CONFIGURABLE WORKFLOW MODELS

FLORIAN GOTTSCHALK

Department of Technology Management, Eindhoven University of Technology, P.O. Box 513,
5600MB Eindhoven, The Netherlands.
f.gottschalk@tue.nl

WIL M.P. VAN DER AALST

Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513,
5600MB Eindhoven, The Netherlands.
w.m.p.v.d.aalst@tue.nl

MONIQUE H. JANSEN-VULLERS

Department of Technology Management, Eindhoven University of Technology, P.O. Box 513,
5600MB Eindhoven, The Netherlands.
m.h.jansen-vullers@tue.nl

MARCELLO LA ROSA

Faculty of Information Technology, Queensland University of Technology, 126 Margaret St,
Brisbane, QLD 4000, Australia.
m.larosa@qut.edu.au

Workflow modelling languages allow for the specification of executable business processes. They, however, do typically not provide any guidance for the adaptation of workflow models, i.e. they do not offer any methods or tools explaining and highlighting which adaptations of the models are feasible and which are not. Therefore, an approach to identify so-called configurable elements of a workflow modelling language and to add configuration opportunities to workflow models is presented in this paper. Configurable elements are the elements of a workflow model that can be modified such that the behavior represented by the model is restricted. More precisely, a configurable element can be either set to enabled, to blocked, or to hidden. To ensure that such configurations lead only to desirable models, our approach allows for imposing so-called *requirements* on the model's configuration. They have to be fulfilled by any configuration, and limit therefore the freedom of configuration choices. The identification of configurable elements within the workflow modelling language of YAWL and the derivation of the new "configurable YAWL" language provide a concrete example for a rather generic approach. A transformation of configured models into lawful YAWL models demonstrates its applicability.

Keywords: Business Process, Workflow, Configuration, Reference Modelling, C-YAWL.

1. Introduction

Legal obligations, the computer or enterprize systems in use, and best-practice force many companies to organize their secondary or supporting business processes in very similar ways. Typical examples for such business processes are purchasing, reporting, recruitment,

CRM, payroll, or call-center processes.[1] Enterprize and workflow systems are used to support the execution of such business processes by guiding and monitoring the process instances throughout the company. The specification of a business process that enables its automated execution in such a system is called a *workflow model*. If such a model exists, the particular business process is also called a workflow.[2]

Vendors of enterprize or workflow systems as well as consultants typically offer generic *reference process models* together with their solutions. These are typically defined on a conceptual level and help understanding how business processes are supported by the particular systems.[3,4,5,6]

Still, secondary business processes will rarely be organized in exactly the same way among companies. Instead, minor, or sometimes even major, adaptations are required to tailor the process to the local environments like local law or company cultures. To support these different environments, larger enterprize systems often offer more than one way to execute a business process. The selection of the used variant must be made only during the implementation of the process/system.

The different process variants should also be reflected and selectable in the reference process models. However, the languages used today for the specification of reference process models and workflows, such as EPCs[7], BPML[8], Protos[9], Staffware[10,11], SAP WebFlow[12], YAWL[13,14] etc., do not provide any dedicated support for this. In this paper we will therefore present an approach to extend common workflow modelling languages with a notion of configuration, allowing for the enabling or disabling of actions in such models. In this way, we enable the integration of several variants of a business process into a single configurable workflow model. Before the workflow can be executed, the proper variant must be selected by configuring the model. We thus distinguish the three phases: (1) *build time* of the model, i.e. the time while the configurable model incorporating all variants of the process was build, (2) *configuration time*, i.e. the time when a particular workflow variant is selected, and (3) *run time*, i.e. the time when process instances are
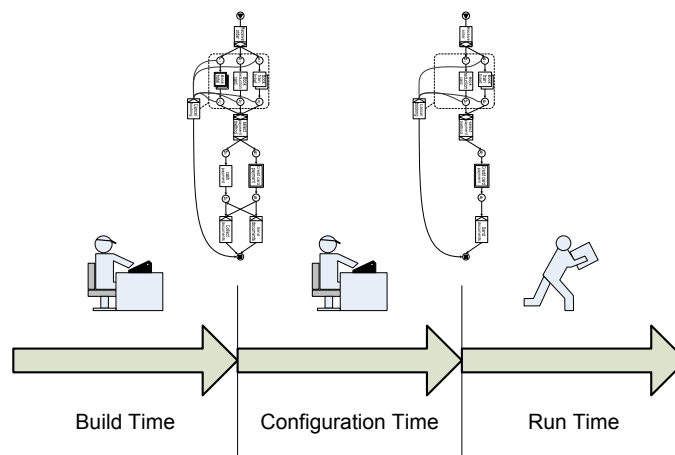

Fig. 1. The model's *build time* is followed by the *configuration time* before the process is enacted during *run time*

executed using the configured model (Figure 1).

Of course, configuration choices can be integrated into workflow models as normal run-time choices. But this solution has two drawbacks. On the one hand the additional choices integrated into the model look like run-time choices although the decision is already made before process instances are started, i.e. there is no decision to be made during run-time. On the other hand, they typically increase the model's size dramatically. Therefore, configurable workflow models transfer these configuration choices to an additional configuration layer, allowing not only for a clear distinction between configuration and run-time choices, but also for the creation of run-time models without the model elements which are already "dead" before any instance of the process has been initiated. Compared to the original workflow modelling language, the complexity of such a configurable modelling language increases of course. But, as the additional configurable elements are only relevant at configuration time, the target group of process designers that is confronted with these additional elements is limited compared to the overall number of users of the model.

The remainder of this paper is structured as follows. In Section 2 will first provide an overview on related work. In Section 3 we will then depict an universal approach to add configuration layers to any workflow modelling language, using a concept of ports as configuration points which can be enabled, blocked, or hidden. In Section 4 we will use YAWL as an example workflow modelling language to depict in detail how the approach can be applied to a concrete language. Although our approach is quite generic and applies to most process modelling languages, we need to select a concrete notation to explain our ideas. YAWL was selected because it supports most of the workflow patterns.[15] Hence, many languages can be seen as a subset of YAWL, thus making the results applicable to a wide variety of languages. We will start Section 4 with an introduction into YAWL models as hierarchical workflow specifications composed of extended workflow (EWF) nets. In the second part of Section 4 we will define configurable EWF (C-EWF) nets by identifying their configurable elements and formally specifying their configurations. In the section's third part we will briefly sketch how the same approach can be applied to develop configurable workflow specifications but without going into the technical details as for C-EWF nets. To demonstrate the applicability of these concepts, Section 5 explains how the configuration of a configurable YAWL model can be transformed into a lawful YAWL model and introduces the corresponding software tool. To show the general applicability of the approach for workflow languages Section 6 briefly highlights how it can be applied to the language used by SAP's workflow engine (SAP WebFlow) and to the workflow standard BPEL. Section 7 will conclude the paper with a short summary and an outlook on open issues.

2. Related Work

Configurable workflow models add a configuration layer to the control flow of workflow models. This layer enables the alteration of the pre-defined case routings through a workflow system during a configuration phase. Besides relating to the broad research areas of workflows, business process enactment, and business process automatization which range

4   *F. Gottschalk, W.M.P. van der Aalst, M.H. Jansen-Vullers, and M. La Rosa*

form environmental and social influences of automated business processes to the transaction management within workflow systems, this topic can in particular be related to (1) the business process management research area of adapting reference models, (2) to ideas of managing software configurations, and (3) to approaches that support the process flexibility.

Adapting Reference Process Models

Reference process models depict in a general manner how processes can or should be performed. They are often considered as best practice, and usually modelled on a conceptual level.[4,5] To avoid developing business process models from scratch, it is also suggested to regard reference process models as masters or templates which can be tailored to individual requirements.[16] Thus, although rarely cited in the literature on reference process models, also repositories of workflow templates that are delivered with or for workflow engines can be considered as reference process models.[17]

To adapt a reference model to individual requirements, either additional elements, routings and information can be added to the model, or the existing elements can be (re-)configured which includes the elimination of existing elements.[18,19] A workflow definition language that focusses on the re-usability of pre-defined workflow blocks when creating new workflow variants, i.e. when adding new content to the model, is suggested by Blin et al.[20] When developing configurable workflow models, we however focus on arranging the components of the existing model without adding any additional content. That means that in configurable workflow models different model variants can only be achieved if all the process variants have been integrated within the model beforehand.[21]

Extensions to conceptual process modelling languages that allow for defining such an integration and configuring it are suggested by several authors.

- Becker et al.[18] suggest the creation of different views on process models for deriving different process variants.
- Rosemann and van der Aalst[22] intuitively extend certain elements in the business process modelling language of Event-driven process chains[23] with configuration options.
- Soffer et al.[24] relate different application scenarios via attributes in so-called Object-Process Diagrams to different modules of Enterprize Systems.
- Puhlmann et al.[25] use attributes from feature diagrams (a technique to represent system properties[26,27,28]) to enable/disable sub-processes in UML hierarchies or to parameterize decision nodes in BPMN by adding a condition which can be evaluated.
- Czarnecki and Antkiewicz[29] explain the usage of feature diagrams to identify elements of UML activity diagrams which should be present or removed from the model.

Efficiency benefits of using configurable process models during the implementation of enterprise systems are highlighted by many of these authors. But none of the approaches

uses a modelling language designed for workflow models and thus produces directly executable models.

An approach to enhance the re-usability of classic workflow models is suggested by Karastoyanova et al.[30] who parameterize BPEL processes to select a desired web-service at runtime from a range of such services. However, this selection is only performed locally and thus neglects the influences among the selections of web-service at different stages during the process execution as well as any form of routing related to splits, joins, cancellations etc.

Dreiling et al.[31] indicate a potential applicability of the approach of Rosemann and van der Aalst to executable workflow models such that the configuration can be enacted using a workflow engine. We follow up on this here not only by presenting a general approach to make workflow modelling languages configurable, but also by applying these ideas to a concrete workflow modelling languages and implementing a tool that demonstrates that it is possible to derive a configured workflow specifications which is executable in the classical workflow engine.

Managing Software Configurations

To reduce the time, the effort, the costs and the complexity of software creation and maintenance, also software families are often created from shared sets of software assets like parameterized libraries. Similar to configurable workflow models, software families thus rely on the reuse of existing solutions (models/assets) instead of building new models/pieces of software from scratch. Software Configuration Management (SCM) is a methodology to control and manage software development projects. It consists of a set of activities to identify the assets that need to be changed and their relationships, to control the products versions and the changes imposed, and to audit and report the changes made.[32]

Tools as the Adele Configuration Manager[33] or CoSMIC[34] support the definition of dependencies or constraints among artifacts composing a software family. Using attributes defined on the software artifacts, they express dependencies and constraints in first-order logic languages. To be valid, every configuration must satisfy all the constraints. Thus, although such SCM tools are usually used for software mass customization whereas configurable workflow models are configured manually, configurable workflow models can use the same methods to ensure the validity of configurations.

Process Flexibility

Research on process and workflow flexibility focusses on the effects of a change in a workflow specification on process instances already running in the system. This becomes important if the execution of single workflow instances takes very long and thus a transformation into the new specification is unavoidable, or if individual instances frequently require ad-hoc treatment. Systems tackling these problems of switching from one configuration to another are also often called configurable, re-configurable or adaptive workflow systems[35,36,37,38,39,40,41], but they typically neglect the preceding problem of how the

workflow model itself can be easily and safely changed which is the focus of this research.

3. Making workflow models configurable

As depicted in the introduction, configurable workflow modelling languages are useful whenever an individual workflow variant should be derived from a more general model. For this, configurable workflow modelling languages enable the restriction of the behavior of workflow models in a controlled manner. This sections outlines a general approach for the development of such configurable workflow modelling languages.

By using the term "workflow models", we explicitly focus on executable business process models, although the approach might be applicable to non-executable modelling languages as well. We assume that every workflow modelling language that explicitly depicts the flow of cases, i.e. executed workflow instances, through a system with tasks, steps, activities, functions or similar concepts of performed actions can be made configurable. Before defining such a configurable workflow modelling language, it is however required to identify what a configuration of a model in a particular language is.

In our previous research on configurable process models[21,42], we identified two general applicable methodologies to configure, i.e. restrict, a workflow model, namely *blocking* and *hiding*. The insight that these are the two basic configuration operations was obtained by a systematic study of inheritance notions in the context of business processes[43,44]. If an action in a workflow is *blocked*, it cannot be executed. The process will never continue after the action and thus never reach a subsequent state or any subsequent action. If an action is *hidden*, its performance is not observable, i.e. it is skipped and consumes neither time nor resources. But the process flow continues afterwards and subsequent actions will be performed. For that reason we also talk about a *silent action*[a] or simply about *skipping the action*. If an action in a workflow model is neither blocked nor hidden, then we say it is *enabled*, which refers to its normal execution. The enabled action is performed as it would be in a classic, un-configurable workflow.

As a rule of thumb this means that if the performance of an action is not desired and the action is not mandatory for subsequent steps, than hiding is the preferred configuration method. If a whole sequence or line of actions is not desired or if the non-desired action is mandatory for subsequent actions, blocking is typically the preferred configuration method.

To develop configurations for a particular language, it is required to identify those element types of the language which represent actions that can be performed. These are usually elements like activities, functions, steps, etc. For an action to be executed, it must be triggered. Triggers are typically represented by arcs pointing into an action. However, the meaning of these arcs leading into the action varies not only among different workflow modelling languages but also within a single workflow modelling language because of different joining patterns for preceding paths leading into the action. For example, some actions require that all preceding paths are completed for the action to be triggered (AND-

---

[a]The term silent action comes from concurrency theory where silent actions are denoted as τ and form the basis for equivalence notions such as branching bisimulation.
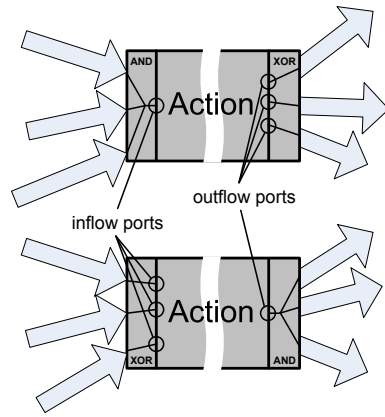
Fig. 2. The number of ports of an action depends on its joining and splitting behavior
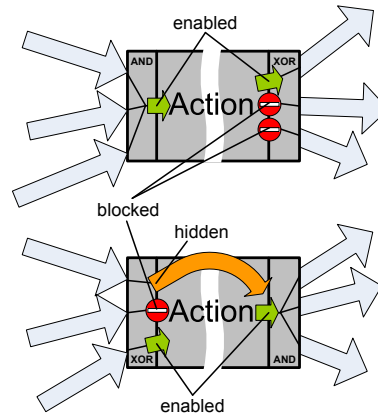
Fig. 3. Ports can be enabled or blocked, and in case of an inflow also be hidden

join), whereas other actions can be triggered via each arc pointing into the action (XOR-join). We call each combination of incoming paths through which an action can be triggered an *inflow port* of the action (see the left side of Figure 2). Thus, an action with an AND-joining behavior for the incoming paths has just a single inflow port whereas a task with an XOR-joining behavior has an inflow port for each incoming path.

After an action has completed, it releases the particular case via the arcs leaving the action. Also here the number of triggered paths depends on the semantics specified for the particular action. An action with an AND-splitting behavior triggers all outgoing paths, whereas an action with an XOR-splitting behavior only triggers one subsequent path. Of course there can also be semantics allowing the triggering of a specific number of paths (OR-split). Aligned with the specification of inflow ports, we say that each case can leave the action only through one distinct *outflow port*, but then triggers all paths connected to this outflow port (see the right side of Figure 2).

Ports are the elements which can be enabled, blocked, or hidden, i.e. they are in fact the configurable elements. Every port can be enabled or blocked while inflow ports can also be hidden.

An enabled inflow port allows the triggering of the action through this port. However, if an inflow port is blocked, no cases can flow into the action through this port. The triggering of the action via the inflow port is inhibited. If an action is triggered via a hidden inflow port, the action itself is skipped and the case is directly forwarded to one of the outflow ports (usually but not necessarily a default output port; see Figure 3).

If an outflow port is enabled, the action can select this port as the port through which the case is released. If an outflow port is however blocked, the port cannot be selected as the used outflow port. Instead another enabled outflow port must be selected. Thus, the blocking of an outflow port inhibits the performance of actions subsequent to the port. However, as cases should always be able to leave a triggered action, at least one outflow port must always be enabled. The hiding of an outflow port is impossible because outflow ports trigger paths instead of actions. A path just forwards the case to the next action without

containing any action itself. Thus, a path contains nothing that can be skipped (and any subsequent action should be hidden via its own input ports).

By deriving ports from the definition of a workflow modelling language instead of defining them as elements which have to be added to the workflow models of the language, each model can serve as the basis for a configurable model without any change. Such a model represents the "Least Common Multiple" (LCM) of all possible model variants.[45] It contains the maximal possible behavior which can be achieved by enabling all variants, i.e. configuring all ports as enabled. We call this initial model therefore the *basic model* whose behavior can be restricted by hiding or blocking of selected ports.

To transform these configuration decisions into a model executable in the traditional workflow engine, blocked elements and all their dead successors must be removed from the model and hidden elements must be replaced by shortcuts.

Obviously, not all models resulting from such a transformation conform to the definition of the used modelling language or represent desirable behavior. For example, blocking too many ports or a "wrong" port might result in an unconnected net which for many workflow modelling languages means that the model would become syntactically invalid. In a similar way hiding of essential actions can prevent the practicability of the depicted process and lead to a semantically incorrect model. To avoid the occurrence of such situations, a configurable model must not only consist of the basic model, but also of a set of requirements restricting the set of permitted configurations and therefore ensuring both syntactical and semantical validity of models. An example for a syntactical motivated requirement would be "Each action must have at least one enabled port which allows the outflow of cases"; an example for a semantically motivated requirement would be "If it is possible to pay in installments, it must be possible to pay by credit card" (because the installments are deducted from the credit card account), or better "If a port is enabled that allows cases to flow into the action *Pay in installments*, then the port allowing for cases flowing into the action *Pay by credit card* must be enabled as well". That means, although the requirement is semantically motivated, it still should be formulated in terms of the model's port configuration.

For a better understanding, we have presented these example requirements in a rather informal natural language. However, the configurable modelling language must be able to test if a configuration of a model satisfies all requirements as otherwise the transformation of the model should not be performed. Therefore, a formal specification of requirements is indispensable. We suggest either the use of a subset of a programming language, or to formulate logical expressions composed of atomic expressions that test individual elements of the net or its configuration.

As mentioned above, the basic model uses the traditional, i.e. non-configurable version of the particular modelling language. Thus, assuming that all ports are enabled, it satisfies all of the language's syntactical requirements. It might however contain semantically conflicting elements if two distinct process variants exclude each other. For that reason, the assumption that all ports can be enabled at the same time is not always valid. Instead, it is required to explicitly specify a configuration for each port. Only if this *complete configuration* of all ports satisfies all requirements, it can be used to transform the configurable

workflow model into a configured net.

By requiring that every valid configurable workflow model contains at least one such valid and complete configuration as a *default configuration*, we ensure the existence of such a configuration. The default configuration also serves as a "starting point" for any individual configuration. In this way, configuring a configurable workflow model to individual requirements just means to modify those port configurations that need to deviate from the default configuration – usually a limited effort even if there are many configurable ports.

When developing the configurable workflow modelling language, it is important to note that workflow modelling languages often abstract from the most basic actions by grouping several actions into a single task, step, function etc. In these cases, it is reasonable to deviate from the pure concept of blocking and hiding single ports and instead subsume several configurable elements into language-specific configuration constructs.

To show how this approach can be applied to a concrete workflow language, we will develop a configurable YAWL (C-YAWL) in the following. C-YAWL will contain both configuration types: configurations of individual ports, as well as the subsuming of several ports into a YAWL-specific configuration construct.

4. Configurable YAWL

C-YAWL is based on the workflow modelling language of YAWL[14], extended with a worklet service architecture[46]. YAWL is chosen on the one hand due to its extensive support of workflow patterns[15,47], and on the other hand because the open-source licence of its editor and workflow engine enable the implementation of C-YAWL tools and workflows. In addition, the hierarchy created by the worklet service architecture creates to a certain extend a second modelling language which can serve as a further example.

For readers unfamiliar with YAWL, we first give a brief introduction into YAWL. This introduction is based on the work of van der Aalst and ter Hofstede[14] and summarizes the main definitions while focusing on the prerequisites needed for the development of C-YAWL. For more elaborated explanations the reader is referred directly to the original article[14]. Afterwards, we will develop and formally define configurable extended workflow (C-EWF) nets as the central configurable model in C-YAWL. We will conclude this section with outlining how the hierarchy in YAWL models can be extended into a configurable workflow specification.

4.1. *YAWL*

YAWL is a workflow modelling language inspired by Petri nets, but with several vital extensions and its own semantics. A *workflow specification* in YAWL is a set of *extended workflow nets* (EWF-nets) which form a hierarchy. Each EWF-net consists of conditions, which in Petri net terms can be interpreted as places, and tasks. By mapping some tasks of an EWF-net onto other EWF-nets within the workflow specification, the hierarchy is created, i.e., there is a tree-like structure where tasks can be decomposed into EWF nets. These "mapped" tasks are called composite tasks, "unmapped" tasks are called atomic tasks.
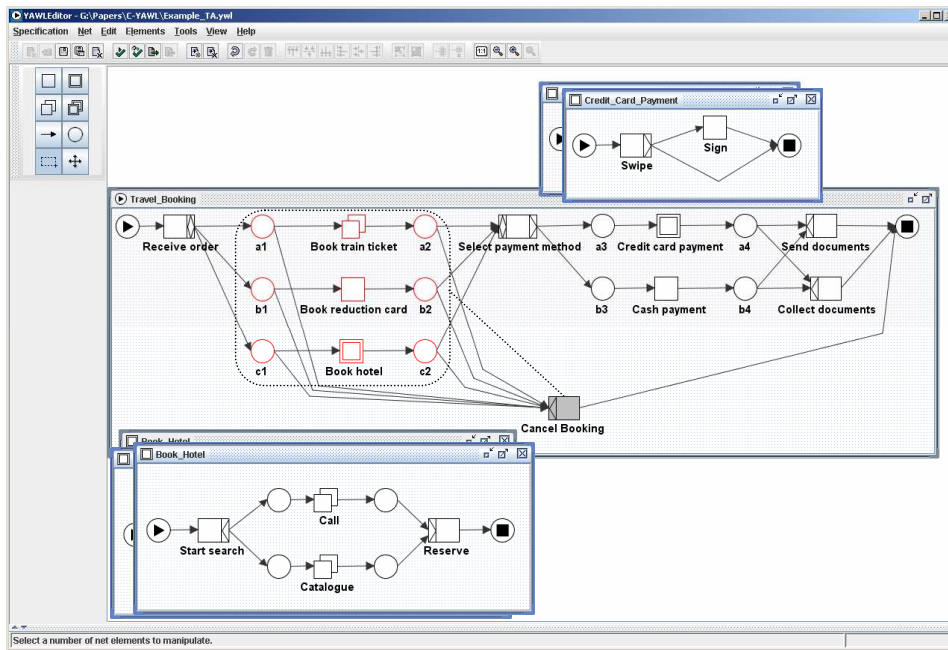
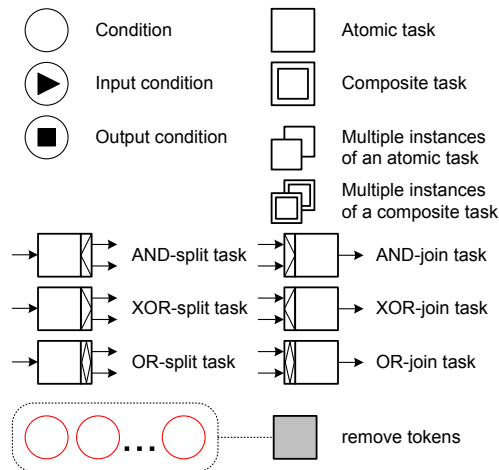Fig. 4. An example YAWL model for a travel booking process

Fig. 5. Symbols used in EWF nets.

Figure 4 shows an example for such a YAWL model while Figure 5 summarizes the graphical symbols for all the element types for EWF nets. The example depicts a booking and payment workflow for train travels. After an order has been received, multiple train tickets, a reduction card for train tickets, and/or multiple hotels can be booked. Until a

payment method has been selected, the booking can also be cancelled. Afterwards the travel has to be paid either in cash or by credit card before the documents can be either send to the customer or collected by him. The tasks "Book hotel" and "Credit card payments" contain further refinements in form of additional EWF nets.

Each EWF-net has exactly one unique input condition and one unique output condition. The control flow determines the flow of tokens through tasks and conditions. AND-, OR-, and XOR-joins and -splits determine the joining and splitting behavior before and after each task. AND-joins like the task *Reserve* in Figure 4 require tokens in all the conditions preceding the AND-join to enable the execution of the subsequent task, AND-splits like the task *Start search* in Figure 4 put tokens into all the post-conditions after the task has completed. Tasks with an XOR-join behavior, as e.g. the *Send documents* task in Figure 4, require a token in only one of the pre-conditions to be enabled, tasks with an XOR-split behavior, as the task *Select payment method*, put a single token into one of the post-conditions after the completion of the task. OR-joins, as in task *Select payment method*, allow a synchronizing merge of several process branches by enabling the subsequent task only if there is no chance that any tokens will arrive in unoccupied pre-conditions of the OR-join at any future point in time. OR-splits, as in task *Receive order*, enable a multi-choice, i.e. a selection of several post-conditions.

The specification of a cancellation region, as for the task *Cancel booking* in Figure 4, allows for the removal of all tokens from the conditions and running tasks within this region during the execution of the task to which the cancellation region is attached to. Independently of the total number of tokens in the conditions, it removes all tokens and therefore supports various cancellation patterns. In addition, tasks can be specified in such a way that they start in multiple instances. Examples are the *Book train ticket* and the *book hotel* tasks from Figure 4 which allow for the booking of multiple tickets or hotels. It is then possible to specify upper and lower bounds for the number of instances of the task that can be started. It can also be specified if instances can only be created at once when the task is started, i.e. statically, or if instances can be added dynamically while the task is running and the number of started instances is lower than the maximum number. The task's threshold value determines the number of instances that have to be completed to complete the task as a whole. As soon as the threshold value is reached, all remaining instances are terminated.

Formally an EWF-net can be defined as follows:

Definition 1 (EWF-net) *An extended workflow net (EWF-net) is a tuple* $(C, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ *such that:*

- C *is a set of conditions,*
- $\mathbf{i} \in C$ *is the input condition,*
- $\mathbf{o} \in C$ *is the output condition,*
- T *is a set of tasks,*
- $F \subseteq (C \setminus \{o\} \times T) \cup (T \times C \setminus \{i\}) \cup (T \times T)$ *is the flow relation,*
- *every node in the graph* $(C \cup T, F)$ *is on a directed path from i to o,*

- *$split : T \rightarrow \{AND, XOR, OR\}$ specifies the split behaviour of each task,*
- *$join : T \rightarrow \{AND, XOR, OR\}$ specifies the join behaviour of each task,*
- *$rem : T \nrightarrow I\!\!P(T \cup C \setminus \{\mathbf{i}, \mathbf{o}\})$ specifies the cancellation region for a task[b], and*
- *$nofi : T \nrightarrow I\!\!N \times I\!\!N^{inf} \times I\!\!N^{inf} \times \{dynamic, static\}$ specifies the multiplicity of each task (minimum, maximum, threshold for continuation, and dynamic/static creation of instances).*

The tuple (C, T, F) corresponds to a classical Petri net[48] where C (the set of conditions) corresponds to the set of places, T (the set of tasks) corresponds to the set of transitions, and F is the flow relation. Different to Petri nets, there are the special conditions $\mathbf{i}$ and $\mathbf{o}$ and tasks can be connected not only via places but also directly to each other by the flow relation. We counteract this "unstructuredness" by defining the extended set of conditions C^{ext} and the extended flow relation F^{ext} for EWF nets, adding the implicit condition $c_{(t_1, t_2)}$ between two tasks t_1, t_2 if there is a direct connection from t_1 to t_2. To navigate through an EWF net it is also useful to define the preset and postset of a node (i.e., of a condition or a task) as shown in the definition below.

Definition 2 (Implicit conditions, preset, postset)
Let $N = (C, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ be an EWF-net. Then $C^{ext} = C \cup \{c_{(t_1, t_2)} \mid (t_1, t_2) \in F \cap (T \times T)\}$ is the extended set of conditions and $F^{ext} = (F \setminus (T \times T)) \cup \{(t_1, c_{(t_1, t_2)}) \mid (t_1, t_2) \in F \cap (T \times T)\} \cup \{(c_{(t_1, t_2)}, t_2) \mid (t_1, t_2) \in F \cap (T \times T)\}$ is the extended flow relation. Moreover, auxiliary functions $\bullet_-, {}_-\bullet : (C^{ext} \cup T) \rightarrow I\!\!P(C^{ext} \cup T)$ are defined that assign to each node its preset and postset, respectively. For any node $x \in C^{ext} \cup T$, $\bullet x = \{y \mid (y, x) \in F^{ext}\}$ and $x\bullet = \{y \mid (x, y) \in F^{ext}\}$.

The four functions of the EWF net $split$, $join$, rem, and $nofi$ specify the properties of each task. As the names imply, the first two functions specify the splitting- and joining behavior for the tasks. rem specifies from which parts of the net the tokens should be removed. Note that the range of rem includes tasks and conditions, but tokens cannot be removed from input and output conditions. Removing tokens from a task corresponds to aborting the execution of that task. If a task is a composite task, its removal implies the removal of all tokens it contains. $nofi$ specifies the attributes related to multiple instances.

Whenever we introduce an EWF-net N we assume C, $\mathbf{i}$, $\mathbf{o}$, T, F, $split$, $join$, rem, and $nofi$ defined as $N = (C, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$. For simplification we also assume that $C = C^{ext}$ and $F = F^{ext}$, i.e. we only consider the extended net with implicit conditions. We use $\pi_1(nofi(t))$ to refer to the minimal number of instances initiated, $\pi_2(nofi(t))$ to refer to the maximal number of instances initiated, $\pi_3(nofi(t))$ is the threshold value, and $\pi_4(nofi(t))$ indicates whether it is possible to add instances while handling the other ones.

For convenience, we extend the functions rem and $nofi$ in the following way. If $t \in T \setminus dom(rem)$[c], then $rem(t) = \varnothing$. If $t \in T \setminus dom(nofi)$, then $\pi_1(nofi(t)) = 1, \pi_2(nofi(t)) =$

---

[b] $A \nrightarrow B$ denotes a partial function. $\mathbf{P}(X)$ is the powerset of X, i.e., $Y \in \mathbf{P}(X)$ if and only if $Y \subseteq X$.
[c] $dom(rem)$ denotes the domain of rem.

1, $\pi_3(nofi(t)) = \infty$, $\pi_4(nofi(t)) = static$. This allows us to treat these partial functions as total functions in the remainder.

The mapping of tasks to lower-level EWF nets which are refining the task (as, e.g., for the tasks "Book hotel" and "Credit card payments" in Figure 4) is not part of the higher-level EWF net, but rather of the workflow specification which organizes the EWF-nets in a tree-like hierarchy. As mentioned above, we deviate here from the original YAWL specification by assigning sets of EWF nets to composite tasks. The selection which EWF nets from such a set is executed when the composite task is triggered is then performed at run-time, similar as the worklet service architecture extension to YAWL suggests.[46] Thus, although several EWF nets are assigned to a composite task, only one of the EWF nets is executed when the task is triggered. In this way, different implementations of a task can be assigned to the same generic task (e.g., the task "Book hotel" can have an implementing EWF net for bookings directly with the hotel by phone and another totally different implementation for bookings via a booking portal in the internet).

Definition 3 (Workflow specification) *A workflow specification S is a tuple $(Q^\diamond, Q, top, T^\diamond, map)$ such that:*

- *$Q^\diamond$ is a set of EWF-nets,*
- *$top \in Q^\diamond$ is the top level workflow,*
- *$Q \subseteq \mathbb{P}(Q^\diamond \setminus \{top\}), (\bigcup_{NS \in Q} NS) = Q^\diamond \setminus \{top\}, \forall_{NS_1, NS_2 \in Q}(NS_1 \cap NS_2 \neq \emptyset) \Rightarrow NS_1 = NS_2$, partitions $Q^\diamond$ into sets of EWF nets,*
- *$T^\diamond = \cup_{N \in Q^\diamond} T_N$ is the set of all tasks,*
- *$\forall_{N_1, N_2 \in Q^\diamond} N_1 \neq N_2 \Rightarrow (C_{N_1} \cup T_{N_1}) \cap (C_{N_2} \cup T_{N_2}) = \emptyset$, i.e., no name clashes,*
- *$map : T^\diamond \nrightarrow Q$ is an injective, surjective function which maps each composite task onto a set of EWF nets, and*
- *the relation $\{(N_1, N_2) \in Q^\diamond \times Q^\diamond \mid \exists_{t \in dom(map)}(t \in T_{N_1} \wedge N_2 \in map(t))\}$ is a tree.*

$Q^\diamond$ is a non-empty set of EWF-nets with a special EWF-net top. The tasks in the domain of map are the composite tasks which are mapped onto sets of EWF nets. This is done in such a way that each EWF net in $Q^\diamond$ can only be assigned onto one task, but each composite task is mapped onto a set of several EWF nets that is specified in Q, i.e. a tree-like structure with top as root node is formed. As top is always the root net, it will be part of any workflow execution. This holds not for the other EWF nets in $Q^\diamond$ (i.e. the child elements of top) if the EWF net or any of its parents has an alternative listed in Q.

Concluding this small introduction into YAWL please note that we assume throughout this paper that there are no name clashes, i.e., names of conditions differ from names of tasks and there is no overlap in names of conditions and tasks originating from different EWF-nets. If there are name clashes, tasks/conditions are simply renamed.

4.2. *Configurable EWF nets*

The general approach for making workflow modelling languages configurable from Section 3 can be divided into three main steps. Thus, we organize this section on the development of configurable EWF (C-EWF) nets in line with these steps. First, the configurable elements of EWF nets are identified and we provide a formal definition of configurations of EWF nets. Second, we develop a language for the specification of requirements on the models and their configurations as well as provide an approach to test if the configuration of a model satisfies the requirements. Finally, we will combine an EWF net with requirements and a default configuration to C-EWF nets.

Configurable elements of EWF nets and their configurations

To determine the configurable elements of EWF nets, all elements of EWF nets that represent some sort of action and the flow of cases through these elements need to be identified. Obviously, in EWF nets actions are represented by tasks and the surrounding arcs depict how tokens can flow into and out of tasks. However, the execution of the task is only enabled if the tokens in the pre-conditions of the task match its joining behavior. A task with an AND-join behavior as the task *Reserve* in Figure 4 can only be enabled and executed if tokens are waiting at all conditions preceding the task. That means, although the task has more than one incoming arc, it contains only a single port through which it can be enabled. On the other hand, a task with an XOR-join behavior can be enabled via every arc pointing at it, i.e. it has a dedicated port for each of these arcs. For example, the task *Send documents* in Figure 4 can be triggered either by a token in condition *a4* or by a token in condition *b4*. Thus it can be enabled through two different ports. A task with an OR-join behavior (like the task *select payment method* in Figure 4) is only enabled if there is at least one token on one of its input conditions and it exists no chance that further tokens can arrive. Thus, similar to the AND-join, it synchronizes all branches. We therefore assign only a single port to the OR-join. All ports through which a task in an EWF net can be enabled are called *input ports* in the following.

Definition 4 (Input ports) *Let $N = (C, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ be an EWF net. Then*

- $ports_{input}^{XOR}(N) = \{(t, \{c\}) | t \in T \wedge join(t) = XOR \wedge c \in \bullet t\}$ *are the input ports for all tasks with an XOR-join behavior,*
- $ports_{input}^{AND}(N) = \{(t, \bullet t) | t \in T \wedge join(t) = AND\}$ *are the input ports for all tasks with an AND-join behavior,*
- $ports_{input}^{OR}(N) = \{(t, \bullet t) | t \in T \wedge join(t) = OR\}$ *are the input ports for all tasks with an OR-join behavior,*
- $ports_{input}(N) = ports_{input}^{XOR}(N) \cup ports_{input}^{AND}(N) \cup ports_{input}^{OR}(N)$ *are all input ports of N, and*
- *for $t \in T$, $ports_{input}(t) = ports_{input}(N) \cap (\{t\} \times I\!P(C))$ are all input ports of the task t.*

Looking at the splitting behavior of tasks, we find similar semantics. If a task with an XOR-split behavior completes, it can choose between all outgoing arcs through which it will release the case. The task "Select payment method" from Figure 4 can, e.g., release a token either to condition *a3* or to condition *a4*. That means, each of the outgoing arcs from the task to the succeeding conditions has its own port. A task with an AND-split behavior like the *Start search* task releases tokens always via all outgoing arcs, i.e. it is only using a single port. Tasks with an OR-split behavior can release tokens to post conditions via any combination of outgoing arcs. This means for the task *Receive order* from Figure 4 that after completion it can put tokens either in *a1*, in *b1*, in *c1*, in *a1* and *b1*, in *a1* and *c1*, in *b1* and *c1*, or into all three of these conditions. Therefore a port exists for each of these combinations. Ports through which tokens are released to post-conditions are called *output ports* in the following.

Definition 5 (Output ports) *Let* $N = (C, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ *be an EWF net. Then*

- $ports^{XOR}_{output}(N) = \{(t, \{c\}) | t \in T \wedge split(t) = XOR \wedge c \in t\bullet\}$ *are the output ports for all tasks with an XOR-split behavior,*
- $ports^{AND}_{output}(N) = \{(t, t\bullet) | t \in T \wedge split(t) = AND\}$ *are the output ports for all tasks with an AND-split behavior,*
- $ports^{OR}_{output}(N) = \{(t, cs) | t \in T \wedge split(t) = OR \wedge cs \subseteq t \bullet \wedge cs \neq \emptyset\}$ *are the output ports for all tasks with an OR-split behavior,*
- $ports_{output}(N) = ports^{XOR}_{output}(N) \cup ports^{AND}_{output}(N) \cup ports^{OR}_{output}(N)$ *are all output ports of N, and*
- *for* $t \in T$, $ports_{output}(t) = ports_{output}(N) \cap (\{t\} \times I\!\!P(C))$ *are all output ports of the task t.*

As shown in Section 3 an input port determines if a task can be executed when it is enabled via this input port. It can be configured as either *enabled*, *blocked*, or *hidden*. The configuration of an output port determines which subsequent (post-) conditions can be marked with tokens after a task has been completed. It can be configured either as enabled or as blocked only.

Figure 6 provides two example configurations of the input and output ports for the main EWF net from Figure 4. The travel agency depicted in Figure 6a only sells reduction cards to clients buying a train ticket at the same time. Train tickets and hotel reservations can also be booked independently. For that reason the ports of the *Receive order* task which theoretically allow for booking the reduction card without booking a train ticket are blocked (indicated by the "Do not enter"-signs labelled b and b, c at the bottom-right corner of the task). The other five output ports, representing all possible booking combinations, are enabled (indicated by the arrows at the bottom-right corner of the task). Only if the customer pays by credit card, the documents can be sent to him. If the customer pays in cash, the documents cannot be sent. Then he has to collect them. This policy is enforced by blocking the input port b of the *Send documents* task (indicated by a "Do not enter"-sign at the bottom-left corner of the task).

The internet shop in Figure 6b uses the same process model, but a different configuration for its ports. It sells reduction cards also without train tickets, payments are only possible by credit card, and documents cannot be collected. In addition, the internet shop does not allow users to cancel their bookings. For that reason, all output ports of the *Receive order* task are enabled (whenever all ports are configured to the same value, we just show a single symbol), the input port of the *Cancel booking* task is blocked, output port b of the *Select payment method* task is blocked, and all input ports of the *Collect documents* task are blocked. In addition, the *Select payment method* task's input port is hidden because the internet shop only offers a single payment method. A selection simply does not need to be made (indicated by the "jumping" arrow at the bottom-left corner of the task).

In addition to the tokens consumed by a task via the the input ports, a task in YAWL can also consume all tokens from a cancellation region. For this reason, we decided to define a *cancellation port* per task in addition to the input ports.

Definition 6 (Cancellation ports) *Let* $N = (C, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ *be an*
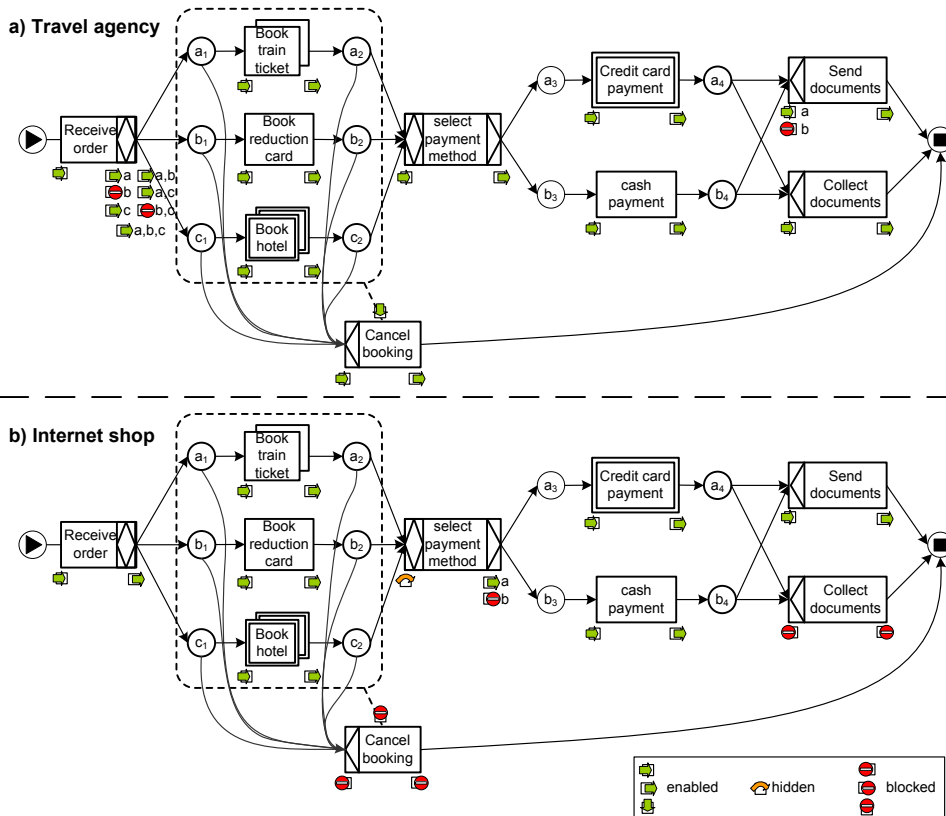

Fig. 6. The booking process from Figure 4, configured to the requirements of a travel agency (a) and an internet shop (b).

EWF net. Then $ports_{cancel}(N) = dom(rem)$ are all the cancellation ports of N.

The consumption of tokens from a cancellation region is similar to the consumption of tokens by an OR-join. During the task's execution always all available tokens are consumed from the cancellation region. However, tokens in the cancellation region cannot trigger a task on their own. The triggering still happens via the input ports and the enablement of a task does not even require the availability of tokens in the cancellation region. Thus, the cancellation port is only a refinement of the input port. If it is enabled, tokens are removed from the cancellation region; if it is blocked, they are not. The decision whether a task is executed or skipped remains determined by the input port configuration. Together they form the theoretic inflow port.

To depict the configuration of cancellation ports, we use the same pictures of an arrow for enabled cancellation ports and a "Do not enter"-sign for blocked cancellation ports (see the top of the task *Cancel booking* in Figure 6).

If a task allows for the start of multiple instances, it in fact combines several actions of the process in a single task. To implement this behavior, we could, e.g., introduce an internal OR-split (see Figure 7) that enables the instances of the task. Of course, the output ports of the OR-split can be configured as enabled or blocked. If some ports of this OR-split are blocked, this might decrease the total number of instances that can be started or increase the minimal number of instances that have to be started. For example, the task depicted in Figure 7 originally allowed a minimum of a single instance of the task and a maximum of three instances of the task to be started. By blocking all ports connected to a single subsequent condition, the minimal number of instances that can be started is increased to two. By blocking the port allowing the start of all instances, the maximal number of instances that can be started is also reduced to two. Therefore, we will talk about increasing the minimum number of instances to be started and decreasing the maximum number of instances to be started in the following, instead of referring to blocking of instances. If a task allows the dynamic creation of instances, the task has an additional internal task which creates new instances. By blocking its input port, we restrict a task with a dynamic creation of task instances to a static creation of task instances.

YAWL also allows to reduce the threshold value of the number of instances that have to be completed to consider the whole task as completed. This behavior – also known as N-out-of-M-join pattern[15] – can be implemented in a YAWL notation by forming several AND-joins instead of one OR-join for joining the multiple instances, each connected to the required minimal number of completed instances. On firing, such an AND-join could cancel all remaining instances. Figure 8 provides an example. Originally, the task required the start of three instances, but only two instances had to complete to consider the task as completed. By blocking the input ports of the AND-joins that require only two instances to be completed, we increase the threshold value of the task to three. Thus, also the increase of the threshold value of a multiple instance task is possible by means of configuration.

To formalize these four types of configuration opportunities we use four configuration functions, one for each type. The configuration functions assign the described configuration decisions to the ports of the EWF net. To allow for the configuration of selected parts of an

18 *F. Gottschalk, W.M.P. van der Aalst, M.H. Jansen-Vullers, and M. La Rosa*
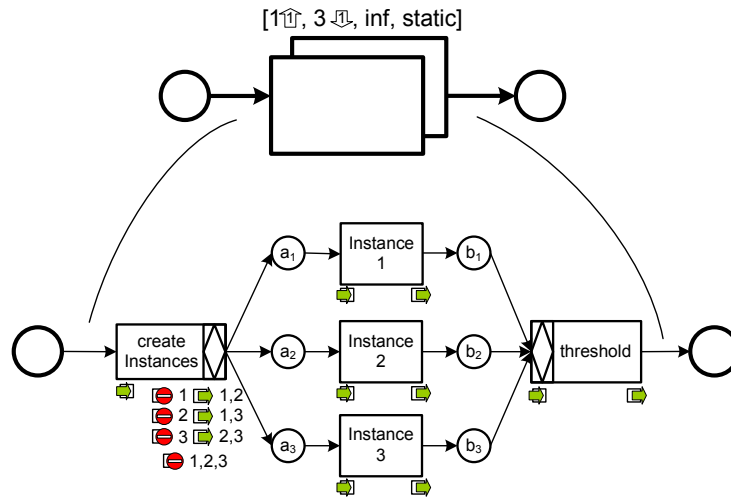

Fig. 7. A multiple instance task withe one to three instances implemented: the configuration restricts the behavior to the start of exactly two instances.
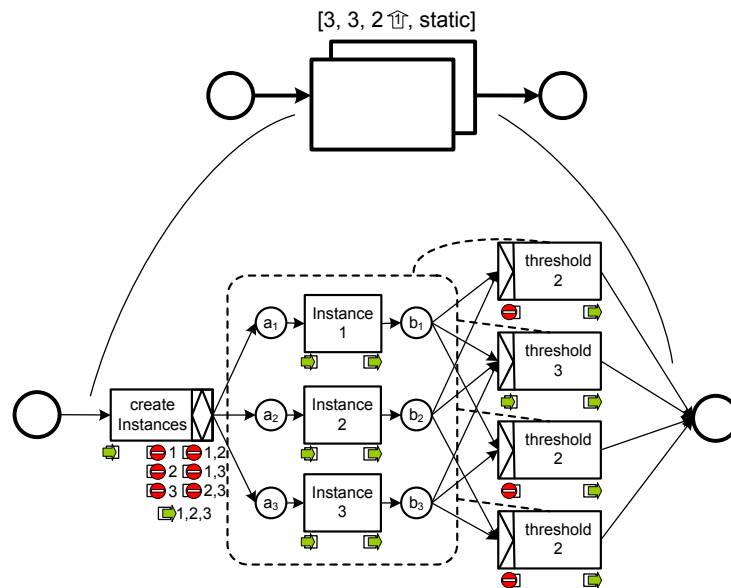

Fig. 8. Increasing the threshold value from two to three within a task with three instances to be started (example implementation).

EWF net, we define the configuration functions as partial functions. Then a configuration does not need to configure every port in the EWF net. However, to be able to transform the EWF net into a lawful, configured EWF net, the configuration functions must be defined on every port, i.e. they must be total functions. If all four configuration functions are total

functions, we call the configuration *complete*.

Definition 7 (Configuration) *Let $N = (C, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ be an EWF net, $ports_{input}(N)$ be the input ports of N, and $ports_{output}(N)$ be the output ports of N. Then $conf_N = (conf_{input}, conf_{output}, conf_{rem}, conf_{nofi})$ is a configuration of N with*

- *$conf_{input}$ defined as a partial function determining configurations for the input ports of tasks:*

$$conf_{input} : ports_{input}(N) \nrightarrow \{enabled, blocked, hidden\}$$

- *$conf_{output}$ defined as a partial function determining configurations for the output ports of tasks:*

$$conf_{output} : ports_{output}(N) \nrightarrow \{enabled, blocked\}$$

- *$conf_{rem}$ defined as a partial function determining configurations for the cancellation regions of tasks:*

$$conf_{rem} : ports_{cancel}(N) \nrightarrow \{enabled, blocked\}$$

- *$conf_{nofi}$ defined as a partial function determining configurations for the multiplicity of tasks:*

$$conf_{nofi} : dom(nofi) \nrightarrow \left(\mathbf{N}^0 \times \mathbf{N}^0 \times \mathbf{N}^{0,inf} \times \{restrict, keep\}\right)$$

such that

for all $t \in dom(conf_{nofi}) : (conf_{nofi}(t) = (min, max, thres, dyn)$ and

$$\pi_1(nofi(t)) + min \leq \pi_2(nofi(t)) - max).$$

The configuration $conf_N$ of N is complete iff

- *$dom(conf_{input}) = ports_{input}(N)$,*
- *$dom(conf_{output}) = ports_{output}(N)$,*
- *$dom(conf_{rem}) = ports_{cancel}(N)$, and*
- *$dom(conf_{nofi}) = dom(nofi)$.*

Similar as for EWF-net, we use $\pi_1(conf_{nofi}(t))$ to refer to the increase of the minimal number of instances that have to be created for task t, $\pi_2(conf_{nofi}(t))$ to refer to the decrease of the maximal number of instances that can be created, we use $\pi_3(conf_{nofi}(t))$ for the increase of the threshold value, and $\pi_4(conf_{nofi}(t))$ to indicate whether the creation of instances for the task t should be restricted to a static creation of instances only.

To form complete (or at least "more complete") configurations out of incomplete configurations, two configurations can be combined to a new configuration. This new configuration is formed by extending the domains of the configuration functions from the first configuration with the domains of the configuration functions from the second configuration. If, for example, a configuration has only configuration values for the ports of the task *Select payment method* from Figure 4 while a second configuration has configuration values for

the ports of the tasks *Receive order* and *Book train ticket*, the combined configuration has configuration values for all three tasks. In case both configurations contain a configuration value for a particular port, the value of the first configuration is used. In this way, combining incomplete configurations with complete configurations always creates complete configurations. The configuration values of the incomplete configuration then overwrite the ones of the complete configuration.

Definition 8 (Combining configurations) *Let* $N = (C, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ *be an EWF net. Let further on* $conf_{N,1} = (conf_{input,1}, conf_{output,1}, conf_{rem,1}, conf_{nofi,1})$ *and* $conf_{N,2} = (conf_{input,2}, conf_{output,2}, conf_{rem,2}, conf_{nofi,2})$ *be two (partial) configurations of N.*
Then $conf_{N,1}$ *and* $conf_{N,2}$ *can be combined and generate configuration* $conf_{N,3} = (conf_{input,3}, conf_{output,3}, conf_{rem,3}, conf_{nofi,3})$ *where*

- $dom(conf_{input,3}) = dom(conf_{input,1}) \cup dom(conf_{input,2})$ *and*
  - $\forall_{p \in dom(conf_{input,1})} conf_{input,3}(p) = conf_{input,1}(p)$,
  - $\forall_{p \in dom(conf_{input,2}) \setminus dom(conf_{input,1})} conf_{input,3}(p) = conf_{input,2}(p)$,
- $dom(conf_{output,3}) = dom(conf_{output,1}) \cup dom(conf_{output,2})$ *and*
  - $\forall_{p \in dom(conf_{output,1})} conf_{output,3}(p) = conf_{output,1}(p)$,
  - $\forall_{p \in dom(conf_{output,2}) \setminus dom(conf_{output,1})} conf_{output,3}(p) = conf_{output,2}(p)$,
- $dom(conf_{rem,3}) = dom(conf_{rem,1}) \cup dom(conf_{rem,2})$ *and*
  - $\forall_{t \in dom(conf_{rem,1})} conf_{rem,3}(t) = conf_{rem,1}(t)$,
  - $\forall_{t \in dom(conf_{rem,2}) \setminus dom(conf_{rem,1})} conf_{rem,3}(t) = conf_{rem,2}(t)$,
- $dom(conf_{nofi,3}) = dom(conf_{nofi,1}) \cup dom(conf_{nofi,2})$ *and*
  - $\forall_{t \in dom(conf_{nofi,1})} conf_{nofi,3}(t) = conf_{nofi,1}(t)$,
  - $\forall_{t \in dom(conf_{nofi,2}) \setminus dom(conf_{nofi,1})} conf_{nofi,3}(t) = conf_{nofi,2}(t)$,

Requirements specification and valid configurations

So far, the configuration opportunities offer a lot of freedom because each task can be configured in all the described facets. This is of course theory and in practice not all configurations are feasible. For example, in the workflow from Figure 6 it must be ensured that the *Receive order* task is always performed, i.e. its input port must always be configured as enabled. Without an order, it is impossible to book any travel. In the same way it must be ensured that if the customer has paid his travel, the documents are either sent to him or collected. That means for each payment method at least one input port of the tasks *Send documents* and *Collect documents* must be enabled. Similarly, we cannot block the input ports of the tasks *Credit card payment* and *Cash Payment* as long as the customer can choose the corresponding payment method in the task *Select payment method*. Otherwise, the case might deadlock in the conditions a_3 or b_3. Thus, in fact, quite restrictive dependencies exist among the configuration decisions for the individual ports.

To formulate such dependencies and restrictions of allowed configurations, we use logical expressions. The logical expressions combine requirements on the configuration of single elements of the EWF-net – the so-called atomic requirements – by means of common logical operators and quantifiers. The requirement that the input port of the task *Receive order* must be enabled is for example atomic and written as $(input, (\text{"Receive order"}, \{\mathbf{i}\}), enabled)$. The requirement that at least one of the input ports of the task *Send documents* or of the task *Collect documents* must be enabled after a credit card payment has been made (i.e. from condition a_4), is composed of two atomic requirements, connected by a logical operator as

$$(input, (\text{"Send documents"}, \{a_4\}), enabled) \vee$$

$$(input, (\text{"Collect documents"}, \{a_4\}), enabled).$$

Definition 9 provides a list of all atomic requirements that can be imposed on a configuration of an EWF net.

Definition 9 (Atomic configuration requirements)
Let $N = (C, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ be an EWF net, $ports_{input}(N)$ be the input ports of N, and $ports_{output}(N)$ be the output ports of N. Then

- $req_{input} = \{input\} \times ports_{input}(N) \times \{enabled, hidden, blocked\}$ *is the set of all requirements on the configurations of input ports,*
- $req_{output} = \{output\} \times ports_{output}(N) \times \{enabled, blocked\}$ *is the set of all requirements on the configuration of output ports,*
- $req_{rem} = \{rem\} \times T \times \{enabled, blocked\}$ *is the set of all requirements on the configuration of cancellation regions,*
- $req_{nofi} = \{nofi\} \times T \times (\mathbf{N}^0 \times \mathbf{N}^{0,inf} \times \mathbf{N}^{0,inf} \times \{restrictable, non\text{-}restrictable\})$ *is the set of all requirements on the configurations of the multiplicity of a task (maximal increase of the minimum, maximal decrease of the maximum, maximal increase of the threshold for continuation, and if restriction to static creation of instances is possible), and*
- $req_N = req_{input} \cup req_{output} \cup req_{rem} \cup req_{nofi}$ *is the set of all atomic requirements for N.*

To combine atomic requirements we allow the use of all the common logical operators ($\neg, \wedge, \vee,$ XOR, $\Rightarrow, \Leftrightarrow$ etc.) as well as the use of the quantifiers $\forall$ and $\exists$. With quantifiers, we enable the specification of requirements which have to hold for the configurations of sets of model elements. In this way it is possible to specify general requirements which are independent of a particular net.

We distinguish *general requirements*, which have to hold for every, or at least a certain group, of EWF-nets, from *specific requirements*, which only have to hold in a specific net. Specific requirements are typically content-driven. Therefore, the examples for requirements we provided above are specific requirements.

General requirements mainly ensure the construction of well-formed nets, i.e. they ensure that the configured EWF-net can be transformed into a syntactically valid EWF-net

which also conforms to any applicable modelling guidelines like, e.g., certain soundness criteria. Typically, general requirements are content-independent and make extensive use of quantifiers. For example, the requirement "each task of an EWF net which can be enabled, i.e. which has at least one enabled or hidden input port, must also have at least one enabled output port" would be a general requirement. It is needed to ensure the flow of tokens through the net. It can formally be specified as follows:

$$\forall t \in T :$$

$$(\exists_{p \in ports_{input}(t)}(input, p, enabled) \vee (input, p, hidden))$$

$$\Rightarrow (\exists_{p \in ports_{output}(t)}(output, p, enabled))$$

On the other hand, if all the input ports of a task are blocked, then there will never be any inflow to the task and consequently also no outflow. For that reason, we could formulate the configuration requirement that if all input ports of a task are blocked also all output ports must be blocked:

$$\forall t \in T :$$

$$(\forall_{p \in ports_{input}(t)}(input, p, blocked))$$

$$\Rightarrow (\forall_{p \in ports_{output}(t)}(output, p, blocked)))$$

Using general requirements, it is also possible to impose requirements on conditions although configuration is defined only on elements of tasks. For example, to ensure the flow of tokens through the net, every token that flows into a condition must also be able to flow out of it (unless it is in the final condition). Therefore at least one subsequent port must be enabled or hidden for such conditions:

$$\forall c \in (C \setminus \{\mathbf{o}\}) :$$

$$(\exists_{(t_1, cs_1) \in ports_{output}(N)} c \in cs_1 \wedge (output, (t_1, cs_1), enabled))$$

$$\Rightarrow (\exists_{(t_2, cs_2) \in ports_{input}(N)} c \in cs_2 \wedge ((input, (t_2, cs_2), enabled) \vee (input, (t_2, cs_2), hidden)))$$

A requirement is fulfilled if it evaluates to true. To evaluate a requirement, its atomic requirements have to be evaluated first. An atomic requirement is fulfilled if the specific port or task addressed by the requirement is configured accordingly. For example, the requirement $(input, (\text{"Receive order"}, \{\mathbf{i}\}), enabled)$ evaluates to true if the particular port between the condition $\mathbf{i}$ and the task *Receive order* is enabled, otherwise it evaluates to false. In the same way requirements for hidden or blocked input ports, and for enabled or blocked output or cancellation ports can be evaluated. A requirement on the number of instances configuration like $(nofi, \text{"Book train ticket"}, (min, max, thres, dyn))$ evaluates to true only if

- $\pi_1(conf_{nofi}(t)) \le min$,
- $\pi_2(conf_{nofi}(t)) \le max$,
- $\pi_3(conf_{nofi}(t)) \le thres$, and
- $dyn = \textbf{non-restrictable} \Rightarrow \pi_4(conf_{nofi}(t)) = keep$.

After all the atomic requirements within a composed requirement are evaluated to true or false, the composed requirement can be evaluated as in propositional logic[d]. Of course, the evaluation of an atomic requirement is only possible if a configuration is defined for the element of the EWF net that is addressed by the atomic requirement. For that reason, we assume complete configurations here.

We say that a complete configuration is *valid* for an EWF net, if the configuration fulfills all configuration requirements that are imposed on the EWF net, i.e. if all requirements can be evaluated to true.

Definition 10 (Valid configuration) *Let $N = (C, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$ be an EWF net, req_N be the set of all atomic requirements that can be imposed on N, and req be a boolean expression over req_N. A configuration $conf_N$ of N is* valid*, if req can be evaluated to true using the values of $conf_N$ to evaluate the atomic requirements contained in req.*

Components of C-EWF nets

To ensure complete configurations without requiring the user of a C-EWF net to configure every single element, a C-EWF net includes a default configuration. This default configuration must be *complete* and *valid* for the EWF net that should be configured. Then, each (incomplete) configuration can be combined with this complete default configuration to form a new complete configuration (as explained in Definition 8). The configuration decisions missing in the incomplete configuration are filled up with the default configuration for these elements. The so-created complete configuration can again be tested on its validity. If it is valid, we also consider the incomplete configuration as valid for the particular C-EWF net.

Summarizing, a C-EWF net consists of a syntactically correct EWF net serving as the basic process model, a set of configuration requirements ensuring syntactical and semantical correctness of the configuration, and the default configuration.

Definition 11 (C-EWF net) *A configurable extended workflow net (C-EWF net) is a tuple (N, R, D) where*

- N *is an EWF net,*
- R *is a set of configuration requirements on N, and*
- D *is the complete and valid default configuration of N.*

Note that the basic EWF net might contain semantically conflicting behavior. Thus, if no explicit configuration decisions are made, the default configuration also ensures that a semantically correct EWF net can be derived for the basic EWF net.

---

[d]Although we are allowing quantifiers, these are always expressed on a finite set of elements (e.g. "for all the conditions of the EWF net"). So this is referable to the conjunction of a set of propositional logic requirements, each of them expressed over an element (e.g. over a condition).

24   *F. Gottschalk, W.M.P. van der Aalst, M.H. Jansen-Vullers, and M. La Rosa*

4.3.   *Configurable workflow specifications*

A workflow specification organizes EWF nets hierarchically by mapping tasks of EWF nets onto other EWF nets (cf. Definition 3). Using C-EWF nets instead of EWF net, we will in this section briefly outline how a configurable workflow specification can be build-up on top of C-EWF nets. In this context we will use the expression *(C-)EWF nets* in statements that hold for both EWF and C-EWF nets.

In a workflow specification each composite task $t_{composite}$ of a (C-)EWF net is mapped onto a set of (C-)EWF nets $NS_{t_{composite}}$ via the map function (i.e. $NS_{t_{composite}} = map(t_{composite})$). Whenever $t_{composite}$ is triggered, one (C-)EWF net from the set $NS_{t_{composite}}$ is chosen as an implementation for $t_{composite}$ and initiated, i.e. there is a choice between the different nets of $NS_{t_{composite}}$. The task $t_{composite}$ only completes when the selected (C-)EWF net signalizes its completion. That means, the mapping between tasks and (C-)EWF nets determines the control flow between the nets. Hence, this mapping offers configuration opportunities in a configurable workflow specification in addition to the configuration opportunities of C-EWF nets.

Every (C-)EWF net has a unique input condition through which it can be triggered. Thus, the interface between the superior task $t_{composite}$ and the input condition of an implementing (C-)EWF net represents the unique inflow port of the action implemented in the mapped (C-)EWF net. If this inflow port is configured as *blocked*, the particular (C-)EWF cannot be triggered at runtime, i.e. it cannot be chosen as an implementation for $t_{composite}$. Instead, another (C-)EWF net from $NS_{t_{composite}}$ has to be selected whose inflow port is configured either as *enabled* or as *hidden*. Those nets can can be selected and triggered as normally in YAWL, i.e. as described above. Nets with an enabled inflow port are also executed in the same way as ordinary (C-)EWF nets. The action within (C-)EWF nets with a hidden inflow port must however be skipped completely. For this reason such a (C-)EWF net should be replaced with the "dummy" EWF net shown in Figure 9 where the τ task corresponds to the skipped action of the original net.

The completion of a (C-)EWF net is signalized via its unique output condition. The interface from the output condition back to the superior task $t_{composite}$ therefore represents the unique outflow port of a (C-)EWF net. As each action needs at least one enabled outflow port to be able to forward the control to subsequent actions, this outflow port must always be enabled, i.e. it cannot be configured.

Thus, on the level of the workflow specification, the only configurable port of an (C-)EWF net is its inflow port. We can therefore depict this configuration on the link between the composite task and the particular (C-)EWF net as, e.g., in Figure 10. The figure depicts an example configuration for the workflow specification of Figure 6.
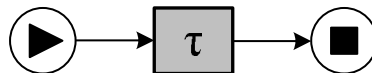

Fig. 9. The dummy net replacing a hidden net, i.e the τ task corresponds to the skipped action of the original net.
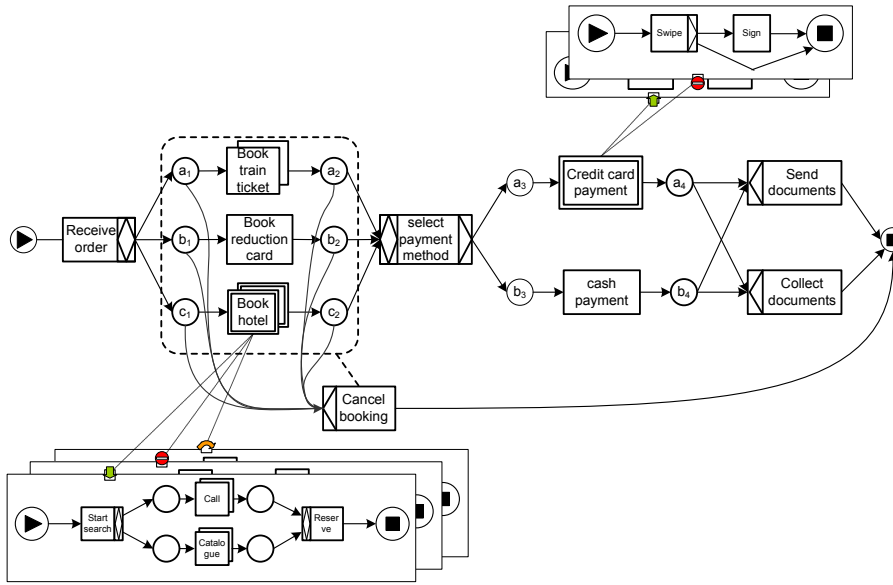
Fig. 10. The configured workflow specification

Again, not all combinations of configurations among the different (C-)EWF nets are feasible. For example, it is not possible to block all the (C-)EWF nets implementing a task because every composite task must have at least one usable, i.e. either enabled or hidden, implementation. For that reason, atomic requirements must also be imposable on the configuration of mappings between a composite task and an (C-)EWF net, e.g. as $(map, (task, net), enabled|blocked|hidden)$. Such requirements can be combined and evaluated as the requirements on configurations of C-EWF nets.

This also enables us to combine requirements on the higher-level configurable workflow specification with requirements on specific C-EWF nets within the configurable workflow specification. For example, the possibility to book reduction cards in the workflow of Figure 10 might require the possibility to use a certain implementation of the payment tasks that includes an address, age, or student status verification. Some implementations of a task may also depend on the data output of a preceding task. Thus, these implementations must be blocked if the particular task has been hidden or blocked.

Altogether, a configurable workflow specification provides two levels for configuring a workflow. Distinct approaches can be handled by different (C-)EWF nets mapped onto a single task. Different variants of the same approach should be handled within a single C-EWF net by using its configuration opportunities.

5. From C-YAWL to YAWL

To demonstrate the applicability of configurable workflow models, we implemented a transformation from C-YAWL to YAWL such that we can derive YAWL models which are

executable in the YAWL workflow engine. As a task can be easily mapped onto a selected, implementing EWF net, we focus in this section just on the transformation from C-EWF nets to EWF nets and do not explain the rather trivial selection mechanism described in Section 4.3.

To be able to perform such a transformation, we first have to define a file format for maintaining the configurations of EWF nets. As the YAWL engine itself uses a well-defined XML representation for storing YAWL workflow specifications, we decided to extend this XML schema with configuration opportunities. Afterwards we will provide a transformation algorithm that "translates" the configuration into a removal of elements from the basic model. All of this has been realized in the context of the YAWL environment[13,49].

5.1. *The C-YAWL XML Schema*

A workflow specification in a YAWL engine file consists of several decompositions, which are its EWF nets or the links to other custom web-services that can be triggered by a YAWL specification. A decomposition of an EWF net contains the data variables used by the net and a list of process control elements. These are all the conditions and tasks of the EWF net, starting with the input condition and ending with the output condition. Within each of these elements, flows define the links to subsequent elements, thus constructing the net. For tasks the joining and splitting behavior, the cancellation sets, and its multiplicity can be specified. Also a decomposition that is used to implement the task (as e.g. another EWF net in case of a composite task) and the data flow in and out of the task can be listed.

As the configuration of a YAWL model is purely defined on the level of tasks, we added configuration on this level. Any configuration can consist of the four configuration elements *join*, *nofi*, *rem*, and *split*. Each of these configuration elements allows for the specification of the particular configuration value, e.g. for the join a configuration value of "enabled", "blocked", or "hidden" can be specified. The value given for the join and split configuration is applicable for all input or output ports of the particular task except for those for which dedicated *port* subelements specify different configurations. These port elements have a configuration value as their parent element plus a set of source or target conditions and tasks identifying the addressed port (see Figure 11).

Instead of or in addition to a configuration of the task, a default configuration can be specified for each task. The default configuration is specified as a normal configuration, but is used whenever no explicit configuration value is given for the particular element. Whenever neither a configuration nor a default configuration is given for a configurable element, we assume that the task keeps its original behavior, i.e. enabled input, output, and cancellation ports, as well as no increase of the minimum number of instances, no decrease of the maximum number of instances, no increase of the threshold value, and keeping of the creation mode. Thus, a configurable YAWL model for which a default configuration is specified can be transformed into a YAWL model without any of the configuration values explicitly specified. As long as all requirements on the configuration are satisfied by configuring all ports as enabled and not deviating from the original multiple instance configuration, the transformation can even be performed without any configuration or default
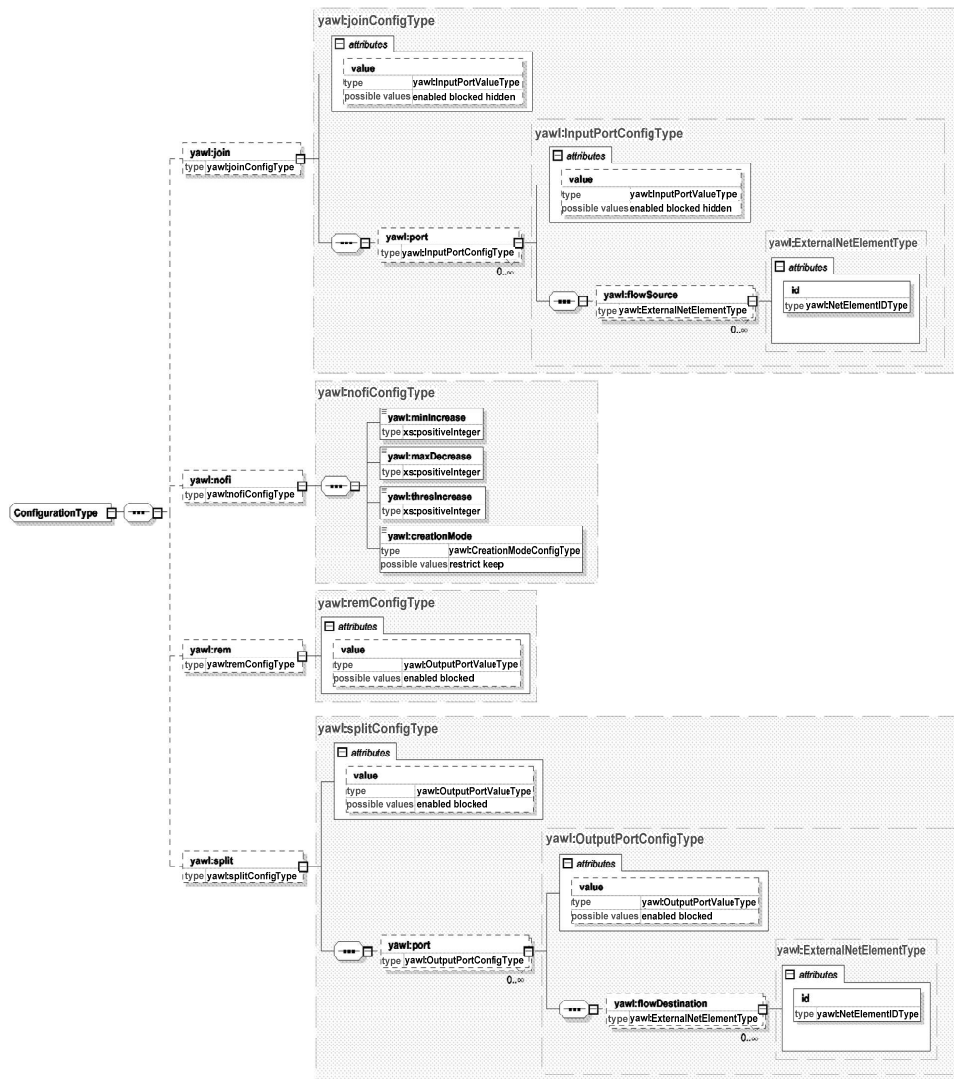
Fig. 11. The XML specification of the configuration (depicted using XMLSpy).

configuration value specified. The resulting YAWL model would then match exactly the basic model.

5.2. *The Transformation Algorithm*

The transformation from C-EWF nets to EWF nets is performed in two steps. First, we will remove the elements directly affected by the configuration decisions. Second, we will perform a clean up by removing elements which became obsolete in the first step. The latter step ensures that the created EWF net conforms to Definition 1 which requires that every

element is on a path between **i** and **o**. In this way, we can neglect this requirement in the first step.

As input to the transformation, let $CEWF = (N, R, D)$ be a C-EWF net with $N = (C, \mathbf{i}, \mathbf{o}, T, F, split, join, rem, nofi)$. In addition, let $conf_N$ be a valid configuration of $CEWF$ that is applied to $CEWF$ to create a configured EWF net. Let then $(conf_{input}, conf_{output}, conf_{rem}, conf_{nofi})$ be the combined configuration of $conf_N$ with D, i.e. $(conf_{input}, conf_{output}, conf_{rem}, conf_{nofi})$ is complete.

As conditions are not configurable, we initially keep all conditions from the C-EWF net also in the configured EWF net and remove the superfluous conditions later during the cleanup. The input and output conditions **i** and **o** are the same in the configured net as in the configurable net.

For transforming the tasks' behavior, we will start with the $conf_{rem}$ and $conf_{nofi}$ configurations as these can be applied to the tasks straightforwardly.

The configuration of the cancellation region $conf_{rem}$ restricts the set of elements returned by the rem function. Whenever the cancellation region is blocked, the function returns an empty list.

- $\forall_{t \in dom(rem) \cap \{t \in T | conf_{rem}(t) = blocked\}} rem^C(t) = \emptyset$
- $\forall_{t \in dom(rem) \setminus \{t \in T | conf_{rem}(t) = blocked\}} rem^C(t) = rem(t)$

The $nofi$ function, assigning the number of instances that can be started to each task, must be adapted according to the configuration $conf_{nofi}$. The configured increase of the minimal number of instances to be started is added to the predefined minimal number of instances, the configured decrease of the maximal number of instances to be started is subtracted from the predefined value, and the configured increase of the threshold value is added to the predefined threshold value. If the predefined task enables the dynamic creation of task instances and the task is configured to keep the current definition, the creation of task instances remains dynamic, otherwise it is set to static.

- $\forall_{t \in dom(nofi)} \pi_1(nofi^C(t)) = \pi_1(nofi(t)) + \pi_1(conf_{nofi}(t))$
- $\forall_{t \in dom(nofi)} \pi_2(nofi^C(t)) = \pi_2(nofi(t)) - \pi_2(conf_{nofi}(t))$
- $\forall_{t \in dom(nofi)} \pi_3(nofi^C(t)) = \pi_3(nofi(t)) + \pi_3(conf_{nofi}(t))$
- $T_{dyn} = \{t \in dom(nofi) | \pi_4(conf_{nofi}(t)) = keep \wedge \pi_4(nofi(t)) = dynamic\}$
- $\forall_{t \in T_{dyn}} \pi_4(nofi^C(t)) = dynamic$
- $\forall_{t \in dom(nofi) \setminus T_{dyn}} \pi_4(nofi^C(t)) = static$

The configuration of output ports $conf_{output}$ influences the flows subsequent to a task. If one of the output ports referring to a particular flow is enabled, the flow is part of the configured EWF net. Otherwise it is not. As tasks with an AND-split behavior have just a single port, all flows subsequent to such tasks must be removed if the port is configured as blocked. In case of an XOR-split, each flow is addressed by exactly one port. Thus, a flow must be removed if the corresponding port is blocked. The output ports of a task with an OR-split semantics can be configured in different ways even if the different ports refer to
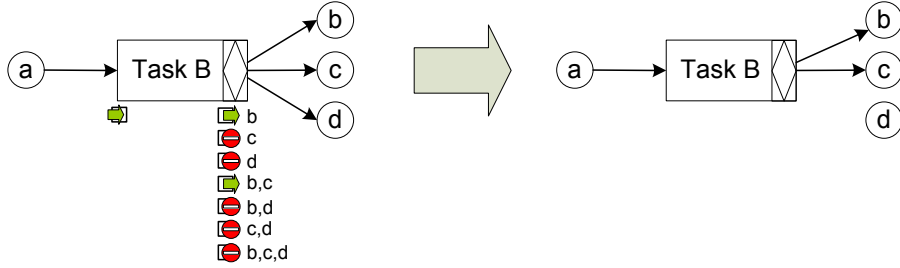
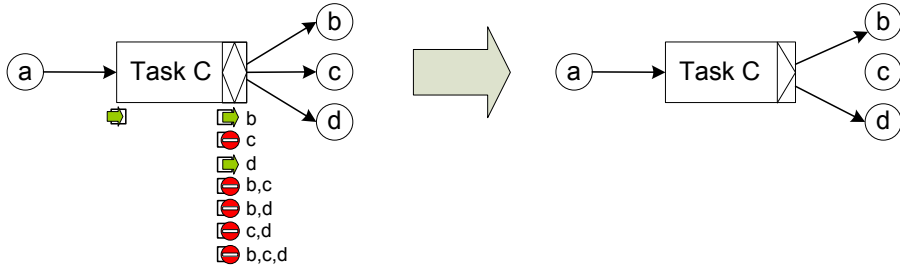Fig. 12. Transforming the output port configuration into a YAWL model

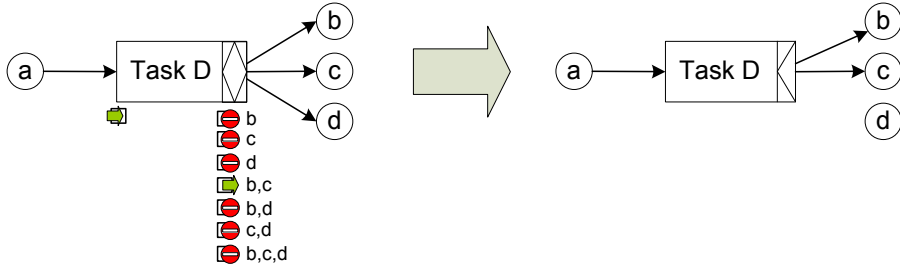Fig. 13. Transformation of an OR-split into an XOR-split.

Fig. 14. Transformation of an OR-split into an AND-split.

the same flow. Then, as said above, the flow must be kept as part of the configured EWF net if any output port referring to the flow is enabled. For example, in Figure 12 only the two ports connecting *Task B* either with condition b or with conditions b and c are enabled. All other output ports are blocked. Therefore, the flow from *Task B* to d can be removed, but the flows to the conditions b and c cannot be removed. The blocking of ports referring to these flows that must be kept in the net, e.g. the blocking of the output port c, is realized by adapting the flows' predicates. Based on process data, predicates determine at run-time if a flow is triggered or not.

- $F_{output}^{C} = \{(t,c) \in F | t \in T \land c \in C \land \exists_{(t,cs) \in ports_{output}(N)} c \in cs \land conf_{output}((t,cs)) = enabled\}$

The splitting behavior of a task basically corresponds to its behavior in the EWF net. Only in two special cases of configuration of a task with an OR-split behavior the splitting changes. If all the output ports of such a task which refer to more than a single flow are blocked, the splitting behavior is changed into an XOR-split behavior (see Figure 13). If all the output ports of a task are blocked except a single port, the splitting behavior is transformed into an AND-split (see, e.g., Figure 14). In all other cases the splitting behavior remains the same.

- $T_{XOR}^C = \{t \in T | \forall_{(t,cs) \in ports_{output}(t)}(|cs| > 1 \Rightarrow conf_{output}((t, cs)) = blocked)\}$
- $T_{AND}^C = \{t \in T | \forall_{(t,cs) \in ports_{output}(t)}(|cs| < |\{(t, c) \in F_{output}^C\}| \Rightarrow conf_{output}((t, cs)) = blocked)\}$
- $\forall_{t \in T_{XOR}^C} split^C(t) = XOR$
- $\forall_{t \in T_{AND}^C \setminus T_{XOR}^C} split^C(t) = AND$
- $\forall_{t \in T \setminus (T_{XOR}^C \cup T_{AND}^C)} split^C(t) = split(t)$

Finally, the configuration of the join behavior $conf_{input}$ has to be applied to the EWF net. If a task has an AND-join or an OR-join behavior, it just has a single input port. If this port is blocked, the task can never be enabled. Therefore, all inflows into the the task are not part of the configured EWF net. If the input port of a task t is hidden, this means that the execution behavior of the task must be skipped. For that reason, task t is replaced with a silent task τ_t. The silent task does not include any "action" as, e.g., any execution of work from the original task but has still exactly the same joining, splitting, and cancellation behavior as the original task.

A task with an XOR-join behavior can have different configurations for different input ports. In this case it might be required that a net includes both a silent version and an active version of a task. For example in Figure 15 the input ports from the conditions a and b are hidden, the input port from condition c is enabled and the input port from condition d is blocked. Then the conditions a and b should trigger the silent task τ_{TaskA}, whereas condition c should trigger the active task. Therefore, we split up the set of tasks for the configured net into a set of enabled and a set of hidden tasks, i.e. $T^C = T_{enabled}^C \cup T_{hidden}^C$. The silent task must be introduced whenever a task has at least one hidden input port. The (normal) enabled task remains in the net whenever there is at least one enabled input port.

- $T_{enabled}^C = \{t \in T | \exists_{(t,cs) \in ports_{input}(t)} conf_{input}((t, cs)) = enabled\}$
- $T_{hidden}^C = \{\tau_t | t \in T \wedge \exists_{(t,cs) \in ports_{input}(t)} conf_{input}((t, cs)) = hidden\}$
- $\forall_{\tau_t \in T_{hidden}^C} :$
  - $join^C(\tau_t) = join(t) \wedge$
  - $split^C(\tau_t) = split^C(t) \wedge$
  - $rem^C(\tau_t) = rem^C(t) \wedge$
  - $nofi^C(\tau_t) = nofi^C(t)$

To connect the tasks with conditions, all flows connected to an enabled port remain in the net. All flows connected to a hidden port are reconnected to the hidden task. Therefore,
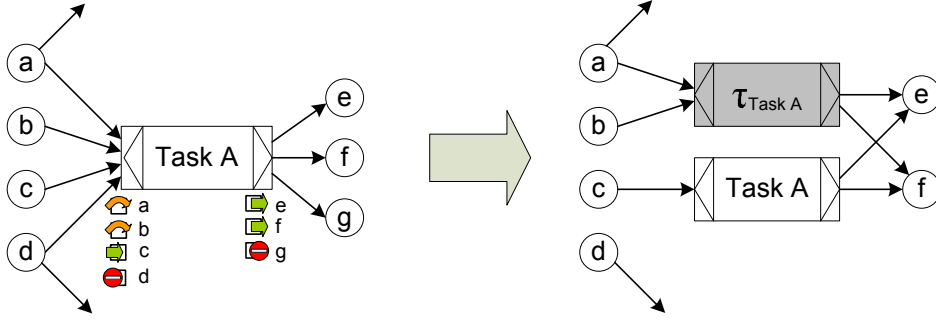
Fig. 15. Transforming the input port configuration into a YAWL model.

conditions a and b in the example of Figure 15 are connected to the silent task, whereas c remains connected to *Task A*. All flows connected to a blocked input port are not part of the configured net. For that reason the flow connecting d with *Task A* is not part of the configured net.

- $F^C_{input} = \{(c,t) \in F | c \in C \wedge t \in T \wedge$
  $\exists_{(t,cs) \in ports_{input}(N)} c \in cs \wedge conf_{input}((t,cs)) = enabled\}$
  $\cup \{(c,\tau_t) | c \in C \wedge t \in T \wedge (c,t) \in F \wedge$
  $\exists_{(t,cs) \in ports_{input}(N)} c \in cs \wedge conf_{input}((t,cs)) = hidden\}$

The joining behavior of all enabled tasks in the configured net is the same as the joining behavior in the configurable net.

- $\forall_{t \in T^C_{enabled}} join^C(t) = join(t)$

Altogether, we transformed the C-EWF net into the net $N^C = (C, \mathbf{i}, \mathbf{o}, T^C_{enabled} \cup T^C_{hidden}, F^C_{input} \cup F^C_{output}, split^C, join^C, rem^C, nofi^C)$. However, as mentioned in the beginning, the resulting net does not necessarily conform to the requirements of an EWF net in which every node in the graph must be on a directed path from $\mathbf{i}$ to $\mathbf{o}$. Due to the removal of flows, some conditions and tasks might not be reachable anymore from $\mathbf{i}$. To create an EWF net from N^C, it is therefore necessary to remove all nodes which are not on a path from $\mathbf{i}$ to $\mathbf{o}$. Removing of conditions preceding AND-joins or succeeding AND-splits leads however to changes in the semantics of the net. Thus, in case such conditions should be removed, not only the conditions must be removed, but also the tasks with the AND-join/ -split, although these tasks might theoretically be on paths from $\mathbf{i}$ to $\mathbf{o}$. If there is no such path at all, the configuration is not transformable into a well defined EWF net and should be forbidden in the requirements on the C-EWF net.

This cleanup step can be performed as a depth-first search, starting with the input condition, looking for all paths to the output condition. If such a path is found, all elements on this path are marked for being kept in the process. In addition, all visited elements are marked, such that elements do not need to be visited multiple times. All tasks and conditions not on such a path are afterwards removed. If a condition is removed which

is preceded or succeeded by an AND-join or -split, the corresponding task is removed as well even though it might be on a path between input and output condition. In this way we prevent changing the semantics of the task. But this might make additional elements "loosing" their path from the input to the output condition. For that reason, we repeat the whole cleanup process until in one iteration no such tasks are removed. Considering usual sizes of workflow models this implementation is sufficient.

We implemented this transformation in the context of the YAWL workflow engine. Using the YAWL editor the integrated model of the different process variants can be defined as depicted in Figure 4 and exported into a YAWL engine file. Such an engine file can be loaded into the YAWL engine to execute the whole workflow. To restrict the workflow to the desired variants, configurations can be added to the YAWL engine file as depicted in Section 5.1. Without any manual modelling effort, the algorithm depicted in this section then generates a new YAWL engine file according to the configuration. As the original file, the generated file can directly be used in the workflow engine to execute the desired workflow variant. It can also be imported back into the YAWL editor to inspect or further adapt the resulting workflow.

The EWF net derived in this way from the example configuration for a travel agency as depicted in Figure 6a is shown in Figure 16; the EWF net derived from the configuration for the internet shop depicted in Figure 6b is shown in Figure 18. To demonstrate the need to remove dead model parts, Figure 17 depicts the net resulting from the internet shop's configuration with the elimination of dead model parts disabled.

This transformation therefore enables a user who wants to implement a variant of a standard workflow (which is already designed, e.g., by consultants) to avoid any expensive and complex workflow modelling. By only adding the individual configuration to the workflow, running the transformation, and loading the resulting model into the YAWL engine, the individual adapted workflow can be executed as the screenshot in Figure 19 shows. It depicts the worklist of the internet shop's travel booking workflow with several bookings in progress. The workflow definition used for this example is the engine file which resulted from the configuration shown in Figure 3.
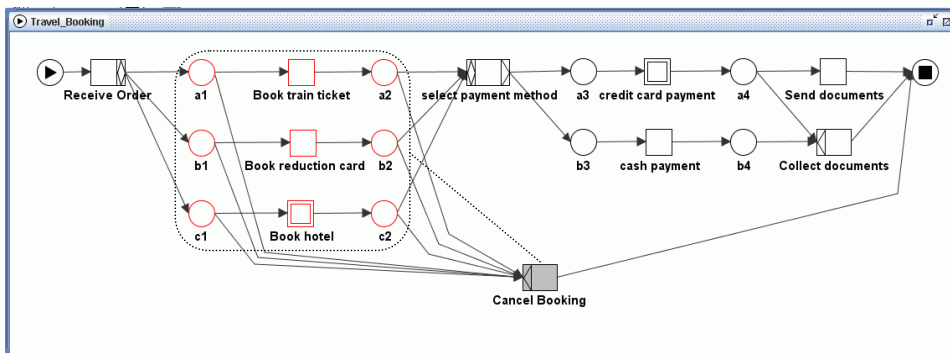

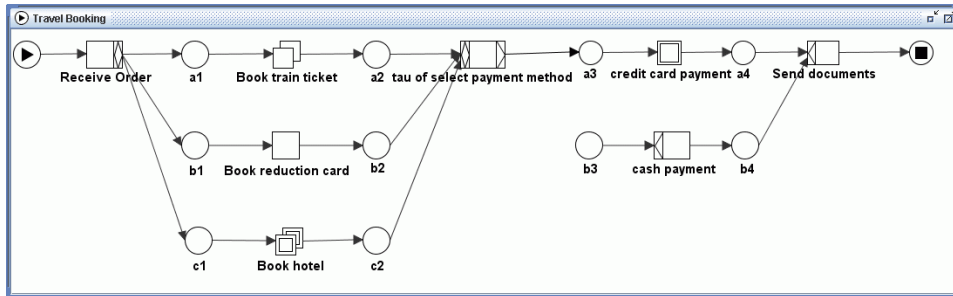Fig. 16. The YAWL net derived from the configuration of the travel agency.

Fig. 17. The net derived from the internet shop's configuration, but without the removal of any "dead" parts.
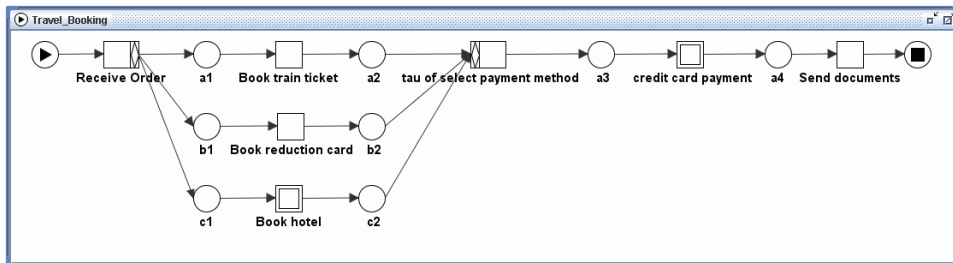

Fig. 18. The YAWL net derived from the configuration of the internet shop having all "dead" model parts removed.

Both C-YAWL and the C-YAWL to YAWL transformation will become part of future YAWL releases and thus be available to all YAWL users.[49]

6. Configuration of other workflow languages

YAWL is a workflow language with an academic background which was developed to demonstrate that a workflow engine can easily support a multitude of workflow patterns without having a very complicated language. To implicitly cover the configurability of all these patterns, we so far demonstrated how a workflow language can be made configurable based on YAWL. To show the universal applicability of the approach to many languages we will now also briefly highlight, how it can be applied to a commercial workflow engine, namely the workflow engine of SAP R/3 (SAP WebFlow), and to a standard notation, namely BPEL.

SAP WebFlow

SAP WebFlow is a workflow engine delivered with every SAP R/3 enterprise system in-stallation since the R/3 Release 3.0.[12] Its workflow language (see Figure 20) is mainly based on so-called *steps* and *events*. Steps can represent both routing constructs and system functionalities. *Activities* such as the *Book train ticket* step or the *Book reduction card* step execute a task defined within the SAP system. Steps representing routing constructs often
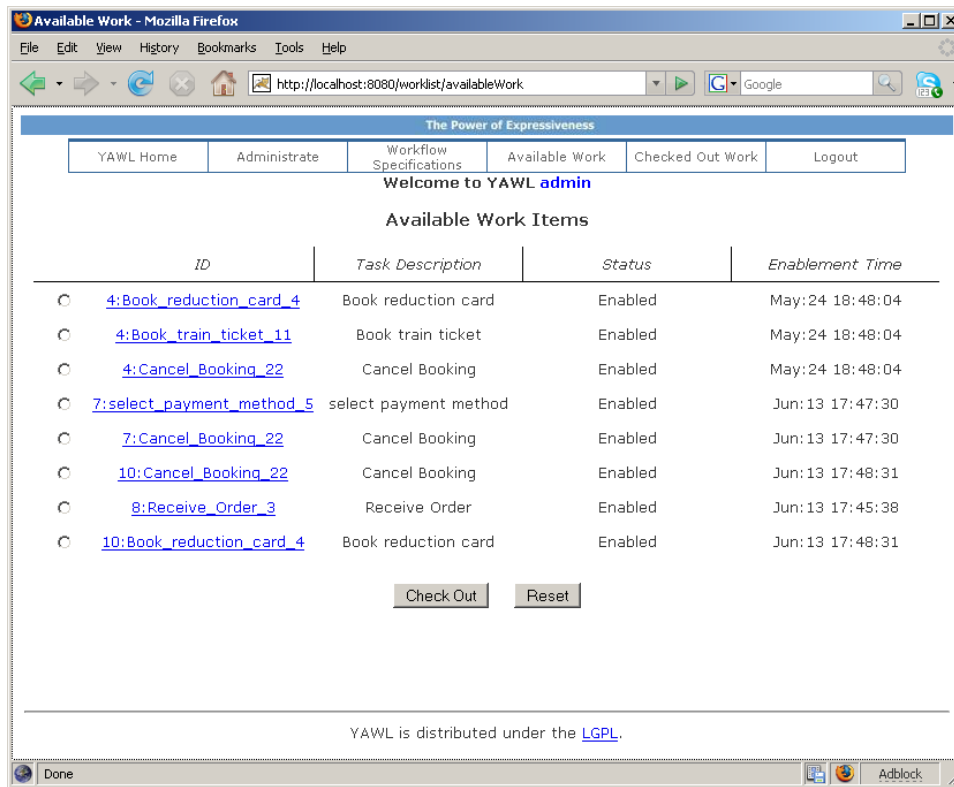
Fig. 19. The worklist of the internet shop's travel booking workflow with several bookings in progress

branch the control flow. Examples for such branching steps are *conditions* like the *Select payment method* step in Figure 20 which represent an XOR choice between two paths, or *forks* like the *Receive order* step which are dependent on a condition of the joining fork and can be used to represent both AND splits/joins or deferred choice constructs. Any such branching of the control flow is matched by a dedicated corresponding join. Similar to composite tasks in YAWL, *ad hoc anchors* can be replaced with another workflow which is selected from a set of possible replacements at runtime.

In this way the workflows of SAP WebFlow are organized in a block structure. Each step represents a dedicated block. Since every split corresponds to a join, the subprocess in between such a split and join also forms a block. All steps included in such a block are then sub-blocks of the routing construct's block. To demonstrate the block structure in Figure 20 the three main blocks are highlighted in grey while their sub-blocks in the various branches have a white background.

To configure such workflow activities in SAP we can use the concepts identified in Section 3. Due to the block structure we can even go a step further and say that each block can be seen as an action. Every block contains just one unique input path and one
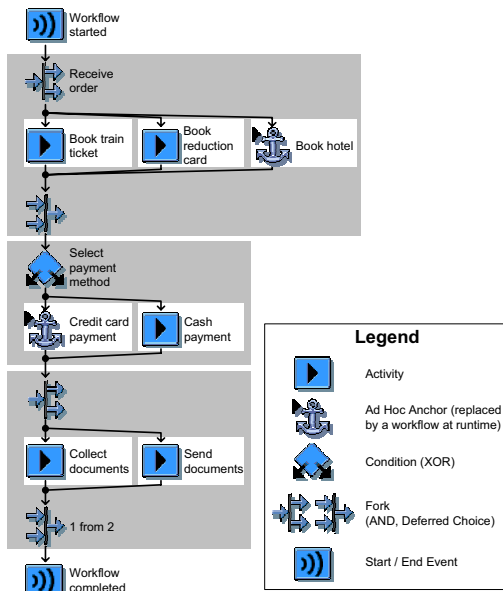
Fig. 20. The travel booking workflow in a SAP WebFlow notation with the block structure highlighted
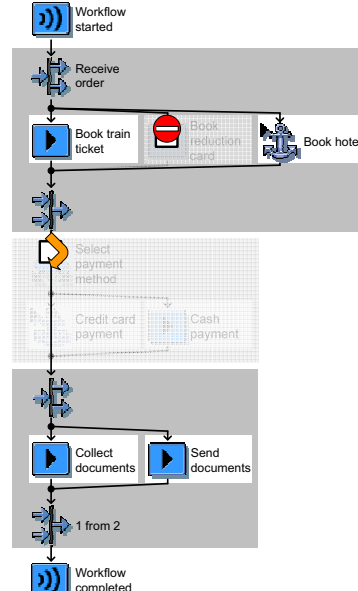

Fig. 21. A configuration of the workflow

unique output path. Thus it contains a unique *input port* and a unique *output port*. The largest block is the complete workflow itself. It is the only block which can be triggered in multiple ways as it can be triggered not only manually but also by (various) external events which are linked to the workflow block. Such events can also be linked to a block to terminate it. Thus, each of these links connecting an event to a workflow block can also be seen as a port. As they have some different characteristics from a block's in- and output ports, we call them *event ports*.

For input ports the concepts of enabling, blocking, and hiding can be applied in a straightforward manner. If the input port of a block is enabled, cases can enter normally and be executed in the block. If the input port is hidden like the one of the *Select payment method* step in Figure 21, a case entering the block is directly forwarded to the unique output port of the block, quasi bypassing all the content of the block. If the input port is blocked as the port of the *Book reduction card* step in Figure 21, the case cannot enter the block at all and needs to continue via one of the other alternative branches.

Each case entering a block must be able to leave the block via its unique output port. Thus, this port can only be blocked if the block's input port is blocked. However, if the input port is blocked no tokens can arrive at the output port, i.e. the configuration has no influence on the execution of the process. We can therefore consider the output port configuration as practically irrelevant in SAP WebFlow.

An event triggers a new instance of a workflow if the corresponding event port is enabled. SAP WebFlow already supports deactivating such a link, quasi corresponding to the blocking of the particular event port. Although a triggering event port is an inflow port, the

hiding of such a port is at least questionable as it would result in the skipping of the whole workflow without performing any steps.

Terminating event ports are output ports. Even though terminating events are externally triggered, they enforce the removal of the case from the particular block. Thus, the functionality of SAP to activate or deactivate such linkages already provides exactly the required functionality to configure terminating event ports.

As in a configurable YAWL, also a configurable SAP WebFlow may not allow for a free configuration of all ports. For example, the three grey blocks are essential to complete the workflow of Figure 20 and can thus not be blocked (but might be hidden, e.g. because someone billing tickets directly to a responsible organization might be able to skip the billing process as in Figure 21). As all these configuration restrictions are on a port level, they can be enforced with boolean expressions in the same way as we have shown for YAWL in Section 4.

The conversion of a workflow according to a configuration which conforms to the configuration requirements is straightforward. The input arc of a hidden block is directly connected to its output arc while blocked blocks are simply removed.[17]

BPEL

The Web Services Business Process Execution Language (WS-BPEL, or just BPEL for short) is a standard developed for the composition and orchestration of web services. BPEL uses an XML-based representation to define workflows and the language can be seen as a mixture of graph-based and programming-like constructs. Similar to SAP WebFlow, activities in BPEL are also block structured. Within this block structure BPEL distinguishes between six structured activities organizing the control-flow of its sub-blocks and three primitive activities that perform the required actions. Examples for structured activities are the *sequence* which enforces the sequential execution of its sub-blocks (see Figure 22), the *flow* which allows for a parallel execution of the sub-blocks, or the *switch* which can be used to define an XOR choice among the sub-blocks. The primitive *invoke* activities call a linked operation, e.g. of a web service or another workflow, and wait for its response. For example, in Figure 22 the first invoke activity calls the operation *requestTravel* of the booking eingine. The other two primitive types of *receive* and *reply* activities wait for such calls and provide the results.

Within the block structure BPEL activities have the same unique input ports and output ports as the steps of SAP WebFlow. Thus, we can enable, block, or hide the input ports of activity blocks in the same way as we have suggested for SAP WebFlow and we do not need to configure the output ports.

However, in addition to the structured activities such as a sequence, switch, etc. BPEL allows for expressing control flow relations between activities also through *control links*. Control links establish a control flow from one to another activity possibly breaking BPEL's block structure. For this reason not only the unique output port is triggered when an activity completes, but also *outgoing control links* can be activated. In Figure 22 three such control links are, e.g., specified for the first invoke activity. An activity for which a corresponding

```
<process ...>
  <sequence ...>
    <flow ...>
      <links>
        <linkName="trainTicket"/><linkName="reductionCard"/><linkName="hotel"/>
      </links>
      <invoke partner="BookingEngine" operation="requestTravel"
          inputVariable=... outputVariable="travelNeeds" ...>
        <source linkName="trainTicket"
           transitionCondition="bpws:getVariableData(travelNeeds, trainReq)=true"/>
        <source linkName="reductionCard"
           transitionCondition="bpws:getVariableData(travelNeeds, cardReq)=true"/>
        <source linkName="hotel"
           transitionCondition="bpws:getVariableData(travelNeeds, hotelReq)=true"/>
      </invoke>
      <invoke partner="TicketProvider" operation="getTicket" ...>
        <target linkName="trainTicket"/>
      </invoke>
      <invoke partner="CardProvider" operation="orderCard" ...>
        <target linkName="reductionCard"/>
      </invoke>
      <invoke partner="HotelReservationSystem" operation="reserve" ...>
        <target linkName="hotel"/>
      </invoke>
    </flow>
    <flow ...>
      <links>
        <linkName="creditCardPayment"/><linkName="cashPayment"/>
      </links>
      <invoke  partner="PaymentEngine" operation="getPaymentDetails"
          inputVariable=... outputVariable="paymentDetails" ...>
        <source linkName="creditCardPayment"
           transitionCondition="bpws:getVariableData(paymentDetails, card)=true"/>
        <source linkName="cashPayment"
           transitionCondition="bpws:getVariableData(paymentDetails, cash)=true"/>
      </invoke>
      ...
    </flow>
    ...
  </sequence>
</process>
```

Fig. 22. The travel booking workflow specified in BPEL

incoming control link is specified can then only be executed if it is triggered through the structure of the workflow, and if all the incoming links have been activated. The invoke activity in Figure 22 that gets the ticket is thus only activated if on the one hand the enclosing flow activity is activated and on the other hand the link *trainTicket* is activated. In the context of an activity also a *join condition* over the data provided by the links has to be true. Therefore, all incoming links and the input port of the activity synchronize the workflow in the same way as an AND join. Even if we consider incoming links, an activity thus has only the single input port which we mentioned before that can be enabled, blocked, or hidden.

For each outgoing control link, a separate condition can be given to specify whether it will be activated. The *trainTicket* link of Figure 22 is thus only enabled if the *trainReq* parameter is true. Depending on the conditions any combination of activating outgoing control links is therefore possible which means that the links are in an OR relation. Like the output ports of an OR split in YAWL, a dedicated *outgoing link port* can be specified for each combination of the outgoing links. If such a port is enabled, the activation of the

particular links is subject to the corresponding conditions. If the port is blocked and exactly the conditions of the links belonging to the particular port evaluate to true while all other conditions evaluate to false, all the links of the port will still not be activated and thus inhibit a continuation of the process through those links.

As for YAWL or SAP WebFlow, configuration requirements for a configurable BPEL can be specified and evaluated using boolean expressions. To derive the configured BPEL workflow, all activities with hidden input ports are replaced with a dummy invoke activity which completes without any work. All activities with blocked input ports and their sub-blocks are eliminated from the workflow. However, note that this results in semantic problems when sub-blocks of a *sequence* or a *while* loop are blocked. If these sub-blocks are removed the BPEL semantics imply that the execution continues with the next task in the sequence or that the workflow contains an empty loop. If on the other hand a new semantics in line with the configuration idea would prevent this then the configuration would result in a deadlock or livelock which is undesired behavior as well. Therefore, blocking is generally considered as not possible for the input ports of sub-blocks of these two BPEL activity types. Instead, the whole enclosing sequence or while activity should be blocked. Blocked outgoing link ports are simply removed from the workflow. In this way they cannot be activated anymore which is the intension of blocking them.

7. Summary and Outlook

In this paper we presented a general approach to extend common workflow modelling languages such as YAWL, BPEL, or SAP WebFlow with opportunities for predefining alternative model versions within a single workflow model. The approach allows the configuration, i.e. the restriction, of workflow models to a relevant variant in a controlled way. To form a concrete configurable language, it is required to identify the configurable elements within the workflow modelling language and to define their configuration options. A set of model-dependent requirements limits these options and a default configuration conforming to these requirements provides a valid starting point for the configuration of such a model. To demonstrate the approach on a concrete language, we added configuration opportunities to YAWL, and formalized these configurable models. An algorithm for transforming configured C-YAWL models into ordinary YAWL models was provided and a tool demonstrating the applicability of the concepts was implemented.

With the help of the tool, we plan to use a few complex, configurable models and numerous configurations of these models to derive a huge set of configured models. Analyzing the cleanup of these models, e.g. using machine learning techniques, we aim at gaining further insights into interdependencies between configurations of single elements of the net, especially regarding their influence on the net's soundness. This will hopefully allow us to provide further guidance on how to set-up requirements on the configuration, and on how to configure a model under various circumstances.

We are also in the process of applying the ideas presented in this paper to other languages. For example and as briefly highlighted here we have successfully applied the concepts of this paper to SAP WebFlow, a workflow engine shipped more than 100.000 times

with every SAP R/3 enterprise system installation.[17] Combining the ideas of how to set up configurable workflow models with ideas of using domain knowledge to configure process models[50], we are also confident to be able to setup a general framework for system configuration. This framework will not only allow for the synchronized configuration of different workflow systems, but also their alignment with other configurable systems as, e.g., configurable software systems.

References

1. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
2. P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition*. John Wiley and Sons, New York, 1997.
3. T. Curran, G. Keller, and A. Ladd. *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Prentice Hall, Upper Saddle River, NJ, USA, 1998.
4. P. Fettke and P. Loos. Classification of Reference Models – a Methodology and its Application. *Information Systems and e-Business Management*, 1(1):35–53, 2003.
5. P. Fettke, P. Loos, and J. Zwicker. Business Process Reference Models: Survey and Classification. In C. Bussler and A. Haller, editors, *Business Process Management Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 469–483, Berlin Heidelberg, February 2006. Springer Verlag.
6. M. Rosemann. Using reference models within the enterprise resource planning lifecycle. *Australian Accounting Review*, 10(3):19–30, November 2000.
7. G. Keller, M. Nüttgens, and A.W. Scheer. Semantische Prozeßmodellierung auf der Grundlage Ereignisgesteuerter Prozeßketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, 1992.
8. A. Arkin et al. Business Process Modeling Language (BPML), Version 1.0, 2002.
9. Pallas Athena BV. *Protos User Manual*. Plasmolen, The Netherlands, 2004.
10. Staffware plc. *Staffware 2000 / GWD User Manual*. Berkshire, UK, 2000.
11. Staffware plc. *Staffware Process Suite Version 2 – White Paper*. Maidenhead, UK, 2003.
12. A. Rickayzen, J. Dart, C. Brennecke, and M. Schneider. *Practical Workflow for SAP – Effective Business Processes using SAP's WebFlow Engine*. Galileo Press, 2002.
13. W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and Implementation of the YAWL System. In A. Persson and J. Stirna, editors, *Advanced Information Systems Engineering, Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04)*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159. Springer-Verlag, Berlin, 2004.
14. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
15. 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.
16. T. Rieke and C. Seel. Supporting Enterprise Systems Introduction through Controlling-enabled Configurative Reference Modeling. In J. Becker and P. Delfmann, editors, *Reference Modeling. Efficient Information Systems Design Through Reuse of Information Models*, pages 79–102, 2007.
17. F. Gottschalk, W.M.P. van der Aalst, and M.H. Jansen-Vullers. SAP WebFlow Made Configurable: Unifying Workflow Templates into a Configurable Model. In G. Alonso, P. Dadam, and M. Rosemann, editors, *BPM 2007*, volume 4714 of *Lecture Notes in Computer Science*, pages 262–270, Berlin Heidelberg, September 2007. Springer Verlag.

40   *F. Gottschalk, W.M.P. van der Aalst, M.H. Jansen-Vullers, and M. La Rosa*

18. J. Becker, P. Delfmann, A. Dreiling, R. Knackstedt, and D. Kuropka. Configurative Process Modeling – Outlining an Approach to increased Business Process Model Usability. In *Proceedings of the 15th IRMA International Conference*, New Orleans, 2004. Gabler.

19. J. Becker, P. Delfmann, and R. Knackstedt. Adaptive Reference Modelling: Integrating Configurative and Generic Adaptation Techniques for Information Models. In J. Becker and P. Delfmann, editors, *Reference Modeling. Efficient Information Systems Design Through Reuse of Information Models*, pages 27–58. Springer, 2007.

20. M.-J. Blin, J. Wainer, and C. Bauzer Medeiros. A Reuse-Oriented Workflow Definition Language. *International Journal of Cooperative Information Systems*, 12(1):1–36, 2003.

21. W.M.P. van der Aalst, A. Dreiling, F. Gottschalk, M. Rosemann, and M.H. Jansen-Vullers. Configurable Process Models as a Basis for Reference Modeling. In C. Bussler and A. Haller, editors, *Business Process Management Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 512–518. Springer Verlag, February 2006.

22. M. Rosemann and W.M.P. van der Aalst. A Configurable Reference Modelling Language. *Information Systems*, 32(1):1–23, March 2007.

23. G. Keller, M. Nüttgens, and A.-W. Scheer. Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)" (in German). Technical report, Institut für Wirtschaftsinformatik, Saarbrücken, 1992.

24. P. Soffer, B. Golany, and D. Dori. ERP modeling: a comprehensive approach. *Information Systems*, 28(6):673–690, September 2003.

25. F. Puhlmann, A. Schnieders, J. Weiland, and M. Weske. Variability Mechanisms for Process Models. PESOA-Report TR 17/2005, Process Family Engineering in Service-Oriented Applications (PESOA), June, 2005.

26. D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

27. V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger. XML-Based Feature Modelling. In *International Conference on Software Reuse (ICSR)*, pages 101–114, Madrid, Spain, 2004.

28. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, USA, 1990.

29. K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In Robert Glück and Michael Lowry, editors, *4th International Conference on Generative Programming and Component Engineering, GPCE 2005*, volume 3676/2005 of *Lecture Notes in Computer Science*, pages 422–437, 2005.

30. D. Karastoyanova, F. Leymann, and A. Buchmann. An Approach to Parameterizing Web Service Flows. In *Service-Oriented Computing - ICSOC 2005*, volume 3826/2005 of *Lecture Notes in Computer Science*, pages 533–538, 2005.

31. A. Dreiling, M. Rosemann, and W.M.P. van der Aalst. From Conceptual Process Models to Running Workflows: A Holistic Approach for the Configuration of Enterprise Systems. In *Proceedings of the 9th Pacific Asia Conference on Information Systems*, pages 363–376, Bangkok, Thailand, 2005.

32. R. S. Pressman. *Software Engineering: A Practitioners Approach*. McGraw-Hill, 6 edition, 2004.

33. J. Estublier and R. Casallas. *The Adele Software Configuration Manager*, chapter 4, pages 99–139. Configuration Management. J. Wiley and Sons, 1994.

34. E. Turkay, A.S. Gokhale, and B. Natarajan. Addressing the Middleware Configuration Challenges using Model-based Techniques. In *Proceedings of the 42nd ACM Southeast Regional Conference*, pages 166–170, Huntsville AL, USA, 2004.

35. I. Classen, H. Weber, and Y. Han. Towards Evolutionary and Adaptive Workflow Systems-infrastructure Support Based on Higher-Order Object Nets and CORBA. In *Proceedings of the*

1st International Enterprise Distributed Object Computing Conference (EDOC '97), pages 300–308, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

36. G. Faustmann. Configuration for Adaptation – A Human-centered Approach to Flexible Workflow Enactment. *Computer Supported Cooperative Work (CSCW)*, V9(3):413–434, November 2000.

37. Y. Han, T. Schaaf, and H. Pang. A Framework for Configurable Workflow Systems. In *Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems*, pages 218–224, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

38. Y. Han, A. Sheth, and C. Bussler. A Taxonomy of Adaptive Workflow Management. In *Workshop of the 1998 ACM Conference on Computer Supported Cooperative Work*, Seattle, Washington, USA, November 1998.

39. P. J. Kammer, G. A. Bolcer, R. N. Taylor, A. S. Hitomi, and M. Bergman. Techniques for Supporting Dynamic and Adaptive Workflow. *Computer Supported Cooperative Work (CSCW)*, V9(3):269–292, November 2000.

40. S. Rinderle, M. Reichert, and P. Dadam. Disjoint and Overlapping Process Changes: Challenges, Solutions, Applications. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE*, volume 3290, pages 101–120, January 2004.

41. S. Tam, W.B. Lee, W.W.C. Chung, and E.L.Y. Nam. Design of a re-configurable workflow system for rapid product development. *Business Process Management Journal*, 9(1):33–45, February 2003.

42. F. Gottschalk, W.M.P. van der Aalst, and M.H. Jansen-Vullers. Configurable Process Models – A Foundational Approach. In J. Becker and P. Delfmann, editors, *Reference Modeling. Efficient Information Systems Design Through Reuse of Information Models*, pages 59–78. Springer, July 2007.

43. W.M.P. van der Aalst and T. Basten. Inheritance of workflows: an approach to tackling problems related to change. *Theoretical Computer Science*, 270(1-2):125–203, January 2002.

44. T. Basten and W.M.P. van der Aalst. Inheritance of behavior. *Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001.

45. W.M.P. van der Aalst and T. Basten. Identifying Commonalities and Differences in Object Life Cycles using Behavioral Inheritance. In J.M. Colom and M. Koutny, editors, *Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 32–52. Springer-Verlag, Berlin, 2001.

46. M. Adams, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Implementing Dynamic Flexibility in Workflows using Worklets. BPM Center Report BPM-06-06, BPMcenter.org, 2006.

47. N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22, BPMcenter.org, 2006.

48. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

49. YAWL Home Page. http://www.citi.qut.edu.au/yawl/.

50. M. La Rosa, J. Lux, S. Seidel, M. Dumas, and A. H. M. ter Hofstede. Questionnaire-driven Configuration of Reference Process Models. In *Proceedings of the 19th International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 4495 of *LNCS*, pages 424–438, Trondheim, Norway, June 2007.