 - Parallel Split. Selected WPSL specification.

The execution of task $t1$ enables execution of tasks $t2$ and $t3$. As a result of the execution of task $t1$ messages must be produced for two outgoing channels $c1$ and $c2$, therefore ports $o1$ and $o2$ are denoted as mandatory (see the output set of task $t1$). Since tasks $t2$ and $t3$ are independent of each other, messages to them are sent via dedicated channels. The enabling of tasks $t2$ and $t3$ is based on the local availability of the messages in channels $c1$ and $c2$, which are chosen to be safe.

Alternative WPSL specifications The variation points based on which alternative configurations of the Parallel Split pattern can be based are listed below.

- **Channel characteristics**. *Channel capacity*: the capacity of channels $c1$ and $c2$ can be bounded or unbounded depending on the number of messages the channel is allowed to store at once. To specify that multiple messages are required to enable an input port(s) of a certain task, the minimal capacity of the corresponding non-safe channels must be set respectively.
  Channel positioning: instead of local channels, external channels can be used with an additional requirement that access to the messages stored in the channel is limited to a single dedicated task.
- **Blocking of output ports**. Depending on the type of channels used, output ports can be set to *blocked* or *open* mode. Setting the *blocked* mode for the ports producing messages to unbounded channels does not make sense, since the capacity of the channel will never be reached and an output port will never be blocked. However, both modes are applicable in the context of safe and bounded channels.
- **Input selection mode**. Since the core logic of the Parallel Split pattern is defined in the task $t1$, the setting of ism is less relevant for this task. Tasks $t2$ and $t3$ have a single input port, and there is no option for input selection. All modes for input selection are equivalent when applied to a task with a single input port. In the remainder of this paper, we will avoid further discussion on input selection mode for tasks with single input ports. However, to make the specification complete, the input selection mode for such tasks will be set to minimal.
- **Message consumption mode**. In *minimal* message consumption mode, the number of messages required to enable the channel (equivalent to the minimal channel capacity) is consumed, the rest of the non-consumed messages are stored in the channel for the subsequent task execution. To consume all messages available in the channel, the message consumption mode should be set to *maximal*.

Note that the original pattern definition in [5] does not specify the blocking mode of the output ports or any of the other issues mentioned above.

Staffware implementation Staffware supports the Parallel Split pattern directly as illustrated in Figure 17 (a) by means of a *Step* object with multiple outgoing arcs. Figure 17 (b) depicts the corresponding WPSL interpretation. Similar to the description of the Sequence pattern, task A corresponding to *Step A* is characterized by the open mode of the output ports, enabling based on the local message availability and the minimal message consumption mode. The split logic of *Step A*, which propagates a message to Steps B and C, is incorporated into the output set of the WPSL task A.

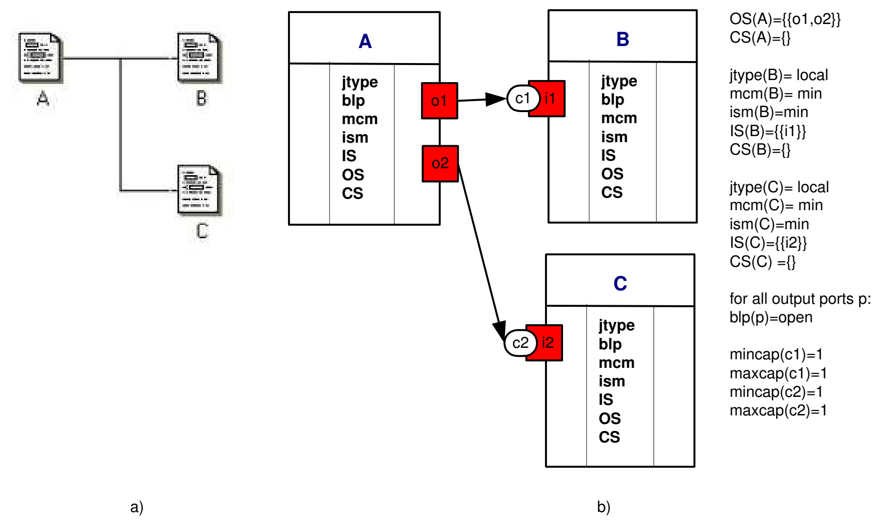


Fig. 17. Staffware implementation of WP2.

Oracle BPEL PM implementation The Parallel Split pattern is implemented in Oracle BPEL PM via the $\langle flow \rangle$ construct presented in Figure 18 (a). The corresponding WPSL interpretation is shown in Figure 18 (b). Note that the $\langle flow \rangle$ construct is of structured nature, therefore it supports the Synchronization pattern. The WPSL model can be logically decomposed into two patterns. As such, tasks *Start Flow*, A and B constitute the Parallel Split pattern, while the Synchronization pattern is composed out of tasks A, B, and *End Flow*. The split and join logic associated with the start and the end of the $\langle flow \rangle$ is reflected by the output set and input sets of the dummy-tasks *Start Flow* and *End Flow*. To denote that both branches in the $\langle flow \rangle$ construct must complete the input selection mode $ism(EndFlow)$ is set to *maximal*. The rest of the settings are the same as described in the Sequence pattern. Since execution of tasks in Oracle BPEL PM may abort, as a result of this no messages will be produced by these tasks. To capture this semantics and avoid blocking of the synchronizing task, the input ports of task *End Flow* must be made optional and the join logic must take into account future possible states. If all tasks in parallel branches fail to execute, no synchronization will be performed. Oracle BPEL PM handles this issue by propagating *true* and *false* tokens via all branches indicating the status of the task execution. An equivalent behavior is obtained in WPSL by direct connection between the start and the end of the flow.

3.3 WP3 -Synchronization

Description A point in the workflow process where multiple parallel subprocesses/activities converge into one single thread of control, thus synchroniz-

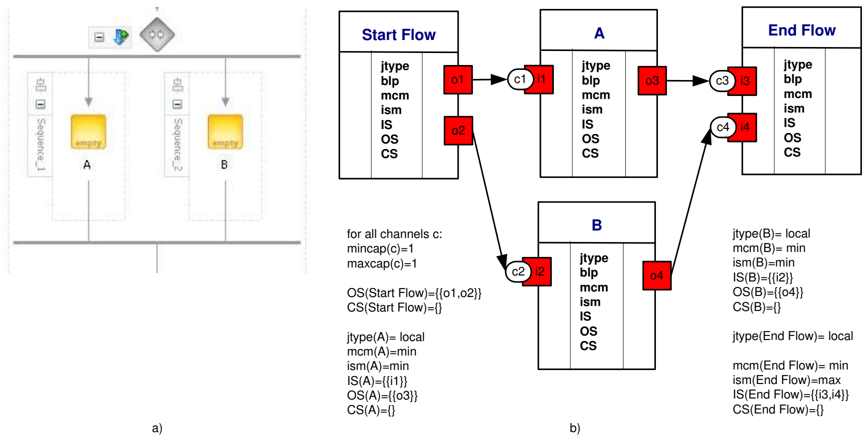


Fig. 18. Oracle implementation of WP2/WP3.

ing multiple threads. It is an assumption of this pattern that each incoming branch of a synchronizer is executed only once.

Selected WPSL specification Figure 19 depicts the Synchronization pattern.

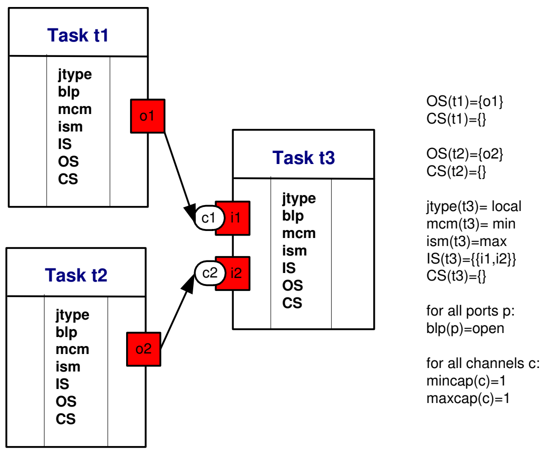


Fig. 19. WP3 - Synchronization. Selected WPSL specification.

Task $t3$ is enabled after the completion of both tasks $t1$ and $t2$. Messages produced by each of tasks $t1$ and $t2$ are sent to task $t3$ via dedicated channels. In order to show that inputs from both channels are required for enabling task $t3$, both input ports $i1$ and $i2$ of task $t3$ are chosen to be mandatory (see the input sets of task $t3$). Task $t3$, when enabled, consumes one message from each incoming channel as indicated by the *minimal* message consumption mode and safe type of the channels. The enabling of task $t3$ is based on the messages currently available in the channels.

Alternative WPSL specifications The points of variation on which alternative configurations of the Synchronization pattern can be based are the same as for the Parallel Split pattern. Note that the initial pattern definition in [5] does not give any indication about the blocking mode of output ports.

Staffware implementation The Synchronization pattern in Staffware is directly supported by the $Wait$ object, depicted in Figure 20 (a). The corresponding WPSL interpretation is shown in Figure 20 (b). It is a feature of the $Wait$ object to wait for all inputs which still may arrive, or in the context of a loop to execute immediately if an input associated with the left arc is available and no top inputs may become available any more. To specify this behavior input sets together with the input selection mode are set for the maximal input selection.

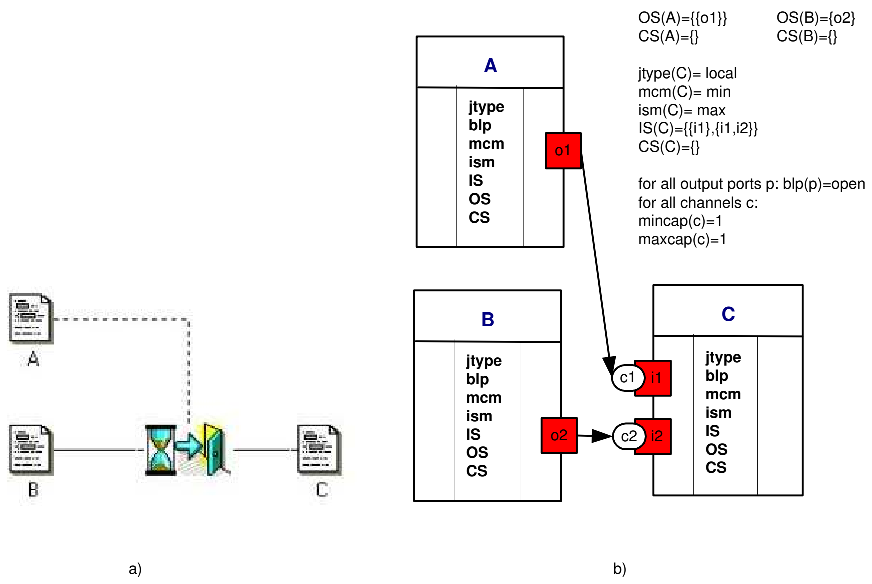


Fig. 20. Staffware implementation of WP3.

Oracle BPEL PM implementation Due to the structured nature of activities in Oracle BPEL, the Synchronization pattern cannot be separately depicted but is included in the $\langle flow \rangle$ construct illustrated in Figure 18. The WPSL description given for the Parallel Split pattern is also valid in the context of this pattern.

3.4 WP4 -Exclusive choice

Description A point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen.

Selected WPSL specification Figure 21 demonstrates the Exclusive Choice pattern.

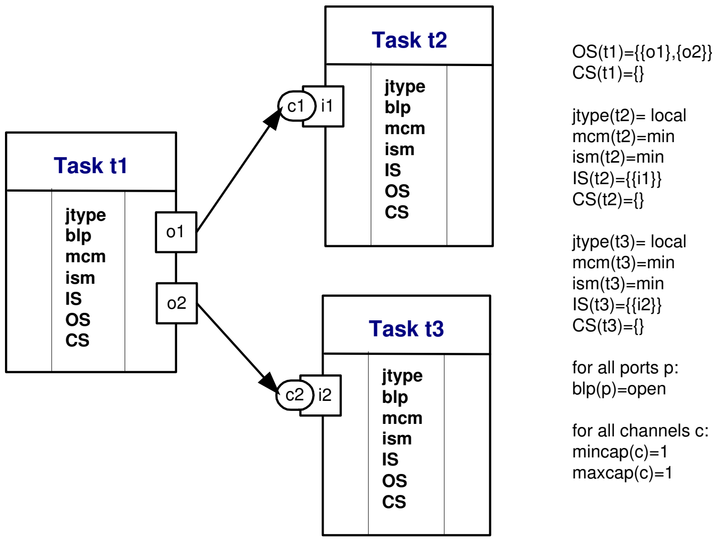


Fig. 21. WP4 - Exclusive choice. Selected WPSL specification.

Task $t1$ is followed either by task $t2$ or task $t3$. To show that the choice is to be made between task $t2$ and task $t3$, output ports of task $t1$ are selected to be optional and output sets of task $t1$ are made disjoint. A selected output port performs a dedicated transfer to a task via a safe channel. The enabling of task-recipients is based on the local availability of messages in the incoming channels.

Alternative WPSL specifications The points of variations on which alternative configurations of the Exclusive Choice pattern can be based are listed below.

- **Channel characteristics**. *Channel capacity*: The capacity of channels $c1$ and $c2$ can be bounded or unbounded depending on the number of messages the channel is allowed to store at once. To specify that multiple messages are required to enable an input port(s) of *Task 3* the minimal capacity of the corresponding non-safe channel must be set accordingly.
  Channel positioning: local channels, external channels or their combination can be used with an additional requirement that access to the messages stored in the external channel is limited to a single task.
- **Blocking of output ports**. Depending on the type of the channels used, output ports can be set to the *blocked* or *open* mode. The blocked mode is to be applied in combination with safe and bounded channels, while the open mode can be used with any kind of channels.

- **Message consumption mode**. The *minimal* and *maximal* message consumption modes can be used interchangeably to guarantee that the minimal or the full channel capacity is consumed by the ports attached to this channel.

The initial pattern definition does not specify the blocking mode of the output ports and the message consumption mode.

Staffware implementation Staffware supports this pattern by means of the binary *Decision* object illustrated in Figure 22 (a). The corresponding WPSL model is shown in Figure 22 (b). To indicate the binary choice between steps B and C, output sets of task A are made disjoint. The output ports of task A produce messages in the open mode.

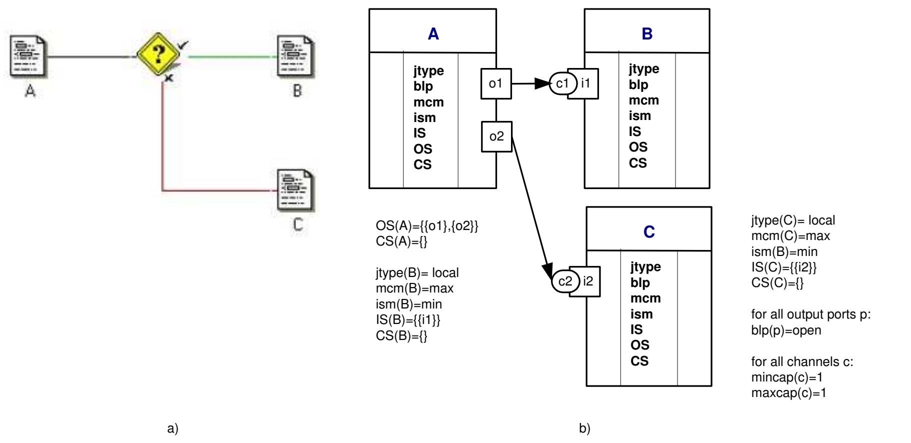
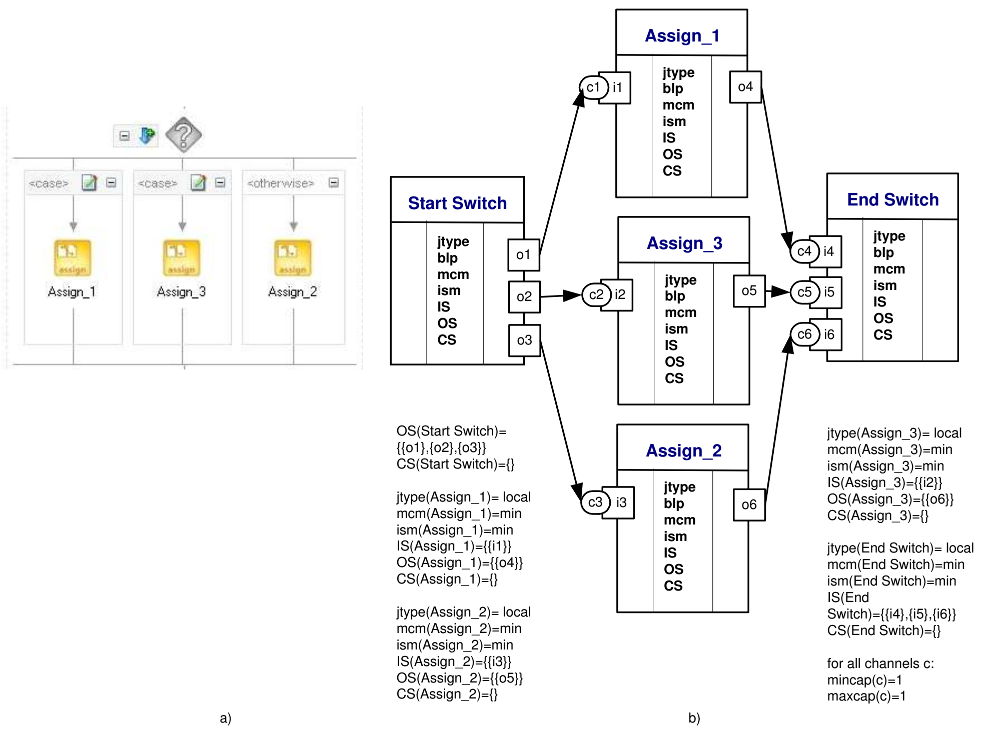


Fig. 22. Staffware implementation of WP4.

Oracle BPEL PM implementation Oracle BPEL PM supports this pattern by means of the $\langle switch \rangle$ construct, which selects the first case. If no case can be selected, the default branch is chosen. Note that due to the structured nature of the considered model, the Exclusive Choice pattern is combined with the Simple Merge pattern. Figure 23 (a) depicts the $\langle switch \rangle$ construct, and Figure 23 (b) shows the corresponding WPSL interpretation. WPSL tasks *Start Switch* and *End Switch* define the split and merge logic of the patterns. To denote that a single branch is to be chosen in the $\langle switch \rangle$ construct, the output sets of the WPSL task *Start Switch* are made exclusive. Similarly, the input sets of task $\langle End\,Switch \rangle$ are made disjoint. Note that a single message from a single branch needs to be received in order to complete the $\langle switch \rangle$ properly.

3.5 WP5 -Simple merge

Description A point in the workflow process where two or more alternative branches come together without synchronization. It is an assumption of this



OS(Start Switch)=
{{o1},{o2},{o3}}
CS(Start Switch)={}

jtype(Assign_1)= local
mcm(Assign_1)=min
ism(Assign_1)=min
IS(Assign_1)={{i1}}
OS(Assign_1)={{o4}}
CS(Assign_1)={}

jtype(Assign_2)= local
mcm(Assign_2)=min
ism(Assign_2)=min
IS(Assign_2)={{i3}}
OS(Assign_2)={{o5}}
CS(Assign_2)={}

jtype(Assign_3)= local
mcm(Assign_3)=min
ism(Assign_3)=min
IS(Assign_3)={{i2}}
OS(Assign_3)={{o6}}
CS(Assign_3)={}

jtype(End Switch)= local
mcm(End Switch)=min
ism(End Switch)=min
IS(End
Switch)={{i4},{i5},{i6}}
CS(End Switch)={}

for all channels c:
mincap(c)=1
maxcap(c)=1

Fig. 23. Oracle implementation of WP4.

pattern that none of the alternative branches is ever executed in parallel (if this is not the case, then see WP8 (Multi-merge) or WP9 (Discriminator)).

Selected WPSL specification Figure 24 demonstrates the Simple Merge pattern.

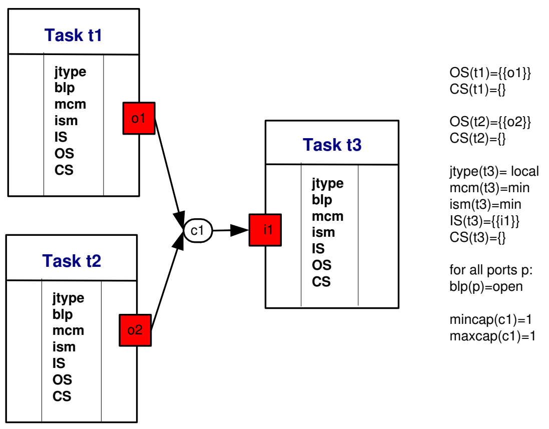


Fig. 24. WP5 - Simple merge. Selected WPSL specification.

Task $t3$ is enabled after either task $t2$ or task $t3$ is executed. Tasks $t1$ and $t2$ are connected to task $t3$ via an external safe channel $c1$. A message produced by an output port of either task $t1$ or $t2$ enables an input port $i1$ of task $t3$. This behavior is reflected by the minimal message consumption mode and the minimal channel capacity limited to 1. Enabling of the input port $i1$ of task $t3$ is based on the messages currently available in the channel.

Alternative WPSL specifications The points of variations based on which alternative configurations of the Simple Merge pattern can be based are listed below.

- **Channel characteristics**. *Channel capacity*: The capacity of channel $c1$ can be bounded or unbounded depending on the number of messages the channel is allowed to store at once. To specify that multiple messages are required to enable an input port of task $t3$ the minimal capacity of the corresponding non-safe channel must be set respectively.
- **Blocking of output ports**. Output ports can be set to the *blocked* or *open* mode to be used in combination with bounded or any type of channels respectively.
- **Message consumption mode**. In *minimal* message consumption mode the number of messages required to enable the channel (equivalent to the minimal channel capacity) is consumed, the rest of the non-consumed messages are stored in the channel for subsequent task execution. To consume all messages available in the channel, the message consumption mode should be set to *maximal*.

Staffware implementation Staffware supports this pattern directly via *Step* objects in the configuration shown in Figure 25(a). The corresponding WPSL interpretation is shown in Figure 25(b). Note that the enabling of task C is based on the messages locally available in the safe channel $c1$. *Step* C will be executed only once even if steps A and B would complete simultaneously, which is reflected by the open mode of the blocking mode of the outgoing channels belonging to the corresponding WPSL tasks A and B.

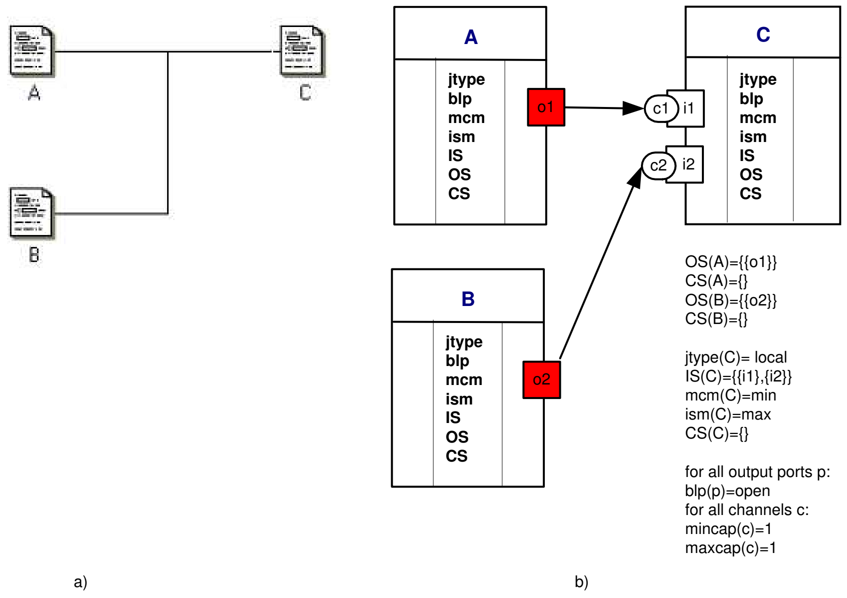


Fig. 25. Staffware implementation of WP5.

Oracle BPEL PM implementation Oracle BPEL PM supports this pattern directly using the ⟨*switch*⟩ construct depicted in Figure 23. Note that since this construct is of structured form, in Oracle BPEL PM the Simple Merge pattern is always used in combination with the Exclusive Choice pattern. This also guarantees that the pattern assumption that none of the alternative branches (in the ⟨*switch*⟩ construct) ever executes in parallel. The description of WPSL for the ⟨*switch*⟩ construct in the Exclusive Choice pattern is also valid in the context of the Simple Merge pattern.

3.6 WP6 -Multiple Choice

Description A point in the workflow process where, based on a decision or workflow control data, a number of branches are chosen.

Selected WPSL specification Figure 26 illustrates the Multiple Choice pattern.

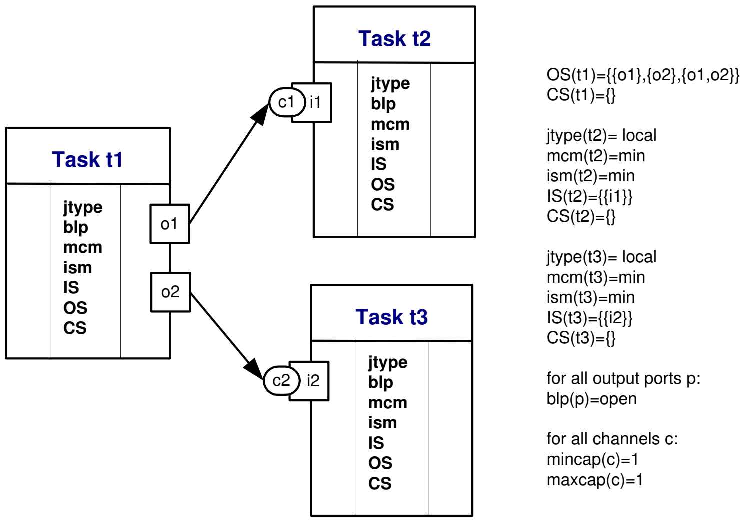


Fig. 26. WP6 - Multiple Choice. Selected WPSL specification.

After executing task $t1$, task $t2$ or task $t3$ is or both are executed. The actual logic associated with the multiple choice is encapsulated in the output sets of task $t1$, which specified that either one of two optional output ports or both are selected for the message production. To make the distinction between the execution pathes following task $t1$, the channels $c1$ and $c2$ are chosen to be dedicated (local). The channels are safe and messages stored in them are consumed in the minimal message consumption mode by task $t2$ and task $t3$. The setting of the input selection mode is omitted, since no option exists for selecting an input for tasks $t2$ and $t3$. The enabling of these tasks is based on the messages currently available in the channels. Note that the original pattern definition given in [5] does not consider the blocking mode of output ports. In the selected specification this parameter has been set to *open*.

Alternative WPSL specifications The points of variations describing alternative configurations of the Multiple Choice pattern are listed below.

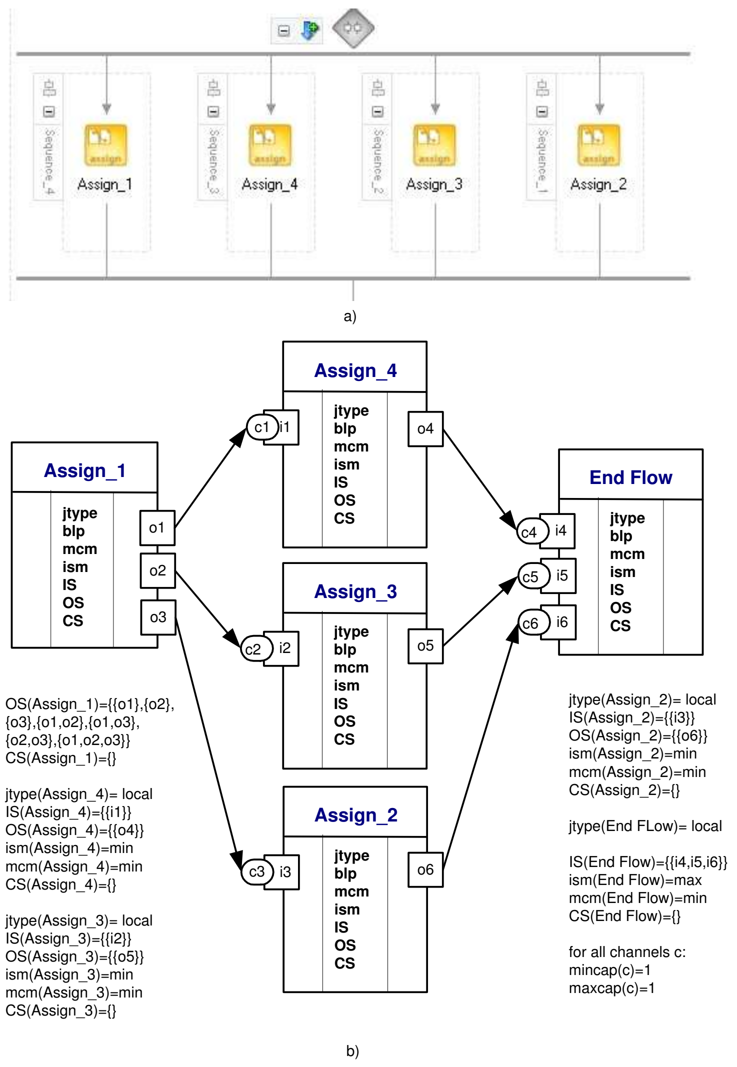
- **Channel characteristics**. *Channel capacity*: The capacity of channels $c1$ and $c2$ can be increased and set to bounded or unbounded depending on the number of messages the channel is allowed to store at once. To specify that multiple messages are required to enable an input port of tasks $t2$ and $t3$ the minimal capacity of the corresponding non-safe channels must be adjusted respectively.
- **Blocking of output ports**. Depending on the type of the channels used, output ports can be set to the *blocked* or *open* mode.
- **Message consumption mode**. For safe channels, the *minimal* and *maximal* message consumption modes are equivalent. If the channels in use are bounded or unbounded, the minimal or maximal message consumption modes must be

selected to specify the limited or full message consumption from the channel respectively.

Staffware implementation Staffware does not support this pattern directly. However, it can be implemented using a series of *Decision* steps which are binary and correspond to exclusive OR-splits (see [24]).

Oracle BPEL PM implementation Oracle BPEL supports this pattern by means of the $\langle flow \rangle$ construct and links. Links are associated with every branch. Branches become enabled when their transition conditions have been satisfied. Note that the graphical notation of Figure 27 (a) does not show links between the branches of the $\langle flow \rangle$, they are listed in the BPEL code. The ordering of activities by means of links is reflected in the logic defined over the input and output sets of the corresponding tasks belonging to the WPSL specification depicted in Figure 27 (b).

```
<flow name="Flow_1">
<links>
    <link name="Link14"/>
    <link name="Link13"/>
    <link name="Link12"/>
</links>
<sequence name="Sequence_4">
    <assign name="Assign_1">
    <source linkName="Link12" transitionCondition=""/>
    <source linkName="Link13" transitionCondition=" "/>
    <source linkName="Link14" transitionCondition=" "/>
    </assign>
</sequence>
<sequence name="Sequence_3">
    <target linkName="Link13"/>
    <assign name="Assign_4">
    ...
    </assign>
</sequence>
<sequence name="Sequence_2">
    <target linkName="Link12"/>
    <assign name="Assign_3">
    ...
    </assign>
</sequence>
<sequence name="Sequence_1">
    <target linkName="Link14"/>
    <assign name="Assign_2">
    ...
    </assign>
</sequence>
</flow>
```



OS(Assign_1)={{o1},{o2},{o3},{o1,o2},{o1,o3},{o2,o3},{o1,o2,o3}}
CS(Assign_1)={}

jtype(Assign_4)= local
IS(Assign_4)={{i1}}
OS(Assign_4)={{o4}}
ism(Assign_4)=min
mcm(Assign_4)=min
CS(Assign_4)={}

jtype(Assign_3)= local
IS(Assign_3)={{i2}}
OS(Assign_3)={{o5}}
ism(Assign_3)=min
mcm(Assign_3)=min
CS(Assign_3)={}

jtype(Assign_2)= local
IS(Assign_2)={{i3}}
OS(Assign_2)={{o6}}
ism(Assign_2)=min
mcm(Assign_2)=min
CS(Assign_2)={}

jtype(End FLow)= local

IS(End Flow)={{i4,i5,i6}}
ism(End Flow)=max
mcm(End Flow)=min
CS(End Flow)={}

for all channels c:
mincap(c)=1
maxcap(c)=1

Fig. 27. Oracle implementation of WP6.

3.7 WP7 -Synchronizing Merge

Description A point in the workflow process where multiple paths converge into one single thread. If more than one path is taken, synchronization of the active threads needs to take place. If only one path is taken, the alternative branches should reconverge without synchronization. It is an assumption of this pattern that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete.

Selected WPSL specification Figure 28 demonstrates the Synchronizing Merge pattern.

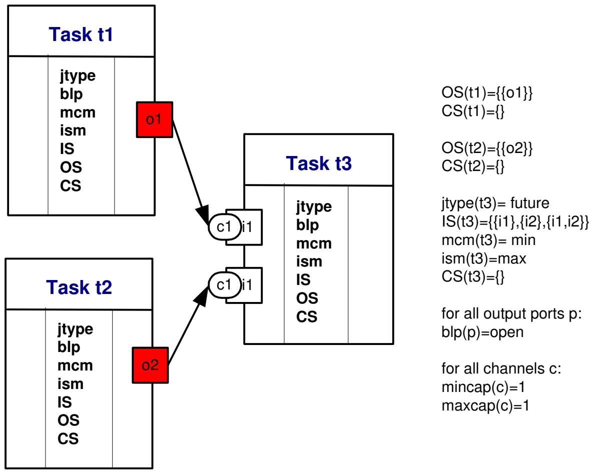


Fig. 28. WP7 - Synchronizing Merge. Selected WPSL specification.

Task $t3$ is executed after executing task $t1$ or task $t2$ or both. In order to specify that task $t3$ should wait until no more messages may arrive to the task inputs, *jtype* parameter of this task is set to *future*. Since several sets from the input set of task $t3$ can be enabled simultaneously, the largest set must be selected, therefore the input selection mode is set to *maximal*. In combination with safe channels, the message consumption mode set to *minimal*, ensures that only one message must be consumed by each enabled port.

Alternative WPSL specifications The points of variation describing alternative configurations of the Synchronizing Merge pattern are listed below.

- **Channel characteristics**. *Channel capacity*: The capacity of channels $c1$ and $c2$ can be increased and set to bounded or unbounded depending on the number of messages the channel is allowed to store at once. To specify that multiple messages are required to enable an input port(s) of task $t3$ the minimal capacity of the corresponding non-safe channel must be set respectively.
  Channel positioning: local channels can be replaced by external ones with an additional requirement that access to the messages stored in the external channel is limited to a single (dedicated)task.

- **Blocking of output ports**. Depending on the type of the channels used, output ports can be set to the *blocked* or *open* mode.
- **Message consumption mode**. In the context of safe channels, the *minimal* and *maximal* message consumption modes are equivalent. If the channels in use are bounded or unbounded, the minimal or maximal message consumption modes must be selected to specify the limited or full message consumption from a channel respectively.

Staffware implementation Staffware does not support this pattern directly. However, similar behavior can be obtained by combining *Decision* constructs and complex routers on the inputs of the *Wait* object as it is shown in [24].

Oracle BPEL PM implementation Oracle BPEL PM supports this pattern via $\langle flow \rangle$ and links as described for the Multiple Choice pattern (see Figure 27). Links with transition conditions are used to define which branches within the $\langle flow \rangle$ construct are to be selected. The synchronization of branches is done by the $\langle flow \rangle$ activity. The $\langle flow \rangle$ activity will only complete when each of its sub-activities has either completed or has been skipped. The continuation of the process after the synchronizing merge can be placed after the $\langle flow \rangle$ activity.

3.8 WP8 -Multiple Merge

Description A point in a workflow process where two or more branches reconverge without synchronization. If more than one branch is activated, the activity following the merge is started for every activation of every incoming branch (including concurrent instances).

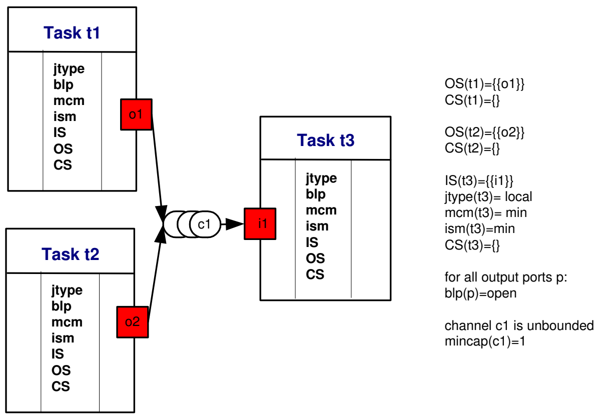


Fig. 29. WP8 - Multiple Merge. Selected WPSL specification.

Selected WPSL specification Figure 29 illustrates the Multiple Merge pattern. Task $t3$ is executed each time the execution of task $t1$ or task $t2$ completes. Messages arriving to the inputs of task $t3$ are aggregated in the unbounded channel $c1$, while task $t3$ is in the active state, thus allowing the task to be enabled and executed again. Since multiple messages can be stored in the channel $c1$ and each of the messages must enable task $t3$, the message consumption mode of this task is set to minimal. The minimal capacity of the channel, which is limited to 1, is consumed by the input port $i1$ of task $t3$ each time any of tasks $t1$ or $t2$ completes. Note that output ports of task $t1$ and task $t2$ are open and never become blocked, since the maximal capacity of the unbounded channel $c1$ can never be reached.

Alternative WPSL specifications The points of variations describing alternative configurations of the Multiple Merge pattern are listed below.

- **Channel characteristics**. *Channel capacity*: The capacity of the channel $c1$ can be bounded to the maximal number of messages the channel is able to store at once. It may be necessary to amend the blocking type of the output ports of task $t1$ and task $t2$ to specify whether the tasks are allowed to complete if the maximal capacity of the outgoing channels has been reached. The minimal channel capacity can be also modified, which will have an influence on the status of the enabling of task $t3$.
- **Input Sets**. If the external channel $c1$ is replaced by two local channels for a dedicated transfer of messages from tasks $t1$ and $t2$, the input sets of task $t3$ should be made disjoint. To guarantee that task $t3$ will be executed upon enabling of each port, but not once if both input ports are enabled, the input consumption mode should be set to *minimal* and messages should be consumed from the channels in the *minimal* message consumption mode.

Staffware implementation This pattern is not supported by Staffware directly. However, similar behavior can be obtained by duplicating task C and placing it on two branches with tasks A and B respectively as it is shown in [24].

Oracle BPEL PM implementation Oracle BPEL PM offers the facilities to implement this pattern by means of an event handler attached to the scope in which multiple branches reside. In every branch enclosed in the $\langle flow \rangle$ construct, an $\langle invoke \rangle$ activity should be placed to invoke a synchronous dummy service. The response message produced by this dummy service needs to be processed by an event handler attached to the scope outer of the $\langle flow \rangle$ construct. This results in an event handler being executed each time a branch completes. For some reason, the event handlers attached to the process scope seem to be unable to respond to messages. Since the desired behavior hasn't been achieved by means of event handlers, this pattern is not considered to be supported. Further details on this issue are available in [18].

3.9 WP9a -Discriminator

Description The discriminator is a point in a workflow process that waits for one of the incoming branches to complete before activating the subsequent activity. From that moment on it waits for all remaining branches to complete and “ignores” them. Once all incoming branches have been triggered, it resets itself so that it can be triggered again (which is important otherwise it could not really be used in the context of a loop).

Selected WPSL specification Figure 30 illustrates the default WPSL configiration of the Discriminator pattern.

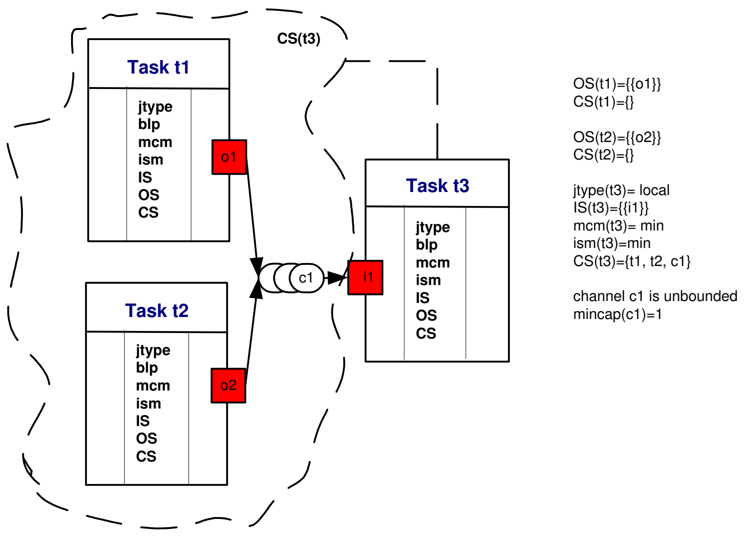


Fig. 30. WP9a - Discriminator. Selected WPSL specification.

Task $t3$ is executed after task $t1$ or task $t2$ completed. When task $t3$ terminates it removes messages inside of the tasks and in channels, thus disabling any remaining triggers. After the messages from the cancellation set are removed, task $t3$ can be triggered again. Although multiple messages can be placed concurrently on the input channel of task $t3$, only one message is consumed from it. This is reflected by setting the minimal channel capacity to 1 and the message consumption mode to *minimal*.

Alternative WPSL specifications The points of variation describing alternative configurations of the Discriminator pattern are listed below.

- **Channel characteristics**. *Channel capacity*: The capacity of the channel $c1$ can be bounded to the maximal number of messages the channel is able to store at once. If the capacity of the channel is bounded, the minimal message capacity of the channel should be set to the number of messages required to enable the input port connected to this channel.

  In this case, the blocking type of the output ports of task $t1$ and task $t2$ should be amended to specify whether the tasks are allowed to complete when the maximal capacity of the outgoing channels has been reached. The minimal channel capacity can be also modified, which will affect the enabling status of the channel $c1$ and task $t3$.

- **Blocking of output ports**. Depending on the type of channels used, output ports can be set to the *blocked* or *open* mode. Safe and bounded channels can be connected to the open and blocked output ports, while unbounded channels can only be connected to open output ports respectively.
- **Message consumption mode**. If the channels in use are bounded or unbounded, then the minimal or maximal message consumption modes must be selected to specify the limited or full message consumption from the channels respectively.
- **Input Sets**. If the external channel $c1$ is replaced by two local channels for a dedicated transfer of messages from tasks task $t1$ and task $t2$, the input sets of task $t3$ should be made disjoint. To guarantee that task $t3$ will be executed upon enabling of each port, and not once (if both input ports are enabled), the input consumption mode and the message consumption mode should be set to *minimal*.
- **Cancellation Set**. The cancellation set defined for task $t3$ can be extended depending on the context in which the discriminator is used.

Staffware implementation Staffware does not support this pattern directly. However, a work around solution can be constructed using a script and *Decision* construct as it is shown in [24].

Oracle BPEL PM implementation Oracle BPEL PM does not support this pattern.

3.10 WP9b -N-out-of-M join

Description N-out-of-M Join is a point in a workflow process where M parallel paths converge into one. The subsequent activity should be activated once N paths have completed. Completion of all remaining paths should be ignored. Similarly to the discriminator, once all incoming branches have "fired", the join resets itself so that it can fire again.

Selected WPSL specification Figure 31 demonstrates the N-out-of-M join pattern.

Configuration shown is for a 2-out-of-3 join, however it can be easily extended to any required number of inputs. Task $t4$ is executed after any two tasks from $t1$, $t2$ and $t3$ have completed. Task $t4$ consumes only 2 messages, which is ensured by setting the minimal message capacity of the channels $c1$, $c2$ and $c3$ to 1 and the message consumption mode to *minimal*. The input set specifies that task $t4$ can be enabled if any two input ports have been enabled. In the case where more than two ports are enabled, two ports are arbitrarily selected. This is reflected by the input selection mode which is set to *random*. After task $t4$ completes, the discriminator is reset, by removing the messages remaining in locations specified in the cancellation set of task $t4$.

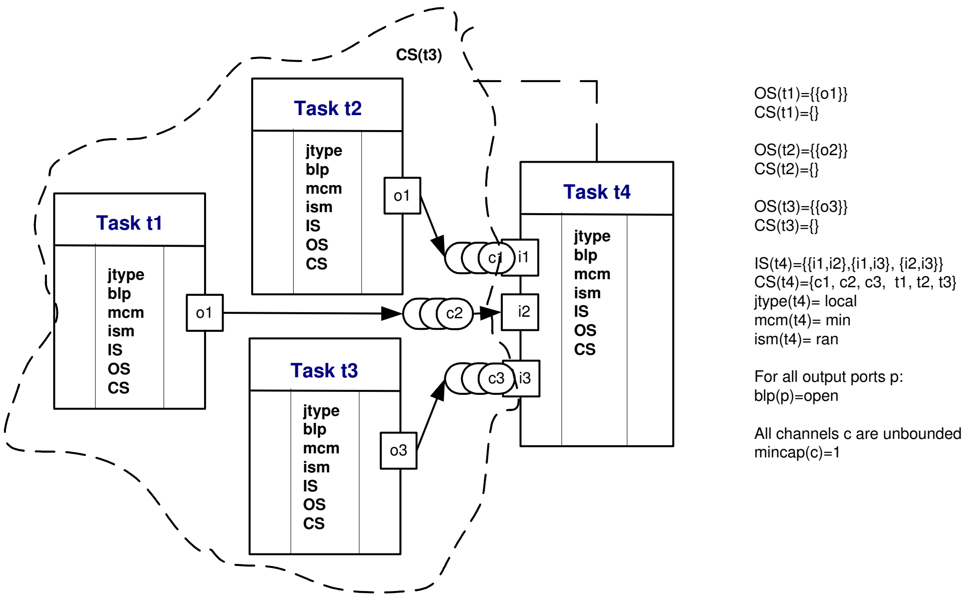


Fig. 31. WP9b - 2-out-of-3 join. Selected WPSL specification.

Alternative WPSL specifications The points of variation describing alternative configurations of the N-out-of-M join pattern are the same as for the Discriminator pattern. In addition, the following variation points apply:

- **Input Sets**. The current definition of the N-out-of-M join pattern captures only an arbitrary combination of branches. The composition of the input sets of task *t*4 for enabling of the N-out-of-M join can be can be tuned to any required combination of inputs through the set restriction or extension.

- **Cancellation Set**. The N-out-of-M join can be used with or without a cancellation set. If at the time of task completion non-consumed messages must be removed from the input locations, the cancellation set should explicitly list these locations. Conversely, if non-consumed messages must be preserved for the subsequent task enabling, the cancellation set should remain empty.

Staffware implementation Staffware does not support this pattern directly. However, the work around solution can be found using a script, a counter for inputs and *Decision* construct as it is described in [24].

Oracle BPEL PM implementation Oracle BPEL PM does not support this pattern.

3.11 WP10 -Arbitrary cycles

Description A point in a workflow process where one or more activities can be done repeatedly.

Selected WPSL specification Figure 32 illustrates a possible configuration of the Arbitrary Cycles pattern.

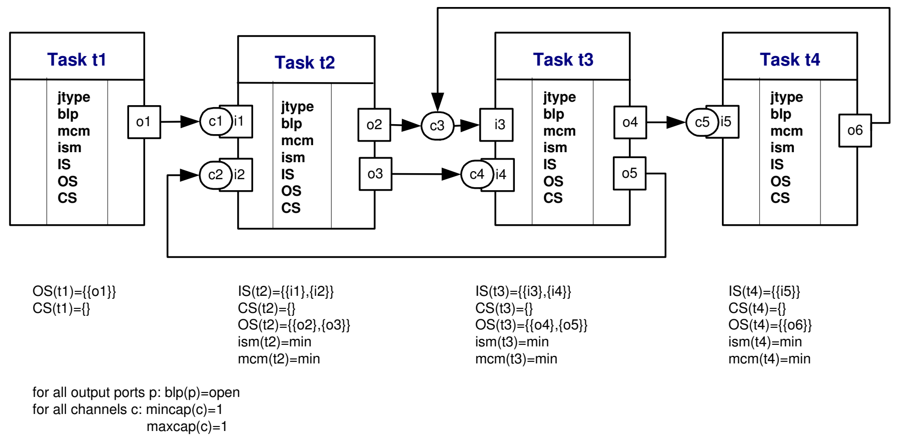


Fig. 32. WP10 - Arbitrary cycles. Selected WPSL specification.

After task $t1$ completes, tasks $t2$, $t3$ and $t4$ can be executed multiple times depending on the choice made after tasks $t3$ and $t4$ completed respectively. The model presented allows for multiple entry and output points from the loop. The choice of branches for the subsequent enabling on completion of tasks $t2$, $t3$, and $t4$ is specified in the output sets of the corresponding tasks.

Alternative WPSL specifications Alternative representations of the Arbitrary Cycles pattern depend on the number of tasks in the loop, the number of entry and output points, additional requirements and constraints set on the tasks that can be expressed by means of the task attributes.

The Arbitrary Cycles pattern is a generic representation of the looping structures, which in special cases can be implemented as "while-do" and "repeat-until" strategies.

Staffware implementation Staffware supports this pattern allowing multiple intertwined cycles as it shown in Figure 33(a). The corresponding WPSL notation is shown in Figure 33(b). After completing step B the choice is made whether to execute this step again or to proceed further. This choice is visualized by means of optional output ports and exclusive output sets of task B. The input ports of task B are made optional respectively and the input sets can be enabled by messages received on any of the input ports.

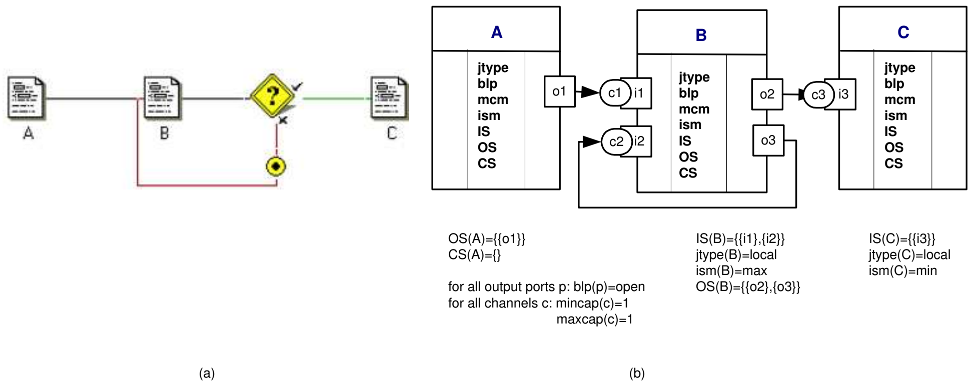


Fig. 33. Staffware implementation of WP10.

Oracle BPEL PM implementation Oracle BPEL PM supports this pattern partially via the $\langle while \rangle$ construct depicted in Figure 34(a). The behavior of this construct is represented in WPSL in Figure 34(b). After task $Assign_2$ completes, the decision is made whether to execute this task again (by producing a message on the output port $o1$) or to exit the loop (by producing a message on the output port $o2$).

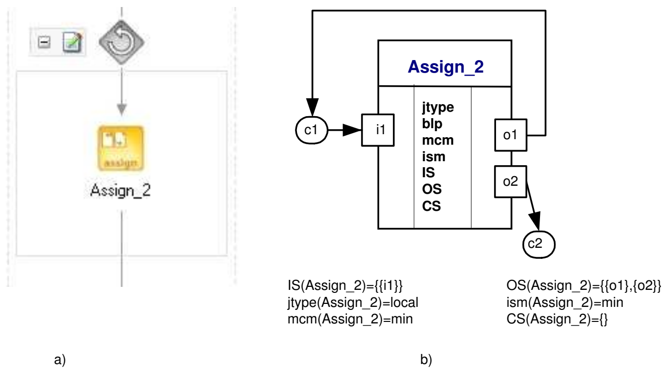


Fig. 34. Oracle implementation of WP10.

3.12 WP11 -Implicit termination

Description A given subprocess should be terminated when there is nothing else to be done. In other words, there are no active activities in the workflow and no other activity can be made active (and at the same time the workflow is not in deadlock).

Selected WPSL specification In order to represent this pattern by means of WPSL, we relax the notion of soundness of the GWF-net, allowing multiple end-point channels, the enabling of which does not lead to immediate case termination but is postponed until no active tasks are left which still may execute or need to complete.

Task *ET* depicted in Figure 35 can be used to denote this pattern. *ET* makes implicit termination explicit and takes as inputs the end-point channels of a GWF-net, synchronizing them all using the information about future possible states. To complete the case, all messages from all end-point channels are consumed and one message is produced to the *et* end-channel.

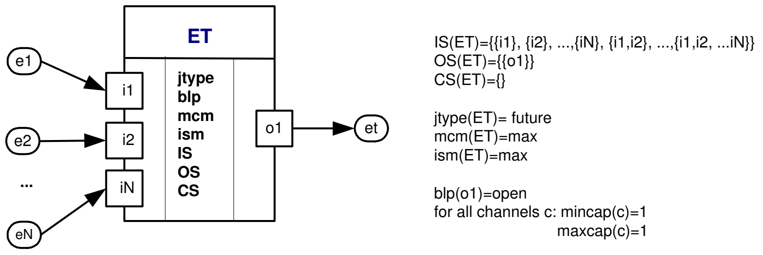


Fig. 35. WP11 - Implicit termination. The WPSL notation.

Staffware implementation Staffware suports this pattern directly. The workflow instance terminates if all of the corresponding branches have terminated.

Oracle BPEL PM implementation Oracle BPEL PM supports this pattern directly by the $\langle flow \rangle$ construct, which terminates when no activities within its body can be triggered and executed any more.

3.13 WP12 -Multiple Instances Without Synchronization

Description Within the context of a single case (i.e., workflow instance) multiple instances of an activity can be created, i.e., there is a facility to spawn off new threads of control. Each of these threads of control is independent of other threads. Moreover, there is no need to synchronize these threads.

Selected WPSL specification Figure 36 demonstrates the Multiple Instances Without Synchronization pattern.

After task $t1$ completes task $t3$ is initiated one or more times. The instances of task $t3$ are not synchronized, which is ensured by means of task $t2$ that is implemented in the body of a loop. At the completion of task $t2$, either task $t3$ is triggered and task $t2$ will be initiated again, or task $t3$ is initiated and the flow of control is passed further via an external channel $c3$. Note, that to realize this behavior, the output ports of task $t2$ are chosen to be optional, and the output logic of this task is captured by the output sets attribute. In the given confuguration unbounded channels are attached to the open output ports.

Alternative WPSL specifications The variation points of the MI without Synchronization pattern are listed below:

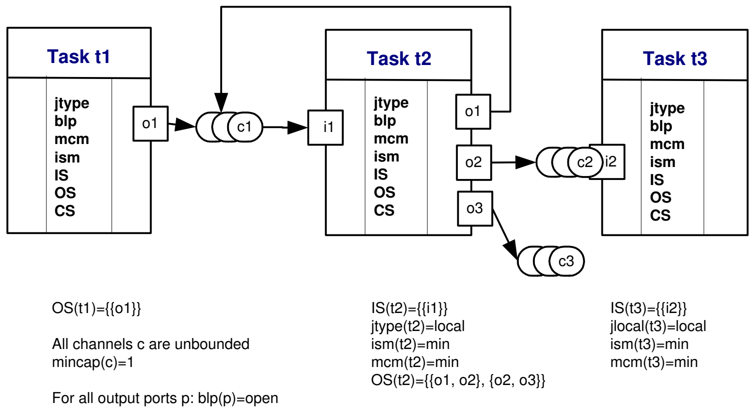


Fig. 36. WP12 - Multiple Instances Without Synchronization

- **Channel characteristics**. *Channel capacity*: The capacity of the channels can be set to bounded or unbounded depending on the number of messages the channel is able to store at once. The minimal channel capacity of the channel should be set conjunction with the maximal channel capacity.
- **Blocking of output ports**. Depending on the type of the channels used, output ports can be set to the *blocked* or *open* mode. Safe and bounded channels can be connected to the open and blocked output ports, while unbounded channels only to the open output ports.
- **Message consumption mode**. If the channels in use are bounded or unbounded, the minimal or maximal message consumption modes must be selected to specify the limited or full message consumption from the channels respectively.

Staffware implementation Staffware does not support this pattern directly. However, a sub-case can be used as a wrapper for an activity multiple instances of which need to be instantiated as described in [24].

Oracle BPEL PM implementation Oracle BPEL PM supports this pattern using the $\langle invoke \rangle$ construct within the body of a $\langle while \rangle$ loop as shown in Figure 37(a). The corresponding WPSL interpretation is given in Figure 37(b).

3.14 WP13 -Multiple Instances with a Priori Design time Knowledge

Description For one process instance an activity is enabled multiple times. The number of instances of a given activity for a given process instance is known at design time. Once all instances are completed some other activity needs to be started.

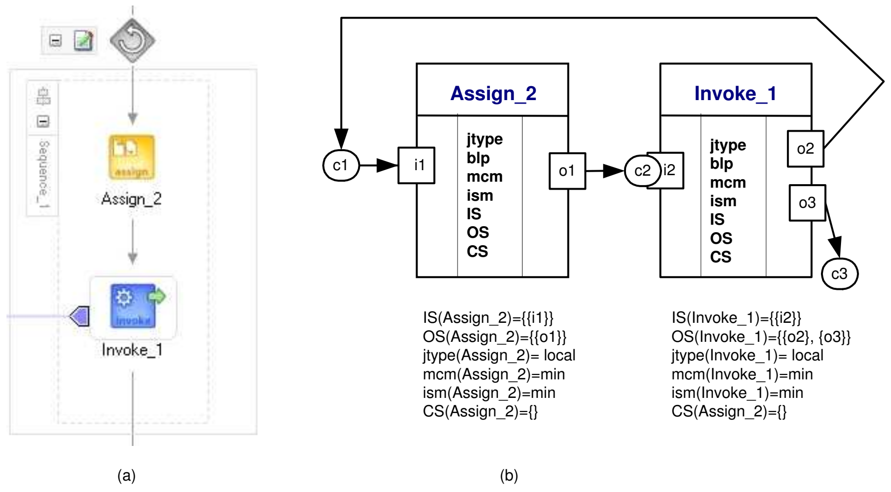


Fig. 37. Oracle implementation of WP12.

Selected WPSL specification Figure 38 demonstrates the default WPSL configuration of the Multiple Instances with a priori design time knowledge pattern. To distinguish between different types of multiple instance tasks we introduce some additional graphical notations. The number of instances that are to be created is denoted as N and listed in the list of the task attributes. The label DTK reflects that the number of instances of the $MItask$ to be created is known at design-time. The execution of this task corresponds to the enabling of N instances of a given task in parallel.

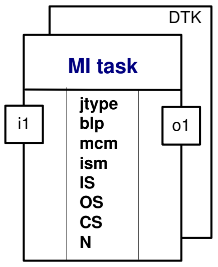
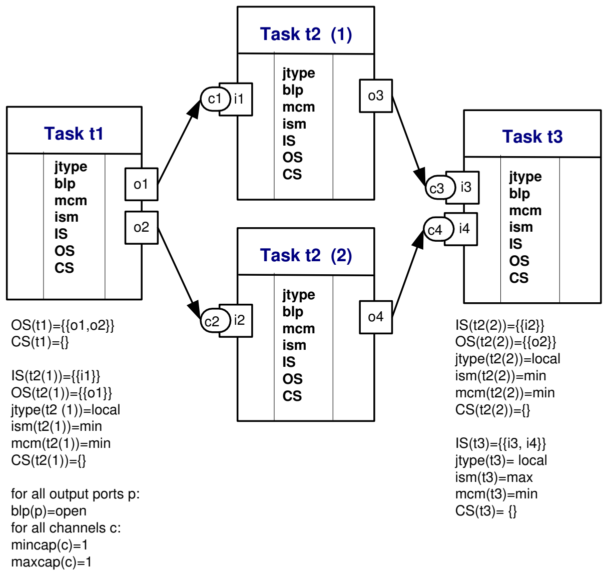


Fig. 38. WP13 - Multiple Instances with a Priori Design time Knowledge. The WPSL specification.

Alternative WPSL specifications An alternative means of specification is to duplicate a task as many times as many task instances are required and to use them together with the Parallel Split and Synchronization patterns as illustrated in Figure 39. After task $t1$ completes and before task $t3$ may commence, two instances of task $t2$ need to be executed. Since all instances need to complete before task $t3$ commences, task $t3$ input sets specifies that both



OS(t1)={{o1,o2}}
CS(t1)={}

IS(t2(1))={{i1}}
OS(t2(1))={{o1}}
jtype(t2 (1))=local
ism(t2(1))=min
mcm(t2(1))=min
CS(t2(1))={}

for all output ports p:
blp(p)=open
for all channels c:
mincap(c)=1
maxcap(c)=1

IS(t2(2))={{i2}}
OS(t2(2))={{o2}}
jtype(t2(2))=local
ism(t2(2))=min
mcm(t2(2))=min
CS(t2(2))={}

IS(t3)={{i3, i4}}
jtype(t3)= local
ism(t3)=max
mcm(t3)=min
CS(t3)= {}

Fig. 39. WP13 - Multiple Instances with a Priori Design time Knowledge. An alternative WPSL specification.

input ports must be enabled. Thus, task $t3$ plays a role of a synchronizer as described in the Synchronization pattern, and task $t1$ represents the Parallel Split.

The task attributes set for the MI task are identical for all instances of this task. The following attributes may be varied from the settings described above: type of channels, message consumption mode, input sets, and blocking mode. The values they can take are the same as those described for the Parallel Split and Synchronization patterns.

Staffware implementation Staffware supports this pattern through combination of splits and joins as illustrated in Figure 40. Multiple instances of a task B are achieved by replicating this task and enabling all instances simultaneously. The synchronization of these tasks is done as the Synchronization pattern describes.

Oracle BPEL PM implementation Oracle BPEL PM supports this pattern by replicating an activity on each of the branches of the $\langle flow \rangle$ activity. There should be one branch for each activity instance. See the Parallel Split pattern for the details of the configuration of the $\langle flow \rangle$ activity.

3.15 WP14 -Multiple Instances with a Priori Runtime Knowledge

Description For one case an activity is enabled multiple times. The number of instances of a given activity for a given case varies and may depend on characteristics of the case or availability of resources, but is known at some

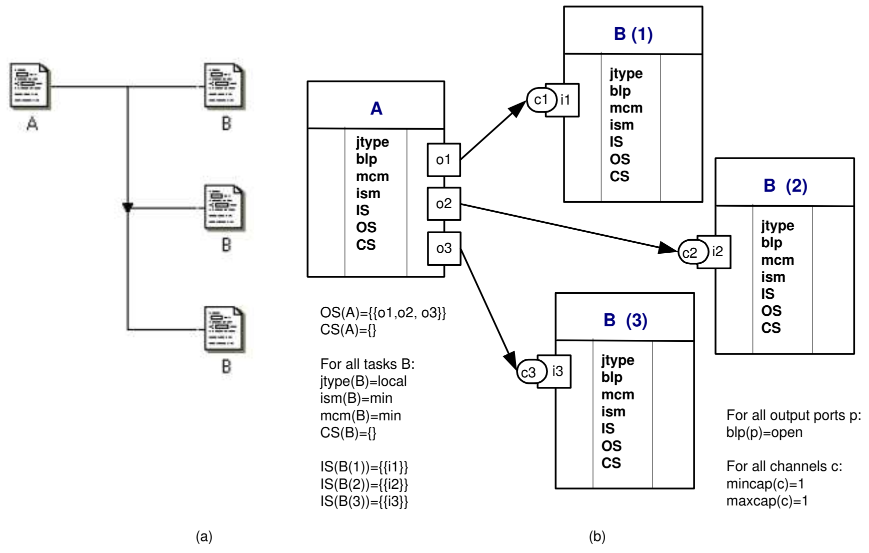


Fig. 40. Staffware implementation of WP13.

stage during runtime, before the instances of that activity have to be created. Once all instances are completed some other activity needs to be started.

Selected WPSL specification Figure 41 illustrates the default WPSL configuration of the Multiple Instances with a priori runtime knowledge pattern. The number of instances that are to be created is denoted as N and listed in the list of the task attributes. The label RTK means that the number of instances of the $MItask$ to be created is known at run-time. The execution of this task corresponds to the enabling of N instances of a given task in parallel.

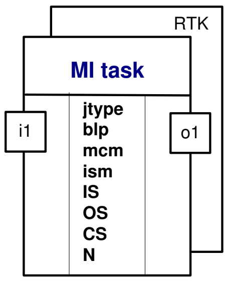


Fig. 41. WP14 -Multiple Instances with a Priori Runtime Knowledge The WPSL specification.

Alternative WPSL specifications Figure 42 shows how to depict the sequential initiation of the task instances.

After completion of task $t1$, task $t2$ needs to be executed several times. Assuming that the information regarding the number of instances of task $t2$

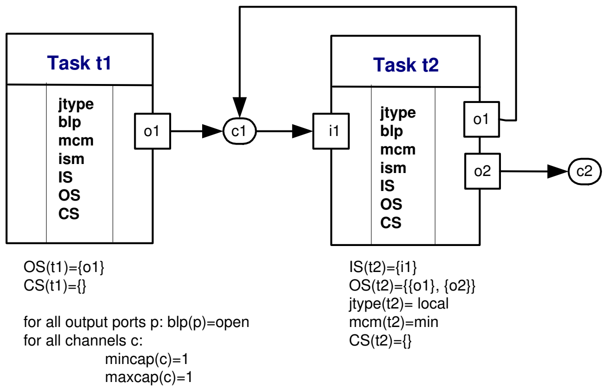


Fig. 42. WP14 -Multiple Instances with a Priori Runtime Knowledge An alternative WPSL notation.

becomes available at some stage during run-time, it can be incorporated into the guard of task $t2$ and/or data conditions associated with enabling of the output ports. In this configuration, task $t2$ is encapsulated within a loop, which imposes the sequential execution of the instances of task $t2$.

Staffware implementation Staffware does not support this pattern directly. However, it allows an activity to be mapped to a sub-case and use a "multiple sub-procedure call-step" in combination with dynamic array assignments as it is described in [24].

Oracle BPEL PM implementation Standard BPEL does not support this pattern, however Oracle BPEL PM has implemented the $\langle flowN \rangle$ construct which supports it (see Figure 43). Note that although for creation of instances, $\langle flowN \rangle$ uses run-time information, the actual execution of instances is done sequentially (thus corresponding to the behavior demonstrated in Figure 42).

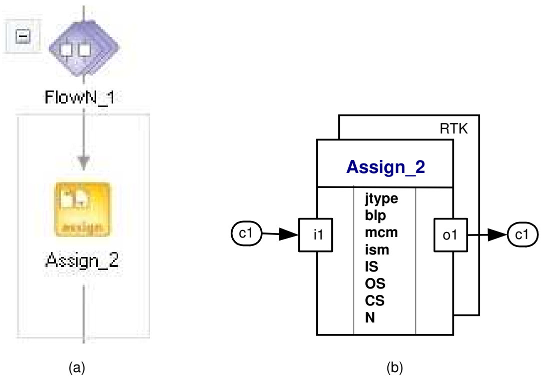


Fig. 43. Oracle implementation of WP14.

3.16 WP15 -Multiple Instances without a Priori Runtime Knowledge

Description For one case an activity is enabled multiple times. The number of instances of a given activity for a given case is not known during design time, nor is it known at any stage during runtime, before the instances of that activity have to be created. Once all instances are completed some other activity needs to be started. The difference with Pattern 14 is that even while some of the instances are being executed or already completed, new ones can be created.

Selected WPSL specification To express this pattern in WPSL, we introduce two types of blocks as shown in Figure 44. Blocks (a) and (b) contain labels D and S denoting dynamic and static modes for creating task instances. In dynamic mode, task instances can be created while other instances are active. In static mode, all task instances are created at once. We extend the list of task attributes for these blocks with the following three parameters: min, max, threshold. Parameters min, max, $threshold$ correspond to the minimum number of instances initiated, the maximum number of instances initiated and the threshold value which indicates how many instances must complete before the process can continue.

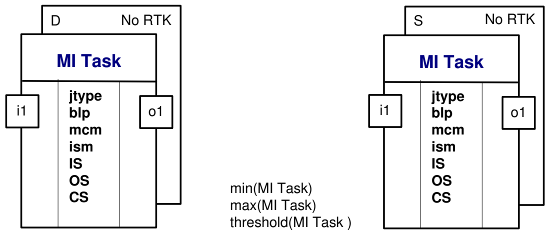


Fig. 44. WP15 -Multiple Instances without a Priori Runtime Knowledge. The WPSL notation.

Staffware implementation Staffware does not support this pattern directly.

Oracle BPEL PM implementation Oracle BPEL PM offers no direct support for this pattern.

3.17 WP16 -Deferred choice

Description A point in the workflow process where one of several branches is chosen. In contrast to the XOR-split, the choice is not made explicitly (e.g. based on data or a decision) but several alternatives are offered to the environment. However, in contrast to the AND-split, only one of the alternatives is executed. This means that once the environment activates one of the branches the other alternative branches are withdrawn. It is important to note that 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.

Selected WPSL specification Figure 45 illustrates the Deferred Choice pattern.

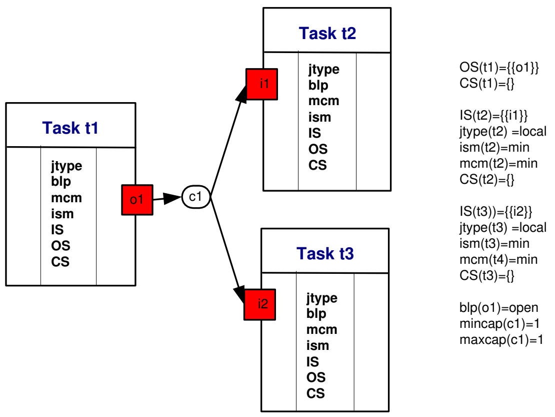


Fig. 45. WP16 - Deferred choice. Selected WPSL specification.

Task $t1$ is followed either by task $t2$ or by task $t3$. To visualize the non-deterministic choice that is to be made, task $t2$ and task $t3$ are connected to a single external channel, thus sharing the messages stored in it. When the minimal capacity of the channel is reached, both task $t2$ and task $t3$ become enabled. However, a message can be consumed only by one task at a time. Therefore, after the message from the channel $c1$ has been consumed by any of task $t2$ or task $t3$, the other task becomes disabled.

Alternative WPSL specifications Alternative configurations of the Deferred Choice pattern may use all variants of the channel capacity, channel positioning, and blocking mode of the output ports as indicated for the Exclusive Choice pattern.

Staffware implementation Staffware does not support this pattern directly. However, it can be implemented by means of a parallel split and a withdraw as it shown in [24]).

Oracle BPEL PM implementation Oracle BPEL PM supports this pattern directly via the ⟨*pick*⟩ construct (see Figure 46)(a), which allows only one of several possible activities or (a set of activities) to be executed based on the type of message received. Alternatively, the ⟨*pick*⟩ construct allows a time trigger to be specified for the timeouts which cancel all other alternative tasks. The WPSL specification corresponding to the ⟨*pick*⟩ construct is shown in Figure 46(b). For the sake of convenience we abstracted from the modelling of the time trigger.

3.18 WP17 -Interleaved Parallel Routing

Description A set of activities is executed in an arbitrary order: each activity in the set is executed, the order is decided at run-time, and no two activities are

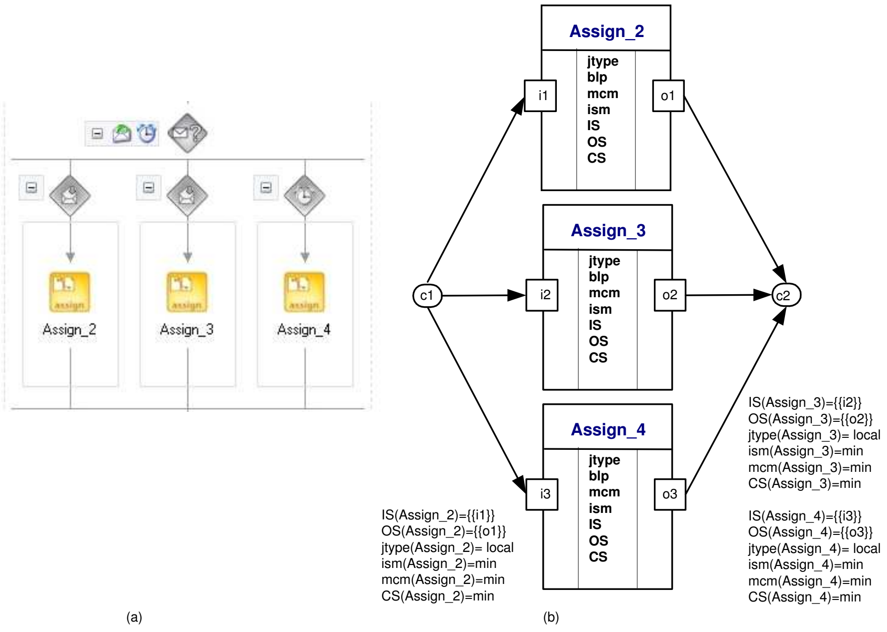


Fig. 46. Oracle implementation of WP16.

executed at the same moment (i.e. no two activities are active for the same workflow instance at the same time).

Selected WPSL specification Figure 47 illustrates the Interleaved Parallel Routing pattern.

Task $t2$ and task $t3$ can be executed in any order after task $t1$ has completed but before task $t4$ has commenced. To realize such a behavior, an external channel $c3$ is used to ensure that the execution of task $t2$ and task $t3$ is mutually exclusive. To enable both task $t2$ and task $t3$ concurrently, task $t1$ is realized according to the Parallel Split pattern, and task $t4$ is realized according to the Synchronization pattern.

Alternative WPSL specifications This pattern allows a partial order to be specified over the sequence in which tasks are executed. This can be achieved by modifying the input sets, the input selection mode and the message consumption mode of the respective tasks (in this case task $t2$ and task $t3$). In addition, the number of tasks involved in the interleaved parallel routing, can be increased.

Staffware implementation Staffware does not support this pattern directly (see [24]).

Oracle BPEL PM implementation Oracle BPEL PM does not support this pattern (see [18]).

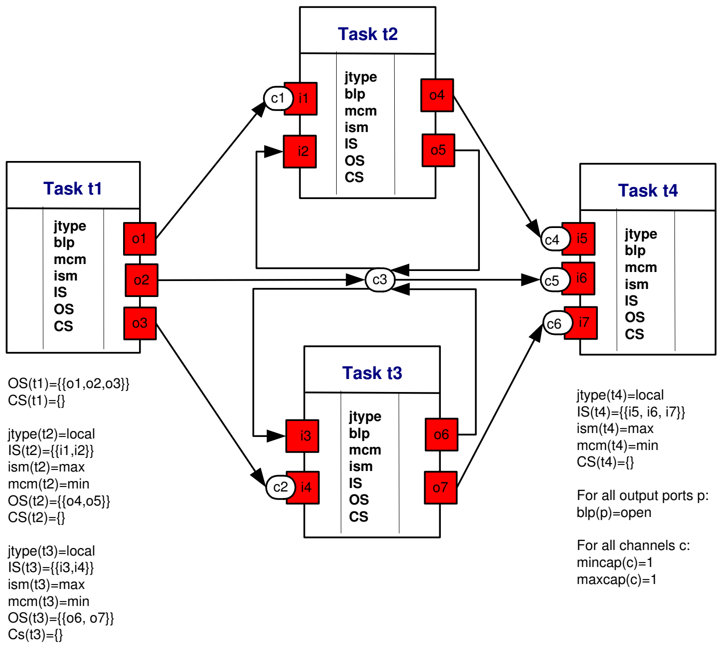


Fig. 47. WP17 - Interleaved Parallel Routing. Selected WPSL specification

3.19 WP18 -Milestone

Description The enabling of an activity depends on the case being in a specified state, i.e. the activity is only enabled if a certain milestone has been reached which did not expire yet. Consider three activities named A, B and C. Activity A is only enabled if activity B has been executed and C has not been executed yet, i.e. A is not enabled before the execution of B and A is not enabled after the execution of C.

Selected WPSL specification Figure 48 demonstrates the Milestone pattern. In this notation, the Milestone is modelled by means of an external channel $c2$. Task $t5$ performs the role of testing the milestone if task $t7$ has not been executed yet. If the milestone (in form of a message) in the channel $c2$ is not available, and task $t4$ has completed, task $t5$ cannot be executed yet. As a result a by-pass is taken, executing task $t6$. After the execution of task $t6$, when a message is placed in channel $c2$, task $t5$ cannot be executed anymore. The described logic is incorporated into the input and output sets of the corresponding tasks.

Alternative WPSL specifications The WPSL specification presented in Figure 48 allows the variations of channel capacity, although the positioning of the channel corresponding to the milestone must be preserved.

Staffware implementation Staffware does not support this pattern.

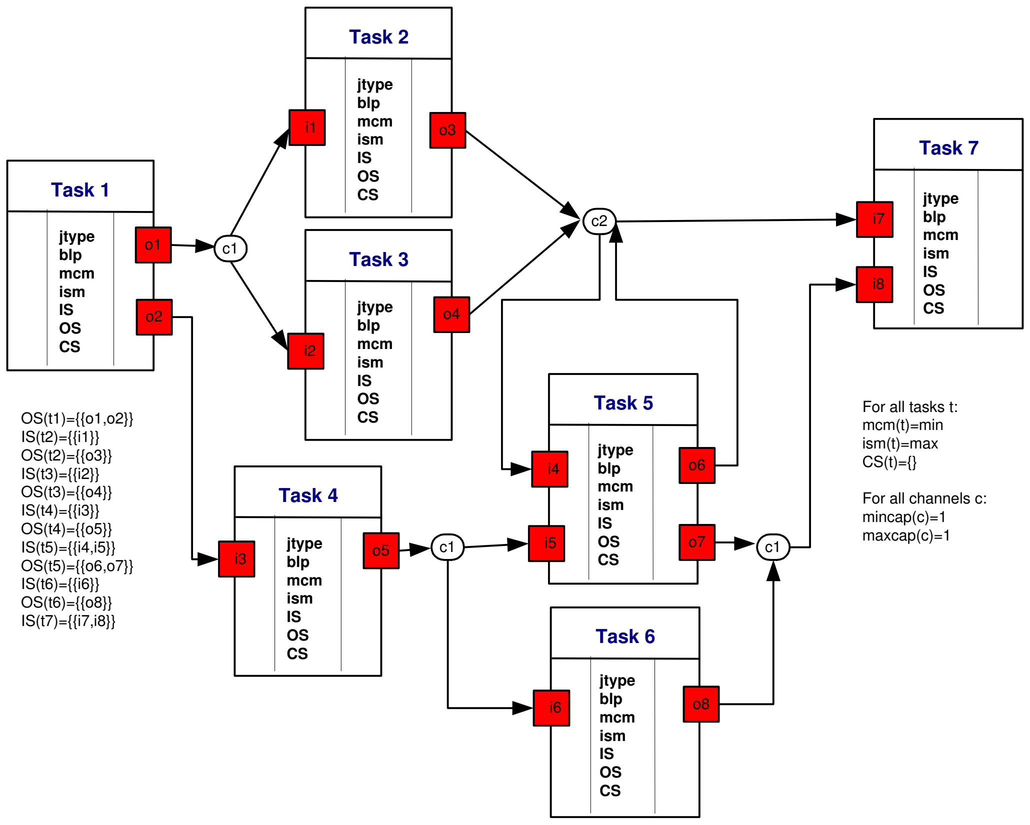


Fig. 48. WP18 -Milestone. Selected WPSL specification.

Oracle BPEL PM implementation Oracle BPEL PM offers no direct support for this pattern, however some specific implementations are given in [18].

3.20 WP19 -Cancel Activity

Description An enabled activity is disabled, i.e. a thread waiting for the execution of an activity is removed.

Selected WPSL specification Figure 49 demonstrates the default WPSL notation of the Cancel Activity pattern.

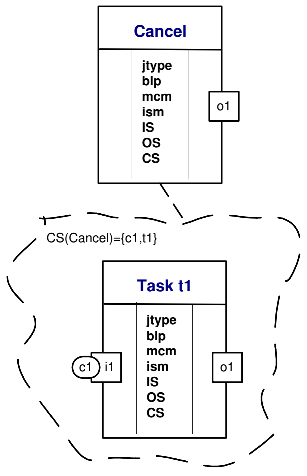


Fig. 49. WP19 -Cancel Activity. The WPSL notation.

Execution of task *Cancel* results in the termination of task $t1$ and removal of messages from its incoming channels. The locations from which the messages should be removed are specified in the cancellation set of task *Cancel*.

The definition of the Cancel Activity pattern treats a task as an atomic action and requires messages to be removed only from the incoming channels and only supports the cancelling of enabled tasks. The WPSL notation allows the scope of the cancellation to be extended. First of all, it incorporates the internal state of a task into the cancellation. Secondly, by extending the cancellation set, an arbitrary set of tasks and channels can be specified, thus allowing to cancel a region instead of just a single task.

Staffware implementation Staffware supports this pattern by means of a *withdraw* as illustrated in Figure 50(a). A trigger arriving on the top input of step A withdraws this task, so that it won't be executed any more. In terms of WPSL this corresponds to cancelling task A.

Oracle BPEL PM implementation An Oracle BPEL PM provides support for this pattern by means of a scope (see Figure 51(a)), a fault and a fault

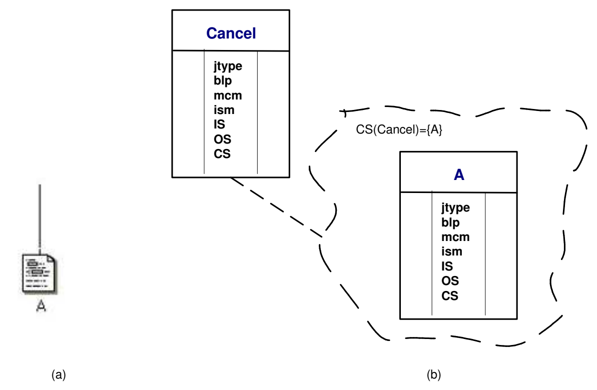


Fig. 50. Staffware implementation of WP19.

handler. The scope is used as wrapper for an activity that should be cancelled. The fault handler is needed to catch the fault message which is to be thrown when the client cancels the activity. The corresponding WPSL notation is shown in Figure 51(b).

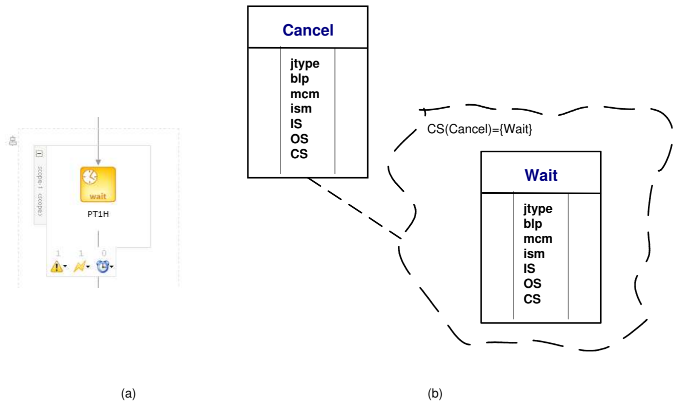


Fig. 51. Oracle implementation of WP19.

3.21 WP20 -Cancel Case

Description A case, i.e. workflow instance, is removed completely (i.e., even if parts of the process are instantiated multiple times, all descendants are removed).

Selected WPSL specification Figure 52 demonstrates the Cancel Case pattern.

Execution of task *Cancel* results in the cancellation of a case, i.e. all messages are removed from all channels and all active tasks are terminated. The range of cancellation is specified in the cancellation set of the *Cancel* task.

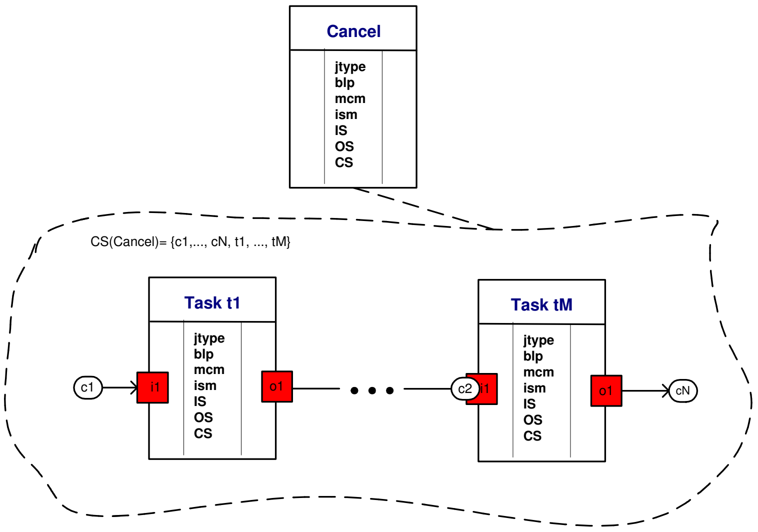


Fig. 52. WP19 -Cancel Case. The WPSL specification.

Staffware implementation Staffware does not support this pattern directly. However, it allows cancelling sub-cases by using the withdraw action of the sub procedure (see [24]).

Oracle BPEL PM implementation Oracle BPEL PM supports this pattern directly by means of the ⟨*terminate*⟩ construct as it is shown in Figure 53 (a). The execution of the ⟨*terminate*⟩ activity leads to cancelling of the whole process instance. Figure 53 (b) shows the corresponding WPSL notation for cancelling the case.

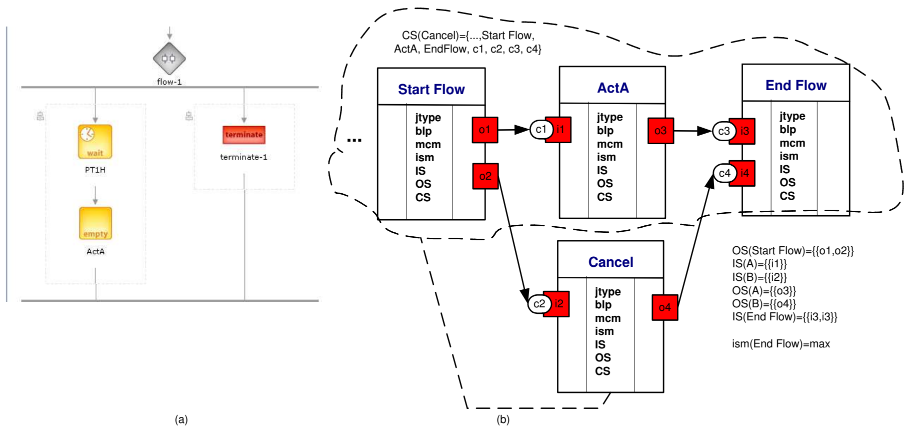


Fig. 53. Oracle implementation of WP20.

4 Lessons Learned

In Section 3 we analyzed the set of classical control-flow patterns, which resulted in the following observations:

- The definitions of patterns were formulated under the validity of implicit assumptions. The lack of precision in pattern definitions leads to an ambiguous interpretation of the patterns as it was illustrated using example models in Staffware and Oracle BPEL PM;
- Expressing the control-flow patterns in terms of WPSL, which forces to think of the behavior inherent to a pattern from the perspective of all entity attributes, allows for the definition of patterns to be made more precise;
- Depending on the value of the attributes set for the modelling entities, a single pattern may have different notations which correspond to the same behavioral pattern or to be treated as different behavioral patterns depending on the values chosen for the variations points.

As we have indicated, the points of variation identified for each of the patterns serve as classification criteria for distinguishing different variants of the same pattern. For instance, the basic pattern Sequence, which is supported by all workflow systems evaluated in [14], has several variants. It is thus important to distinguish the characteristics of the channels used to model this pattern, specifying explicitly the maximal channel capacity indicating the number of messages the channel is able to store at once; the minimal channel capacity and the message consumption mode specifying how many messages is required for enabling tasks in the sequence; the blocking mode of the output ports of tasks in a net defining whether the tasks producing messages may complete when the maximal channel capacity has been reached.

Although such a classification could potentially lead to an explosion in the number of patterns, it offers the potential to increase the precision of the pattern definitions significantly, thus avoiding pattern misinterpretations. Moreover, the properties of the basic model entities that are captured in WPSL are orthogonal. By explicitly identifying the various dimensions and by distinguishing all combinations of values from different dimensions we can obtain all possible variants of control-flow patterns and thus guarantee the completeness of the patterns in the light of the identified dimensions.

WPSL is a powerful tool for the analysis of workflow systems. So far, we have analyzed the modelling capabilities offered by Staffware and Oracle BPEL PM using WPSL as a means of comparing the expressiveness of the tool implementations of the control-flow patterns. Note that WPSL can be easily used for the analysis of other workflow systems. To illustrate this, we will show the mapping of the XOR-join and the AND-join constructs implemented in COSA and YAWL to WPSL. These are the systems which were also mentioned in Section 1. The mapping of process modelling languages to WPSL allows us not only to reason about features of specific languages but also to compare them while using WPSL as an evaluation tool.

Figure 54 depicts the mapping of the XOR-join construct to WPSL (see models (a) and (b) respectively). COSA activities A, B and C and states $s1$ and $s2$ correspond to WPSL tasks A, B and C and channels $s1$ and $s2$ respectively. The maximal capacity of channels in COSA is bounded at 1, therefore the channels are safe. In COSA tasks A and B can not be enabled if state $s1$ is not empty. This corresponds to the blocked mode set for all output ports in the GWF-net.

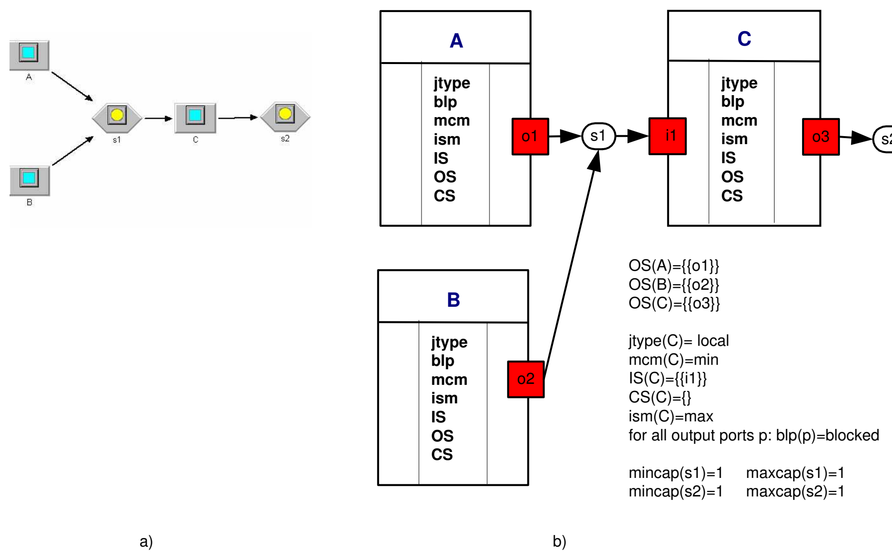


Fig. 54. XOR-join. Mapping of Cosa to WPSL.

Figure 55 shows the mapping of the COSA AND-join construct to WPSL (see models (a) and (b) respectively). COSA activities A, B and C and states $s1$, $s2$ and $s3$ correspond to WPSL tasks A, B and C and channels $s1$, $s2$ and $s3$ respectively. Activity C requires both inputs from states $s1$ and $s2$, which corresponds to the synchronization of incoming threads. Such AND-join behavior is denoted in WPSL by means of the input sets of task C. Similar to the description of XOR-join implementation in COSA, this model corresponds to a GWF-net with safe channels, blocked output ports, and maximal input selection mode. The enabling of tasks is based on the locally-available information.

Figure 56, Figure 57 and Figure 58 depict the mapping of the XOR-join, AND-join and OR-join implemented in YAWL to WPSL respectively. In contrast to COSA, YAWL allows an unlimited number of messages to be stored in places, therefore channels in the GWF-net are chosen to be unbounded. Since the maximal capacity of the channels can never be reached, all output ports produce messages in the open mode. The input selection mode for all joins is set to maximal, meaning that the maximal set of enabled input ports from the perspective

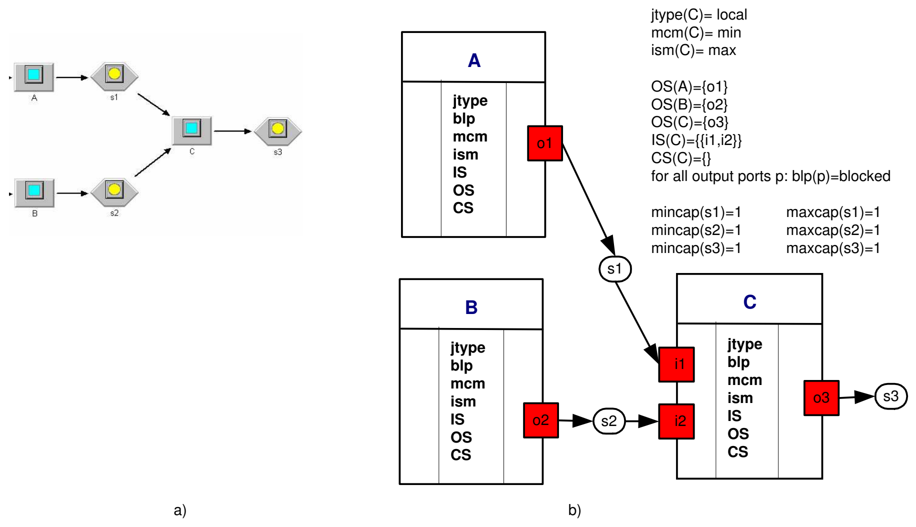


Fig. 55. AND-join. Mapping of Cosa to WPSL.

of set inclusion is selected by task C. The join logic is incorporated into the input sets of task C, the enabling of which is based on local and future states for XOR-join and AND-join, and OR-join respectively. The messages are consumed from channels in the minimal mode, i.e. one message is consumed from every channel selected for the message consumption.

We have shown that WPSL can also be used for interpreting functionality of such systems as COSA and YAWL and thus is generic enough to serve as a point of reference for comparing modeling languages adopted by workflow systems.

5 Related Work

The concept of patterns has been originated by Christopher Alexander, who defined them in the architectural context. In the early nineties, the idea of patterns became popular in the object-oriented software community. This is evidenced by the 23 design patterns by Gamma [12], and their numerous successors, such as patterns for knowledge and software reuse by Sutcliffe [25], design patterns in communication software by Rising [21], framework patterns by Pree [19], etc. In contrast to the generic patterns, sets of language-specific pattern languages (UML, Smalltalk, XML, Python, etc.) have been discovered and documented. Although there is still a lot of interest to object-oriented patterns, a new pattern trend focuses on reliable, scalable, efficient, parallel and distributed programming [**?**,**?**].

Furthermore, some work has been done on formalizing organization, process, analysis, and business-related patterns. Among them analysis patterns by Fowler [11]; enterprise architecture patterns by Beedle [8]; framework process patterns by Carey [10]; patterns for e-business [6], which focus on Business patterns, In-

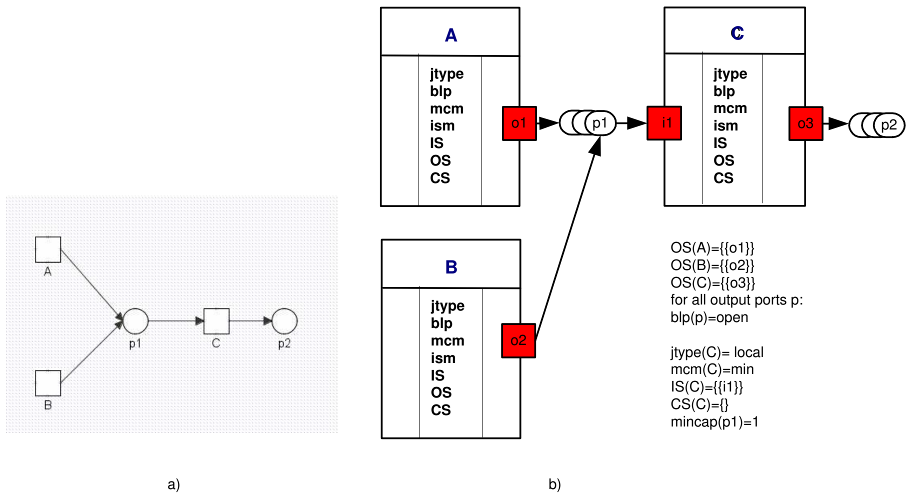


Fig. 56. The XOR-join. Mapping of YAWL to WPSL.

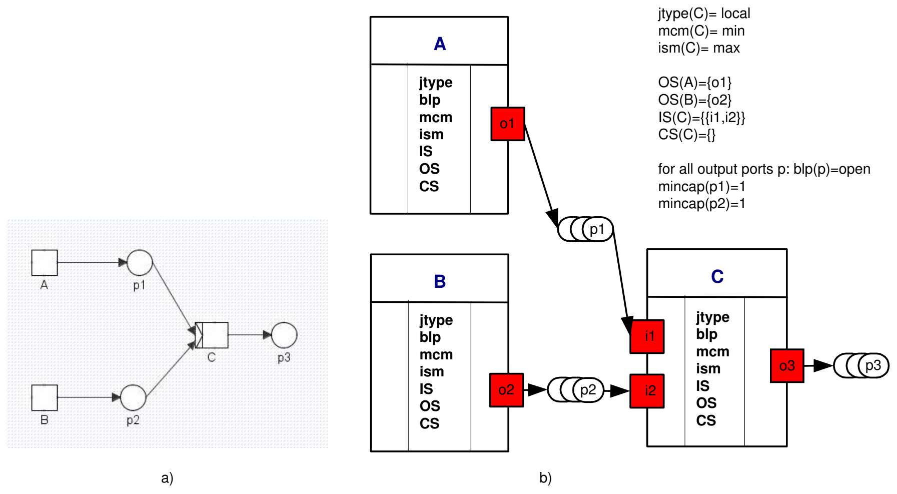


Fig. 57. The AND-join. Mapping of YAWL to WPSL.

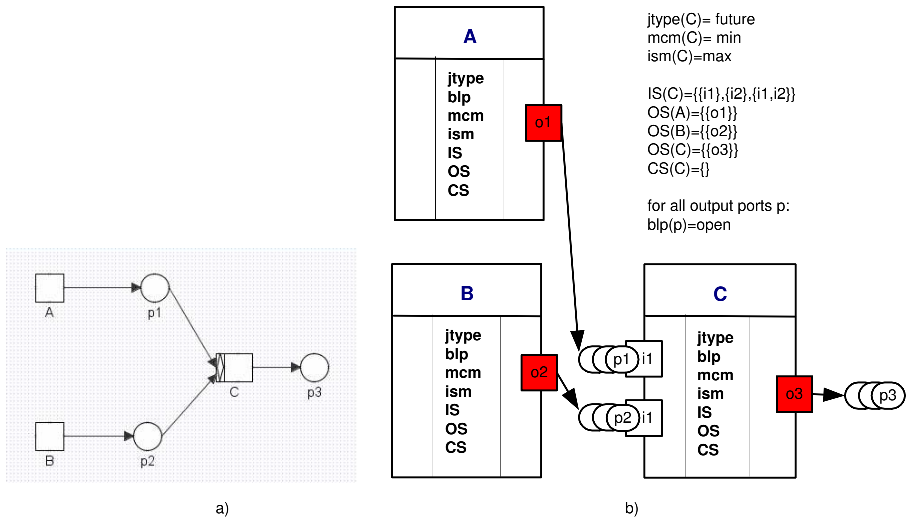


Fig. 58. The OR-join. Mapping of YAWL to WPSL.

tegration patterns, and Application patterns; and business patterns at work [25], which use UML to model a business system; process patterns [7].

The recent *Workflow Patterns initiative* [4] has taken an empirical approach to identifying the most common control constructs inherent to modelling languages adopted by workflow systems. In particular, a broad survey of modelling languages resulted in 20 workflow patterns being identified [14]. The collection of patterns was originally limited to the control-flow perspective, thus the data, organizational and application perspectives were missing. In addition, the set of control-flow patterns was not complete since the patterns were gathered non-systematically: they were obtained as a result of an empirical analysis of the modelling facilities offered by selected workflow systems.

The first shortcoming has been addressed by means of the systematic analysis of data and resource perspectives and resulted in the extension of the collection of the control-flow patterns by 40 data patterns and 43 resource patterns [23, 22]. The issue of the incompleteness of the control-flow patterns we have addressed in this paper and resolved it by means of the systematic analysis of the classical control-flow patterns against WPSL. Note that we are also working on revising the current set of the control-flow patterns and extending it with new patterns.

Our work is related to investigations into the expressive power and suitability of workflow languages, cf. [14]. Note that the term "workflow patterns" has been also addressed by other authors from a different perspective. In [26] a set of workflow patterns inspired by Language/Action theory specifically aiming at virtual communities was introduced. Patterns at the level of workflow architectures rather than control flow were given in [17].

In [20] authors examined the suitability of pi-calculus for formalizing the control-flow patterns. Such a formalization does not guarantee an unambiguous description of a pattern since the authors did not take into account the implicit assumptions made in the original pattern definitions and thus analyzed specific interpretations of the patterns excluding other interpretations.

Many workflow systems and standards such as XPDL, UML, BPEL, XLANG, WSFL, BPML, and WSCI were evaluated from the perspective of the classical control-flow patterns, a summary of which is available at [4]. Although multiple standards have been proposed, none of them have become a common standard for modeling business processes. BPEL4WS 1.1 proved to be incomplete and weak, and has been gradually refined and extended. To increase the capabilities of BPEL, new accompanying standards are being proposed to extend BPEL capabilities in version 2.0 (BPELJ, BPEL4People, and BPEL-SPE). WS-BPEL 2.0 is being constantly revisited, examined and extended, and thus has not been finalized so far. In this setting, WPSL could be utilized to evaluate the current BPEL functionality, identifying strengthes and weaknesses, hence serving as a starting point for improvement.

6 Conclusions

In this paper, we introduced the Workflow Pattern Specification Language and used it as an evaluation strategy for comparing the process modelling languages employed by the workflow systems COSA, YAWL, Staffware, and Oracle BPEL PM, to show that they interpret the same concepts differently while imposing syntactic restrictions on their languages. Furthermore, we showed that ambiguous definitions of the control-flow patterns can be made more precise by the means of setting the structural and behavioral properties of the basic process modelling entities in more detail. One of the immediate effects of doing this is the extension of the pattern collection by pattern variants. The orthogonality of the dimensions used for the classification of the basic modelling entities guarantees the completeness of the patterns within the scope of the identified dimensions.

The practical value of WPSL is multifaceted: it can be used as a means for expressing the control-flow patterns more precisely, and as an analysis tool for comparing the modelling languages adopted by various workflow systems. Furthermore, WPSL can be used by workflow systems developers to prevent them from imposing restrictive constraints on workflow specifications and it can help in identifying which pattern variants a system needs to support.

In the light of the presented work, we plan to classify the various pattern variants identified by means of WPSL and develop a taxonomy for the control-flow patterns. Further research is to be conducted into the identification of meaningful combinations of the patterns.

References

1. W.M.P. van der Aalst, A.P. Barros, A.H.M. ter Hofstede, and B. Kiepuszewski. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer-Verlag, Berlin, 2000.

2. W.M.P. van der Aalst, J. Desel, and E. Kindler. On the Semantics of EPCs: A Vicious Circle. In M. Nüttgens and F.J. Rump, editors, *Proceedings of the EPK 2002: Business Process Management using EPCs*, pages 71–80, Trier, Germany, November 2002. Gesellschaft für Informatik, Bonn.
3. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language (Revised Version). QUT Technical report, FIT-TR-2003-04, Queensland University of Technology, Brisbane, 2003.
4. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns Home Page. http://www.workflowpatterns.com.
5. 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.
6. J. Adams, S. Koushik, G. Vasudeva, and G. Galambos. *Patterns for e-Business. A Strategy for Use*. IBM Press, 2001.
7. S.W. Ambler. *Process Patterns*. Cambridge University Press, 1998.
8. M.A. Beedle. *Enterprise Architecture Patterns*. Cambridge University Press, 1998.
9. BPEL. *Wikipedia. The free encyclopedia*. http://en.wikipedia.org/wiki/BPEL.
10. J. Carey and B. Carlson. *Framework Process Patterns*. Addison Wesley Longman, 2001.
11. M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, Massachusetts, 1997.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison Wesley, Reading, MA, USA, 1995.
13. Object Management Group. *OMG Unified Modeling Language, Version 1.4*. OMG, http://www.omg.com/uml/, 2001.
14. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2003. Available via http://www.workflowpatterns.com.
15. E. Kindler. On the Semantics of EPCs: A Framework for Resolving the Vicious Circle (Extended Abstract). In M. Nüttgens and F.J. Rump, editors, *Proceedings of the GI-Workshop EPK 2003: Business Process Management using EPCs*, pages 7–18, Bamberg, Germany, October 2003. Gesellschaft für Informatik, Bonn.
16. E. Kindler. On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. In J. Desel, B. Pernici, and M. Weske, editors, *International Conference on Business Process Management (BPM 2004)*, volume 3080 of *Lecture Notes in Computer Science*, pages 82–97. Springer-Verlag, Berlin, 2004.
17. G. Meszaros and K. Brown. A Pattern Language for Workflow Systems. In *Proceedings of the 4th Pattern Languages of Programming Conference*, Washington University Technical Report 97-34 (WUCS-97-34), 1997.
18. N.A. Mulyar. Pattern-based Evaluation of Oracle-BPEL (v.10.1.2). Technical report, Center Report BPM-05-24, BPMcenter.org, 2005.
19. W. Pree. *Framework patterns*. SIGS Books, 1996.
20. F. Puhlmann and M. Weske. Using the pi-calculus for Formalizing Workflow Patterns. In *Proceedings of Business Process Management: 3rd International Conference*, Nancy, France, 2005.
21. L. Rising. *Design Patterns in Communication Software*. Cambridge University Press, 2000.
22. N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow Data Patterns. QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

23. N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow Resource Patterns. BETA Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven, 2004.
24. Staffware. Workflow Patterns According to Staffware. Technical report. (Available via http://is.tm.tue.nl/research/patterns/download).
25. A. Sutcliffe. *Patterns for Knowledge and Software Reuse.* Lawrence Erlbaum Associates Inc., 2002.
26. H. Weigand, A. de Moor, and W.J. van den Heuvel. Supporting the Evolution of Workflow Patterns for Virtual Communities. *Electronic Markets*, 10(4):264–271, 2000.
27. WFMC. Workflow reference model. Technical report, Workflow Management Coalition, Brussels, 1994.
28. WFMC. Workflow Management Coalition Workflow Standard: Workflow Process Definition Interface – XML Process Definition Language (XPDL) (WFMC-TC-1025). Technical report, Workflow Management Coalition, Lighthouse Point, Florida, USA, 2002.
29. M.T. Wynn, D. Edmond, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Achieving a General, Formal and Decidable Approach to the OR-join in Workflow using Reset nets. QUT Technical report, FIT-TR-2004-02, Queensland University of Technology, Brisbane, 2004.