Chapter 18
Process Configuration

Florian Gottschalk and Marcello La Rosa

18.1 Introduction

The Order Fulfillment process of Genko Oil was set up based on recommendations from the Voluntary Inter-industry Commerce Solution Association (VICS).\(^1\) These recommendations are best practices derived from the experience of hundreds of organizations in the area of logistics and supply chain management. Although these companies offer products varying in many aspects, they all have similar order fulfillment processes in place. For this process, an efficient collaboration with many trading partners is very important. Thus, companies prefer to use standards and suggestions, such as the VICS recommendations, for setting up their workflows instead of trying to innovate in executing these processes.

Of course, not all companies have the same requirements on order fulfillments. For example, Zula Exquisite might only provide Full Truck-Load and Single Package Shipments, and neglect Partial Truck-Loads. Contrary to Genko Oil, they therefore do not need to implement those parts of the Carrier Appointment sub-process dealing with Partial Truck-Loads. Still, there will be many commonalities with the order fulfillment process deployed at Genko Oil. Thus, instead of re-implementing this process from scratch, it is far more attractive for Zula Exquisite to amend the YAWL workflow model deployed at Genko Oil to their needs.

Process configuration can prove beneficial when process adaptations are needed to suit specific requirements, e.g. if a process model that depicts a certain business process in a general manner should be adapted to the requirements of an individual organization. It restricts the possible behavior with respect to a previously built process model in a controlled way before the process’ execution. The configuration of a process model therefore takes place in an adaptation phase between build time and runtime of a process model. Hence, at build time process designers can integrate several variants of how a particular process can be executed into a single process model. Afterwards, the model user selects those parts relevant to the spe-

\(^1\) http://www.vics.org
pecific setting (e.g. an organization or project). All irrelevant model parts can then be automatically removed before the process is executed at runtime. This approach not only allows organizations like Zula Exquisite to adapt an existing model for the same process to individual needs, but it also enables organizations like VICS to provide templates for workflow models which are easily adaptable to user requirements while still conforming to their recommendations. These templates are called *configurable reference process models*. Furthermore, process configuration decisions can be mapped to domain related, natural language questions. In this way, users which are not very proficient in workflow modeling can adapt a process model through simply answering a questionnaire.

This chapter is organized as follows. Section 18.2 discusses in more detail what process configuration means. Afterwards, Section 18.3 introduces a configurable YAWL (C-YAWL) notion by showing how YAWL processes can be restricted through process configuration. How such a configuration can be steered through a questionnaire with domain-related questions is depicted in Section 18.4. Section 18.5 finally shows how the configuration decisions of a C-YAWL model lead to a new, executable YAWL model. The corresponding software tools are introduced in Section 18.6. The chapter concludes with a summary and suggestions for further readings.

### 18.2 How Does Process Configuration Work?

When developing a configurable workflow model, the aim is a model that can be adapted without any manual modeling efforts. Thus, during such an adaptation phase it is not possible to add behavior that has not been modeled beforehand. Instead, all the possible behavior must already be contained in the configurable model. The adaptation of the model to individual needs is then based on restricting this behavior.

To understand how the process behavior of a YAWL model can be restricted, theoretic results from analyzing inheritance relations between process models must be explored first. For this, let us abstract for a moment from YAWL and instead use the very simple notation of labeled transition systems (LTSs) to depict the process flow. This will help understanding the essence of process configuration without being influenced by a specific process modeling language like BPMN, EPCs, BPEL or YAWL. Afterwards we will discuss how these theoretical results can be applied to the behavior represented in a YAWL workflow model.

An LTS is a directed graph where transitions depict the possibilities to switch between states. The states are represented as the nodes of the graph, while the transitions are represented as the arcs in between the states. For an example, let us have a look at the LTS labeled $A$ on the left-hand side of Figure 18.1. Initially that process is in state $s_1$. In this state there is a choice to do either $a$, $b$, or $c$ leading to states $s_2$, $s_3$, or $s_4$. In $s_3$ there is the choice to do either $d$ or $e$. Executing $d$ leads to state $s_2$, i.e. the same state that is reached by executing $a$ initially and so on. The process
completes when the final state \( s_6 \) is reached either from \( s_2 \) through executing \( g \) or from \( s_5 \) by executing \( h \). Thus, an LTS can always be only in one state at a time and the various transitions leaving a state depict the options for state changes.

![Diagram](image)

**Fig. 18.1** The configuration of an LTS can be represented through inheritance relations.

When configuring a model, the goal is to execute a subset of this behavior only. For example, in LTS \( B \) on the right-hand side of Figure 18.1 no transition other than \( a, b, g, \) and \( h \) can be executed. More explicitly, the behavior should always be that either the execution of \( a \) is followed by \( g \) or the execution of \( b \) is followed by \( h \). That means, compared to the original model the execution of \( e \) between \( b \) and \( h \) must be skipped, which is indicated through the grey arc labeled \( \tau \).

Inheritance relations between two models can be used to identify two operators necessary to configure such models. The basic idea of inheritance is to construct different subclasses which on the one hand inherit all behavior and features of a common superclass, and on the other hand extend this behavior with additional behavior or features. The inheritance relation between two models is thus the inverse of the configuration relation between the two models where the original model extends the behavior of the configured model (see Figure 18.1).

The first operator completely inhibits functionality of the original model, i.e. the subclass, in the configured model, i.e. the superclass. The corresponding inheritance relation says that a model \( B \) is a superclass of another model \( A \) if it is not possible to distinguish the behavior represented by \( A \) from the behavior represented by \( B \) when only those transitions of \( A \) are executed which also exist in \( B \). In Figure 18.1, for example, the only transitions leaving \( s_1 \) are \( a \) and \( b \). Thus, when comparing the behavior of \( A \) and \( B \), we are not allowed to execute transition \( c \) that exists in \( A \), but not in \( B \). We say that all transitions of \( A \) which are not present in \( B \) are blocked in \( B \).

The second operator also compares the behavior of the two models, but now the execution of all transitions in both models is allowed. During the comparison, however, only those transition executions of the subclass (\( A \)) are taken into consideration that also exist in the superclass (\( B \)). For example, we can execute from \( s_1 \) in \( A \) first
the transition \(b\), then \(e\), and finally \(h\). In model \(B\) we execute first \(b\) and afterwards directly \(h\) as we skip \(e\). If we now consider only the transitions of \(B\), i.e. transitions \(a, g, b\) and \(h\), these two traces are identical. We say, the transitions of \(A\) which do not exist in \(B\), such as transition \(e\) in Figure 18.1, are hidden in \(B\).

Using these two operators in Figure 18.1, \(B\) can thus be derived from \(A\) by blocking \(c, d,\) and \(f\) and hiding \(e\) as can be seen in Figure 18.2a. The difference between both operators becomes clear if, for example, transition \(d\) would be hidden instead of blocked (see Figure 18.2b). In this case, a trace of executing \(b\) and afterwards \(g\) would become possible in the configured model. This is not possible when transition \(d\) is blocked. On the other hand, when \(e\) is blocked instead of hidden, then it is no longer possible to execute \(h\) after \(b\). Hence, both operators are needed to restrict the possible behavior of an LTS.

![Fig. 18.2 Configurations of the LTS.](image)

In the following section, these results are transferred to the configuration of YAWL models.

### 18.3 Configuring YAWL Models

The blocking and hiding operators that were introduced in the previous section apply to the transitions of an LTS. Transitions are the active elements of an LTS as they lead to changes of state in the model. In YAWL the execution of a task leads to a change of the overall state of a process instance. Thus, the active elements of a YAWL model are the tasks. When we want to configure a YAWL model we therefore have to configure its tasks.

Compared to transitions in LTSs, YAWL tasks comprise far more behavior as they allow different split and join patterns, the start of multiple instances, as well as
the cancelation of other tasks through cancelation regions. Therefore, simply saying that we configure a whole task as blocked or hidden would not be fine-grained enough. Instead we have to identify when and how each task can change the overall state of the process. That means, a state change that is represented by a single transition in an LTS corresponds in YAWL to the combination of one of the alternatives how a task can be triggered with one of the alternatives how a task can complete.

### 18.3.1 Input Port Configuration

Let us first analyze when a task can be started. For a YAWL task this is defined through its pre-conditions and the specified join behavior. A task with an XOR-join, for example, has several incoming arcs from (potentially implicit\(^2\)) pre-conditions. The task however only requires one of these pre-conditions to be marked by a token to be executed because it only consumes a single token from one of its marked pre-conditions (see Figure 18.3a). Each of the incoming arcs therefore represents a different alternative to change the state of the process. For that reason, each of these alternatives must be configurable in the way shown for transitions in LTSs, i.e. through blocking and hiding. In the following each of these alternatives of how a task can be started is called an *input port* of the task.

![Fig. 18.3 Ports of a YAWL task.](image)

The amount of input ports is different for tasks with an AND-join. Here, the control flows of all the incoming branches are synchronized (see Figure 18.3b).

\(^2\) see Chapter 2, Section ?? for further details on implicit conditions
That means, for its execution the task needs to consume tokens from all the pre-conditions which implies that it can change the state of the process only if all these pre-conditions are marked with a token. Only this single alternative to execute the task must thus be configurable. In a similar way, there is also only one alternative when a task with an OR-join can be executed and thus change the state of the process. This is, when there is no longer a chance that any further token can arrive in any of the task’s pre-conditions (see Figure 18.3c).

As a transition in an LTS, an input port can be blocked or hidden by configuration. If it is blocked the task cannot be executed through that port. Instead the token must be consumed from the particular pre-condition by another task. If the input port is configured as hidden this also implies that the task itself is not executed. But then the task is still able to consume and forward the token which means the task can still change the state of the process. Only the task’s execution is skipped.

### 18.3.2 Output Port Configuration

How the state of the process is changed depends on the outcome of the task’s execution, i.e. it depends on which conditions are marked after the task’s execution. This is determined through the split behavior of a task. An XOR-split only marks one post-condition (see Figure 18.3d). Thus each post-condition corresponds to a different change of the process state. A task with an OR-split can not only mark a single post-condition, but also any combination of the task’s post-conditions (see Figures 18.3f and 18.3g). As the overall state of a YAWL model depends on which conditions are marked, each of these marking combination conforms to a different overall state change. A task with an AND-split simply marks all its post-conditions after its execution (see Figure 18.3e). Thus in this case only a single alternative to change the state of the model exists.

Not all these state changes might be desired. Thus, it should be possible to inhibit undesired state changes by configuration. For this, each combination in which post-conditions can be marked after the execution of a task is called an output port of the task. To generally inhibit the corresponding state change, this output port can then be explicitly blocked. In this way, the marking of the corresponding set of post-conditions is no longer possible.

An output port cannot be hidden because, considered on its own, it does not correspond to the full state change of a transition in an LTS. The state change depicted by a transition in an LTS rather corresponds to a complete task execution in YAWL, i.e. the consumption of tokens from pre-conditions and the production of tokens in the post-conditions. Each LTS transition corresponds to such a possible combination. Its configuration therefore corresponds to a combination of an input port configuration with an output port configuration. If one of them is blocked the execution of this combination is inhibited which corresponds to the blocking of the

---

3 see Chapter 3 for further details on the semantics of the OR-join
particular transition in the LTS. If the input port is hidden and the output port is enabled, then this corresponds to hiding the particular transition as the task will not be executed but the target state will be reached. If neither the input port nor the output port is blocked or hidden, then the task can be executed as usual. We then say that the ports remain enabled.

18.3.3 Example

Figure 18.4 provides some examples of task configurations, where conditions have been tagged with a unique identifier. The model depicts the configuration of Genko Oil’s Carrier Appointment sub-process such that it is applicable to Zula Exquisite. As mentioned in the introduction, Zula Exquisite ships only full Truck-Loads or Single Packages, but no partial Truck-Loads. To reflect this situation in the YAWL model, the output port from task *Prepare Transportation Quote* to the branch handling Partial Truck-Loads is blocked, indicated by the *Do not enter* symbol on the corresponding arc. To ship single packages, Zula Exquisite uses common express mail services which do not require delivery appointments as these service providers have fixed delivery routes. For that reason, a delivery appointment does not need to be arranged and the particular process step can be skipped. To reflect this in the configuration, the input port of the corresponding task is configured as hidden (indicated by the ‘jumping’ arrow).

The shipment information document which Zula Exquisite uses to ship full Truck-Loads requires both a pre-arranged pickup appointment and a pre-arranged delivery appointment. To enforce the availability of this information, we have blocked all input ports of task *Create Shipment Information Document* that do not enforce this information, i.e. the ports from conditions $c_6$, $c_7$ and $c_8$. Only the two input ports, which require arranging both appointments beforehand, remain enabled. As all appointments are arranged before the shipment information document is created, there is no need to arrange any of those appointments afterwards. Thus, also all output ports of this task leading to such an arrangement are blocked. Task *Create Shipment Information Document* has an OR-split leading to seven output ports: one port to each of the two subsequent tasks *Arrange Pickup Appointment* and *Arrange Delivery Appointment*, one port to condition $c_{10}$, one port triggering the paths to both tasks at the same time, one port triggering the paths to task *Arrange Pickup Appointment* and to condition $c_{10}$, one port triggering both the paths to task *Arrange Delivery Appointment* and to condition $c_{10}$, and one port triggering the paths to all three of these elements. By directly blocking an arc it is implied that all ports involving this arc are blocked (if only a subset of these ports should be blocked, a dedicated list of the blocked ports will be provided as an annotation of the task). Indeed, by evaluating the result of task *Shipment Information Document*, the outgoing branch tagged as [else] would be automatically chosen anyway. However, blocking these ports explicitly, allows skipping the evaluation of the results at runtime, and thus simplifies the process’s runtime execution.
Prepare Route

Prepare Transportation Quote

Estimate Trailer Usage

Arrange Pickup Appointment

Arrange Delivery Appointment

Create Bill of Lading

Create Carrier Manifest

Produce Shipment Notice

Modify Pickup Appointment

Modify Delivery Appointment

Modify Carrier Manifest

Create Shipment Information Document

Fig. 18.4 The Carrier Appointment process configured according to the needs of Zula Exquisite.
18.3.4 Further Configuration Options

Besides the state changes along the control-flow arcs, a YAWL task can also change the state of a process through consuming all tokens from its cancelation region. While this might be desirable in many cases to preserve correct process execution, there are also situations when the task’s influence on other process parts should be restricted. For example, a configurable YAWL model might contain two tasks for doing the same work at different stages of a process. One of these two tasks is in the cancelation region of a third task. If this third task is executed, the first occurrence of the task in question is canceled, i.e. it is no longer guaranteed to be executed. In this case, the second occurrence of the task will guarantee that the work is still executed. However, if the execution of the second occurrence of the task is inhibited by process configuration, this guarantee is lost. Hence, in this case it might necessary, to block the cancelation of the task’s first occurrence. Thus, a cancelation port is assigned to every cancelation region. This cancelation port can then either remain enabled in which case the triggering of an enabled or hidden cancelation task would still cause the cancelation of all behavior in the cancelation region, or it can be configured as blocked, which then inhibits that the execution of tasks in the cancelation region is canceled while executing the task connected to the cancelation region.

YAWL’s multiple instance tasks depict multiple executions of a single task as one task. As such a task represents more behavior than a simple state change, this behavior can also be restricted by process configuration. On the one hand, it is possible to restrict the freedom of how many task instances can be started maximally by decreasing the maximal number. For example, if an original value would allow for a maximum of 10 instances that can be started, this can be restricted in a configuration to a maximum of 5 instances that can be started. On the other other hand, the minimal number of started instances can be increased. For example, a multiple instance tasks might allow for multiple checks while requiring at least one execution of the task, i.e. at least one check. By increasing this minimal number of task instances to two through process configuration, the behavior can be changed such that the check is performed at least twice, i.e. a four-eyes principle is enforced. Thus, both these configuration options limit the freedom how many process instances can be started, increasing the lower bound of task instances that are started or decreasing the upper bound. Furthermore, the threshold value of a multiple instance task determines how many of these started instances need to complete. Also this freedom can be restricted by increasing the threshold. Finally, by restricting a dynamic creation of task instances to a static creation, it is no longer possible to dynamically start additional task instances while the multiple instance task is running.

To summarize, in C-YAWL the behavior of a YAWL model can be restricted by configuration in four ways: by hiding and blocking input ports, by blocking output ports, by blocking a task’s cancelation region, and by restricting the parameters of a multiple instance task (see Table 18.1).

---

4 see Chapter 2, Section ?? for further details on multiple instance tasks
<table>
<thead>
<tr>
<th>Configurable element</th>
<th>No. of ports</th>
<th>Configuration options</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND-join</td>
<td>1</td>
<td>enabled, blocked, hidden</td>
</tr>
<tr>
<td>XOR-join</td>
<td>n</td>
<td>enabled, blocked, hidden</td>
</tr>
<tr>
<td>OR-join</td>
<td>1</td>
<td>enabled, blocked, hidden</td>
</tr>
<tr>
<td>AND-split</td>
<td>1</td>
<td>enabled, blocked</td>
</tr>
<tr>
<td>XOR-split</td>
<td>n</td>
<td>enabled, blocked</td>
</tr>
<tr>
<td>OR-split</td>
<td>$2^n - 1$</td>
<td>enabled, blocked</td>
</tr>
<tr>
<td>Cancelation region</td>
<td>1</td>
<td>enabled, blocked</td>
</tr>
<tr>
<td>Multiple instances</td>
<td></td>
<td>reduce maximal number of instances,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>increase minimal number of instances,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>increase threshold,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>forbid dynamic creation of instances</td>
</tr>
</tbody>
</table>

Table 18.1 Configuration options for a task with $n$ incoming/outgoing process branches.

As configuration operators are applied at the granularity of single tasks, the configuration of a YAWL workflow would typically require a deep understanding of the process model and its notation. Furthermore, each task’s configuration would very likely affect (directly or indirectly) other parts of the process model. For example, the blocking of one port can also indirectly deny the execution of the flow downstream. Therefore, an approach that can streamline the configuration process by abstracting from the intricacies of YAWL (or any other specific modeling notation), would prove beneficial, especially if it could also improve understanding the impact of configuration decisions throughout the workflow model.

In the next section an approach to process configuration will be presented that is based on the use of questionnaires which focus on allowing process users, or more generally, subject matter experts, to reason in terms of the domain concepts involved.

### 18.4 Steering Process Configuration Through Questionnaires

So far we have seen how the behavior of a YAWL model can be restricted by means of, e.g., the blocking and hiding operators. The reason for hiding or blocking a certain port should derive from practical needs for the corresponding functionality, i.e. it should be driven by the requirements of the organization or project for which we want to configure our workflow. That means, before we can answer a technical question like “Do we need to hide the input port of the task *Arrange Delivery Appointment*?”, we first have to ask ourselves “Do we need to arrange delivery appointments when shipping Single Packages?”. Only the answers to such domain-related questions can lead to the necessary configuration decisions on the process model level.
Moreover, configuring a YAWL model to the requirements of a specific setting requires a user to be proficient in both the YAWL notation and the application domain. This assumption might be unrealistic in application domains where users are unfamiliar with modeling notations. This is the case of the supply chain management domain in which the Order Fulfillment process model has been constructed: users of this process would very likely be logistics experts only.

In this section, it will be shown how process configuration can be facilitated via a questionnaire-driven approach, where variations are presented in terms of domain-related questions expressed in natural language, which are mapped to configuration decisions on a YAWL model. Firstly, the features of questionnaire models will be presented, which allow a systematical encoding of questions and their answers. Secondly, it will be shown how such models can be linked with a YAWL model for the automatic derivation of a configured workflow thereof.

### 18.4.1 Questionnaire Models

A questionnaire model is a structured representation of domain choices and their dependencies. Key components of a questionnaire model are domain facts and questions. Domain facts capture domain choices, i.e. features that may or may not exist in a given setting of the domain. For example, three possible features of a logistics domain are “Truck-Load”, “Less than Truck-Load”, and “Single Package”, which denote three shipment types (or products) a logistics department of a company may provide. A domain fact (fact for short) is modeled as a boolean variable which the user can set to true if they are interested in having the respective feature, or false otherwise. For example, at Genko Oil all the above facts are set to true, while for Zula Exquisite “Less than Truck-Load” is false.

Facts can be grouped into questions according to their content, so that facts belonging to the same question can be set at once. In our example, a possible question to group the above three facts is “Which shipment types have to be provided?”.

Questions are expressed in natural language. Therefore they can be answered by domain experts without extensive knowledge of the underlying process model.

Each fact is given a default value which is used to suggest the most recurring value for that fact. In the Order Fulfillment example, fact “Single Package” is assigned a default value of true as the majority of logistics organizations, included Genko Oil, typically provide such a product because it is the easiest of the three to be handled.

Moreover, a fact can be marked as ‘mandatory’ if it needs to be explicitly set by the user. This is used to prevent users from overlooking important aspects of a domain. In fact a question which has at least a mandatory fact must be explicitly answered by the user, while a question with no mandatory facts can be skipped. In this case the question is automatically answered by using the default values of its facts.
Figure 18.5 shows an excerpt of a possible questionnaire model for the supply chain management domain, where all questions and facts are assigned a unique identifier and a description.

```
q3: Which shipment types have to be handled?
f1: Ordering
f2: Logistics
f7: Truck-Load (TL)
M
f8: Less than Truck-Load (LTL)
M
f3: Payment
T
f5: Freight in Transit
T
f4: Carrier Appointment
T
f6: Freight Delivered
T
f9: Single Package (SP)
M
q2: Which Logistics sub-phases have to be implemented?
f11: Pickup Appointment
T
f12: Delivery Appointment
T
q4: Is the Order preparation to be governed by a deadline?
f10: Deadline imposed by the Carrier
T
f13: Pickup Appointment
M
q5: Which appointments have to be pre-arranged for Single Packages?
f14: Delivery Appointment
T
q6: Which appointments have to be pre-arranged for Truck-Loads?
f15: Delivery Appointment
T
q7: Which appointments have to be pre-arranged for Less than Truck-Loads?
```

As we can see, there exist interdependencies among the above questions. For example, an implication of setting to true any fact of $q_2$ (the Logistics sub-phases), is to set $f_2$ (Logistics) to true in $q_1$. This is because only if Logistics is implemented, any of its sub-phases can also be implemented. Similarly, the answer given to the remaining questions ($q_3$ to $q_7$) is affected by the decision taken in $q_2$ whether to enable Logistics or not. Indeed, it would make no sense to inquire after the shipment types to be handled, the deadline on the order preparation, or the arrangement of pickup and delivery appointments, if the organization will not support Logistics. In particular, which appointments can be agreed upon depends on the shipment type that is selected to be handled: the facts of $q_5$ depend on $f_3$, those of $q_6$ on $f_5$ and so on.

Since facts are boolean variables, their interactions are captured in a questionnaire model by means of boolean expressions over their values, which are called
domain constraints. For example, the interaction between \( q_1 \) and \( q_2 \) can be modeled by the constraint \((f_4 \text{ or } f_5 \text{ or } f_6) \text{ implies } f_2\), which essentially means that if any fact of \( q_2 \) is selected, then \( f_2 \) must be set to true as well. On the other hand, if \( f_2 \) is deselected, all the facts of \( q_2 \) must be set to false too. Similar constraints can be defined to capture the interactions among the other questions.

Questionnaire models also allow the definition of an order relation for posing questions to users. There are two types of order dependencies that can be defined among questions: partial and strict dependencies. A partial dependency (represented by a dashed arrow in Figure 18.5) is used to model an optional precedence between two questions: e.g. \( q_3 \) will be posed after either \( q_1 \) or \( q_2 \) have been answered. A full dependency (depicted by a full arrow) captures a mandatory precedence: e.g. \( q_4 \) will be posed only after both \( q_1 \) and \( q_2 \) have been answered. These dependencies can be arbitrary so long as undesired cycles are avoided.

The combined use of constraints and dependencies can be exploited by a supporting tool (cf. Section 18.6) to provide an interactive interface to the user when answering a questionnaire. Specifically, according to the constraints, the answers given to some questions can affect the answers of subsequent questions. For instance, questions \( q_1 \) and \( q_2 \) correspond to high-level decisions: if Logistics is disabled in \( q_1 \), all the subsequent questions become irrelevant (with all their facts being set to false). A similar result is obtained by disabling Carrier Appointment in \( q_2 \). Therefore, if these most discriminating questions are asked first, it is possible to (partly) infer the answer to subsequent questions automatically. In our example, \( q_1 \) and \( q_2 \) have not been assigned any order dependencies so as to be posed first.

In conclusion, a domain configuration is the result of completing a questionnaire where questions are posed in an order consistent with the order dependencies, and each fact is set to a boolean value complying with the domain constraints. This experience is interactive as the user is aided in the completion of the questionnaire according to the choices they take.

18.4.2 From Answers to Configured YAWL Process Models

Questionnaire models are meant to be constructed and linked to configurable YAWL process models by experienced modelers. This should be done in collaboration with subject matter experts, who possess expertise in the application domain. By answering questions, domain experts assign values to domain facts and once the questionnaire has been completed, the resulting facts’ valuation is used to configure the respective YAWL process model through an individualization algorithm. An overview of this approach is depicted in Figure 18.6.

The idea is that each configuration decision for a port, a task’s multiple instance parameter or a cancelation region in a YAWL model is captured by a boolean variable, namely \textit{process fact}. Only if the process fact is true, the specific configuration is applied, e.g. an input port is hidden. Each process fact is then linked to the domain facts of a questionnaire model through a boolean expression. Such expressions
embody the constraints of the domain. Thus, a process fact is set to true whenever the corresponding boolean expression evaluates to true.

Let us have a look at some mapping examples. Question $q_6$ asks whether to arrange pickup and delivery appointments for Single Packages. Here domain fact $f_{14}$ is set to true if delivery appointments have to be arranged, and to false otherwise. This fact can be mapped to the input port of task *Arrange Delivery Appointment* (in the process branch handling Single Packages), such that when $f_{14}$ is false, the port is set to hidden. Therefore, if the configuration *hidden* of this port is encoded with a process fact, say $p_1$, the boolean expression linking the latter with $f_{14}$ would be: 
\[ \text{not } f_{14} \implies p_1. \]
Likewise, $f_{13}$ is mapped to the value *hidden* of the input port of task *Arrange Pickup Appointment* in the same process branch, so as to skip this task if the pickup does not need to be arranged.

In a similar way, the three facts of question $q_3$ can be mapped to the value *blocked* of the three output ports of task *Prepare Transportation quote*, such that when any of these facts is false, the respective port is blocked.

A more complex mapping exists between the facts of $q_5$ and the process elements in the branch handling Truck-Loads. Depending on the values of $f_{11}$ and $f_{12}$, a number of ports can be blocked at the same time. In case both facts are true, i.e. both appointments need to be pre-arranged, the input ports of task *Create Shipment Information Document* from conditions $c_6$, $c_7$ and $c_8$ will be blocked, as well as its output ports to the [no pickup information] and [no delivery information] branches. The blocking of the input ports allows both the pickup and delivery appointments to be executed (in any order), while the blocking of the output ports avoids the rescheduling of the appointments. This situation is depicted in Figure 18.7a. Similarly, if only one of the two appointments can be pre-arranged, or none of them can be, a different set of ports needs to be blocked. These situations are depicted in Figure 18.7 b, c, and d. Whether a domain fact should be mapped to a hidden port

![Fig. 18.6 The questionnaire-based framework for process models configuration.](image)

- Domain expert
- Modeler
- Questionnaire model
- C-Mapping
- C-YAWL
- Interactive questionnaire
- Answers
- YAWL model
- configuration and individualization
- creates
- answers
or to a blocked port depends on what model transformations one expects to achieve by answering the questionnaire.

**Fig. 18.7** Illustration of how the Truck-Load fragment of the Carrier Appointment process changes on the basis of the answer given to question q5: a) if both the appointments are to be pre-arranged, b) if only the Pickup needs to be pre-arranged, c) if only the Delivery needs to be pre-arranged, d) if there is no pre-arrangement.
The advantage of using questionnaires to configure more than one port at a time is twofold: on the one hand the user can save time during configuration (especially in huge process models), on the other hand it is easier to visualize the impact of a domain decision (e.g. pre-arrange Truck-Load appointments) on the process model. Let us consider the mapping for questions $q_1$ and $q_2$. Their domain facts are mapped to the input ports of the first net in the decomposition hierarchy of the Order Fulfillment process, such that when any of these facts is set to false, the corresponding port is configured as hidden. Figure 18.8 shows the configuration of the Order Fulfillment process when only the Carrier Appointment phase in Logistics is implemented, which corresponds to facts $f_2$ and $f_4$ being true and all other facts being false. In this process model, composite task Carrier Appointment is the only one not being skipped. The complexity of the initial model has been dramatically reduced since only one of the five composite tasks will be executed. This has been achieved by answering only two questions.

![Diagram]

Fig. 18.8 The configuration of the Order Fulfillment process when only Carrier Appointment is selected in the questionnaire.

18.5 Applying Configuration Decisions to YAWL Models

The C-YAWL models to which the configuration decisions are applied, i.e. the models which are used as the basis for configuration, integrate all the process execution variants. By configuration, the intension is to eliminate all the behavior from the
model, which is irrelevant in a specific context. Until now, we have discussed how such intentions can be expressed through configuration decisions. However, we have not yet derived any YAWL models which only contain the desired behavior. For this, obviously, all the undesired parts of the model must be removed. Based on the configuration decisions, this removal can be performed completely automatically in two phases. First, all the direct implications of configuration decisions on tasks and arcs are implemented in the YAWL model. Secondly, the implications of these configuration decisions on other elements in the workflow are resolved in a clean-up phase.

### 18.5.1 Applying Configuration Decisions to Tasks

The first phase starts with adjusting the parameters of multiple instance tasks, as well as implementing blocked cancelation ports and blocked output ports as these are local adaptation. The implementation of the input port configurations should happen only afterwards as this might require some copying of the other configuration decisions.

For adjusting the parameters of multiple instance tasks as already described in Section 18.3.4, the values for the minimal number of instances that need to be started, and the threshold for the number of instances that need to complete for a successful completion of the overall task are increased according to the corresponding configuration values. Also, the maximal number of instances that can be started is decreased according to the corresponding configuration value. Furthermore, the instance creation is changed to a static creation if necessary. In the example from Section 18.3.4, the task parameters could change from $[1,10,1,dynamic]$ to $[2,5,2,static]$ in this way.

Implementing blocked cancelation regions in the configured model is similarly straightforward. For all tasks with such a blocked cancelation region, all references to the elements contained in the cancelation region are simply deleted. For the enabled cancelation regions the references are preserved.

The configuration of output ports influences how threads of control can leave the task. If a port is blocked, there should be no way that the corresponding, subsequent conditions are marked. In case of an XOR-split, each port refers exactly to one outgoing arc. Thus, if this port is blocked, the corresponding arc can simply be removed. In this way, any process flow through this port that would mark the particular, subsequent condition is inhibited. Inhibiting the process flow is not that simple in case of an OR-split. Here several ports can refer to the same arc. In this case, the arc can only be removed if all the ports referring to the arc are blocked. For example, let us have a look at the first configuration in Figure 18.9. Here we can remove the arc to $c_4$ as all ports referring to it are blocked. The only ports enabled

---

5 Note, that if the arc that is chosen by default, i.e. if the predicate of all other arcs evaluate to false, is removed, YAWL automatically defines the last evaluated arc as the new default arc and uses it, even if the corresponding predicate evaluates to false.
are the ones leading to \( c_2 \) and to both \( c_2 \) and \( c_3 \). Although the port to \( c_3 \) is blocked, the arc to \( c_3 \) must be kept in the net because of the enabled port leading to both \( c_2 \) and \( c_3 \). The blocking of such ports where the corresponding arc has to remain in the net is then realized through adapting the arc’s predicate.\(^6\) By concatenating the predicate from the arc to \( c_2 \) with the original predicate from the arc to \( c_3 \) as the new predicate for the arc to \( c_3 \), the blocking of the port to (only) \( c_3 \) can, e.g., be implemented for the first configuration of Figure 18.9. That means, that if the predicate for the arc leading to \( c_2 \) would be \( x > 25 \) and the original predicate for the arc leading to \( c_3 \) was \( y < 100 \) then the new predicate for the arc to \( c_3 \) would be \((x > 25) \land (y < 100)\). The condition \( c_3 \) can then only be marked if the arc’s original predicate is true (i.e. \( y < 100 \)), and in addition the predicate to \( c_2 \) is true (i.e. \( x > 25 \)), which thus always implies the marking of \( c_2 \) in addition (for which \( x > 25 \) is the only condition).

\[\text{Fig. 18.9} \quad \text{Three configurations of a task with an OR-split transformed into new models.}\]

\(^6\) see Chapter 8, Section ?? for details on how to set up predicates
In two special cases, this complex changing of predicates can be avoided. Firstly, if all the output ports referring to a combination of arcs are blocked, then it is possible to change the split behavior of the task from an OR-split to an XOR-split and keep the predicates as they were before. For example, in Configuration 2 of Figure 18.9 the port leading to c2 only as well as the port leading to c4 only are enabled, while all other ports are blocked. Thus, a simple XOR-split between these two process flows is sufficient. If, on the other hand, all ports but one are blocked as in Configuration 3 of Figure 18.9, the behavior is exactly captured by an AND-split. All predicates of the resulting task are simply set to true.

After having done these three transformations, the input port configurations of tasks can be implemented. Task with an AND-join or OR-join have just a single input port. In case this port is blocked, the corresponding task is not allowed to be executed. Therefore, all arcs leading to the task are simply removed. In case this port is hidden, the incoming arcs cannot be simply removed — the process flow has to continue in this case. However, the task should not be executed either. Thus, in these cases any task decomposition\(^7\) must be removed and the task has to be transformed into a so-called silent task (or \(\tau\) task). That means, in this way, the task no longer leads to any work or to any changes of data, corresponding to the behavior of \(\tau\) transitions in LTSs.

Tasks with an XOR-join have several input ports. The configuration of these ports can therefore vary. Each port refers, however, to exactly one arc. Hence, to inhibit the flow through a blocked input port, the corresponding arc can simply be removed. This is, e.g., the case in Figure 18.10, where the input port corresponding to the arc from c1 to the task is blocked. In the resulting model on the right, the arc is therefore removed. Besides being blocked, an input port can also be hidden, such as the ports from c3 and c4 in Figure 18.10. The task’s decomposition must then be skipped. However, in case of an XOR-join, the decomposition of the task cannot be generally

\(^7\) see Chapter 8, Section ?? for details on how to assigning decompositions like codelets, external applications, or sub-nets to tasks
skipped for all process instances as the decomposition might still be needed by instances arriving through other, enabled ports of the same task (such as the port from \(c_2\) in Figure 18.10). For that reason, it is necessary to duplicate the current task (i.e. with multiple instance, cancelation port, and output port configurations already applied). While one copy remains the original task, the task’s decomposition is eliminated from the other copy, i.e. this copy becomes the silent \(\tau\)-version of the task without any behavior. Any multiple instance parameters or cancelation regions remain the same for both copies. Also the outgoing arcs of both tasks target at the same subsequent conditions. In case of a direct connection to the subsequent tasks in the original model, an intermediate condition will be introduced for this (i.e. the implicit conditions between the two tasks will be made explicit). Then, the arcs of all enabled ports are directed to the original task while the arcs of hidden input ports are re-directed to the silent copy (compare the arc from \(c_2\) with the arcs from \(c_3\) and \(c_4\) on the right of Figure 18.10).

Figure 18.11 shows the resulting YAWL model after applying all the configuration decisions depicted in Figure 18.4 as described. All undesired routings of the process are no longer possible and the \textit{Arrange Delivery Appointment} task for single packages is replaced with a silent \(\tau\) task such that it can be skipped. However, while the arcs incorporating undesired behavior are removed, the undesired tasks themselves are still in the net. Just their reachability is inhibited. The second phase thus has to rid the model of such non-executable elements.

\subsection{18.5.2 Cleaning-up the Configured YAWL Model}

For being reachable, a model element needs to be on a path from the input condition. Additionally, a YAWL model requires that the final condition indicating the workflow’s completion can always be reached, i.e. there also needs to be a path from every element to the final condition. To identify all the elements that are on such a path, we simply search for all paths from the initial condition to the final condition and mark all elements that are on a path. All the non-marked elements will be removed from the net.

When removing all these elements it is important to take into consideration that a task with an AND-join needs tokens in all its preceding conditions to execute. If a condition that precedes an AND-join is removed, this condition can never be marked anymore and hence the task should not be executed any longer either. For that reason, tasks subsequent to a removed condition have to be removed whenever they have an AND-join behavior. As this removal of a task might break an existing path between the input condition and the output condition (because the task has been on such a path), the check for paths between the input condition and the output condition has to be repeated afterwards. The removal of the elements not on such a path must be continued until a search confirms that all elements are on such a path (or the only remaining elements are the input and output conditions which then indicates a misconfiguration).
the freight is physically picked up

order preparation took too long

Fig. 18.11 The Carrier Appointment process after applying the configuration decisions, but without a clean-up of the process.
Fig. 18.12 The cleaned YAWL model for the Carrier Appointment process of Zula Exquisite.
If all elements are on a path between input condition and output condition, the clean-up is complete. The resulting model, such as the one in Figure 18.12 for the Carrier Appointment sub-process of Zula Exquisite, can then be directly loaded as a workflow specification in the Engine. It can also be imported into the Editor to make further manual changes, e.g. to update the necessary resources for certain tasks.

18.6 Tool Support

YAWL workflow models can be configured through the *Synergia* toolset. The purpose of this toolset is to foster synergism between domain models and process models via the use of questionnaires. *Synergia* assists domain experts and process modelers with creating questionnaire models, mapping questionnaires to process models, answering questionnaires and applying the result of a questionnaire to a process model for the configuration of the latter.

*Synergia* consists of a set of rich client applications, with each application taking care of a specific task. The architecture, shown in Figure 18.13, is pluggable, thus allowing other tools to be added in a seamless manner. Currently, the toolset supports the configuration of process models defined in C-YAWL and in C-EPC (the latter being the configurable extension to the Event-driven Process Chains (EPC) language.

The first tool is the *Questionnaire Designer*, which allows users to visually create questionnaire models. Besides the definition of questions, domain facts, order dependencies and domain constraints, it is possible to assign textual guidelines to questions and facts, which will be used to provide advice to users while the questionnaire is being answered. Moreover, the tool can detect cyclic dependencies in the order of questions and inconsistencies in the domain constraints. This prevents the user from producing invalid questionnaire models that may deadlock while being answered due to an order dependency that cannot be resolved. A screenshot of the tool with its validation feature is shown in Figure 18.14.
A serialization of a questionnaire model (.qml) is generated by the Questionnaire Designer tool. This file can be imported into Quaestio, which is the second tool of Synergia. Quaestio prompts questions to users in an order consistent with the order dependencies defined in the questionnaire model. The tool is interactive as it prevents users from entering conflicting answers to subsequent questions by dynamically enforcing the domain constraints according to the answers given at any time. In this way, questions are only posed if they are relevant to the context. Questions can be explicitly answered or can be skipped if they do not contain mandatory facts. In this case they are answered by using the default values of their facts. Furthermore, questions already answered can be rolled back should a decision need to be reconsidered. A screenshot of Quaestio showing the questionnaire for the Order Fulfillment process model is depicted in Figure 18.15. It represents the state in which question $q_1$ (“Which phases have to be implemented?”) has already been answered and the user is presented questions $q_2$ (“Which Logistics sub-phases have to be implemented?”) and $q_3$ (“Which shipment types have to be handled?”), with the latter question having become available after answering $q_1$. The facts and guidelines for $q_2$ are also shown.

Once a questionnaire has been completed, it can be exported as a domain configuration (.dcl). This file can be used by the Process Configurator tool to configure...
a process model, i.e. to apply the configuration decisions to the process model as shown in Section 18.4.2. The Process Configurator needs a serialization of a process model (e.g. .yawl for YAWL) and a configuration mapping (.cmap) between the process model and the questionnaire model. Mappings can be generated by the Mapper tool, which allows the connection of process facts to domain facts by means of boolean expressions.

Finally, the Process Individualizer tool performs the individualization algorithm as discussed in Section 18.5. Given a configured process model as input, this tool takes care of removing all those process fragments that are no longer required, and generates a new process model as output. The algorithm preserves the syntactic correctness of the input model: the resulting model will be well-formed provided the input model is well-formed. The tool currently generates YAWL models from configured C-YAWL models and EPC models from configured C-EPC models.

18.7 Summary

Process configuration enables domain experts, who are usually not proficient in modeling notations, to adapt process models to specific settings, e.g. a new organization or project. While process configuration does not extend the behavior represented by a process model, it allows users to restrict the already modeled behavior in such a way that only desired steps of the process will be executed. Thus, a con-
A configurable process model is meant to incorporate execution variants from which the user can select.

To provide configuration opportunities to YAWL models, input and output ports are defined for each task of a YAWL model. By blocking such ports, the process flow through the port is inhibited and thus tasks subsequent to this port will not be executed. An input port of a task can also be hidden. This means, the execution of the task itself should be skipped, while the process flow continues afterwards. Moreover, the behavior of multiple instance tasks and tasks’ cancelation regions can be restricted through process configuration. This extension to the YAWL notation is called C-YAWL.

Modeling and configuring the variations of a process model in terms of input and output ports being blocked or hidden requires a thorough knowledge of both the YAWL modeling notation and of the domain in which the process has been constructed. An approach based on the use of questionnaires can streamline the configuration of process models. A questionnaire model defines a set of questions grouping the features a domain can support. The possible values these features, called domain facts, can take, are restricted by a set of domain constraints, which model the interdependencies of the domain. Also, a questionnaire model defines the order in which questions can be posed to users.

Questionnaire models are then linked to the ports of a YAWL process model by means of boolean expressions. Whenever such an expression evaluates to true on the basis of an answer to the questionnaire, the given port is configured accordingly. In this way it is possible to configure a number of YAWL ports simultaneously by answering a single question. This also applies to the configuration of multiple instance tasks and cancelation regions.

The fact that questionnaire models are expressed in terms of domain concepts, enables subject matter experts, who are usually not proficient in modeling notations, to benefit from process configuration. Moreover, it allows them to better estimate the impact of a domain decision (e.g. disabling Truck-Loads shipments) throughout the process model.

Once a C-YAWL process model has been configured, it can automatically be transformed into a well-formed YAWL model including only those process variants that have been selected in the configuration. To load this new process specification into the YAWL engine, the workflow developer only needs to adjust the resource assignment and the necessary data parameters.

In this chapter, the benefits of using process configuration for configurable reference process models were demonstrated. However, the concepts of process configuration and questionnaires might have a wider applicability beyond the scope of reference models. For example, it can be useful to apply process configuration when implementing seasonal switches between different process variants. Questionnaires could also be used to steer decisions independent of a process model’s control-flow. In general, the various elements of process configuration can be beneficial whenever a selection from a set of pre-defined variants should be made.
Exercises

1. Via the use of input and output ports, configure the Payment sub-process of Figure .5 so as to:
   - disable the payment for the freight (only the payment for the shipment is to be performed);
   - disable the approval of Shipment Payment Orders;
   - disable the possibility of having Debit adjustments.

   How will the resulting process look like after applying this configuration?

2. Can the cancelation region of task *Carrier Timeout* be blocked without violating the soundness of the net?

3. Complete the questionnaire model of Figure 18.5 to incorporate the variations of the Payment sub-process defined in Exercise 1. Also, create a mapping between the new facts and the new ports in terms of boolean expressions.

4. Using Worklet Services (cf. Chapter 11), several sub-processes can be assigned to a single composite task in YAWL, i.e. there is a runtime choice among the execution of the various sub-processes. How could such a choice be restricted by configuration? Define the required ports and their possible configuration values in the configuration of Worklets.

5. The following Ordering sub-process (Figure 18.16) is used at Zula Exquisite. Combine it with the Ordering sub-process at Genko Oil (Figure .3), so as to obtain a single YAWL model. Afterwards, determine the necessary configurations to derive the two original models again.

![Diagram](image-url)

Fig. 18.16 The ordering sub-process at Zula Exquisite.
This chapter introduced the topic of process configuration through the use of LTSs and the application of the concept of inheritance of workflow behavior. The interested reader can find more information on the configuration of process models defined as LTSs in [6] and on the inheritance of workflow behavior in [1]. After this introduction, the chapter focused on the configurable YAWL notation and presented a questionnaire-based approach to steer the configuration of YAWL models. The technical details of the C-YAWL notation are provided in [7], while a complete description of questionnaire models can be found in [11]. The application of questionnaire models to the configuration of process models defined in C-YAWL and C-EPC is discussed in [13, 10], whereas in [8, 14] the reader can find insights on the application of this approach in practice.

Furthermore, this chapter presented an individualization algorithm that transforms a configured YAWL process model while preserving the syntactical correctness of the model. The formal algorithm is available in [7], while an in-depth discussion on the topic of correctness in process configuration can be found in [2]. Specifically, here the authors define a technique for incrementally preserving the correctness of a process model during its configuration, given some assumptions on the structure of the configurable model. This technique is first defined in the context of Petri nets and then extended to the specificities of C-EPCs.

More information on process configuration can also be found on the Process Configuration web-site, where the Synergia toolset described in this chapter is available for download along with several examples.

The possibility of re-using process models as best-practice templates has led to a number of commercial reference models for various industry sectors. For example, the Order Fulfillment model is inspired by the Voluntary Inter-industry Commerce Solutions (VICS) model. Other well-known examples are the Information Technology Infrastructure Library (ITIL), the Supply Chain Operations Reference Model (SCOR), and the SAP R/3 reference model [9], which was utilized to capture the processes supported by the SAP’s ERP system. An overview and classification of reference models can be found in [5].

Although existing reference models as the above ones depict recommendations for the execution of specific processes, they hardly provide insights on how these recommendations can be adapted to the individual needs of an organization. In this chapter we presented process configuration for YAWL models as one such an adaptation mechanism. A similar mechanism is suggested in [17] for the configuration of process models defined in EPCs (via the C-EPC extension). The approach in [3] is based upon the principle of model projection. Since a reference process model typically contains information for multiple application scenarios, it is possible to create
a projection of an EPC model for a specific scenario (e.g. a class of users) by fading out those process branches that are not relevant to the scenario in question. Another proposal is defined in [15], which relies on stereotype annotations to accommodate variability in a so-called variant-rich process model. Although stereotypes are an extensibility mechanism of UML, in this proposal they are applied to both UML Activity Diagrams and BPMN models. A subset of these stereotypes also appears in [16] while the general idea of annotating model elements to represent variability has also been investigated in [4].

The above approaches, including the one presented in this chapter, focus on the configuration of the process control-flow. For the application of configuration mechanisms to other process perspectives such as the resources and the business objects participating in a process, the reader is referred to [12]. Here the authors define a process meta-model to capture complex task-resource and task-object associations and configurations thereof. This meta-model is embodied in an extension of the C-EPC notation, namely integrated C-EPCs (C-iEPCs).

Process configuration allows users to adapt process models by restricting their behavior. However, other adaptation mechanisms are available in literature, such as modification, abstraction, generalization and specialization. A comparison of different adaptation mechanisms is provided in [3]. This might help the reader to identify if process configuration is the required way of adapting a process model to their specific scenario.
References


