RZ 3678 (# 99688) 05/21/07
Computer Science 40 pages

Research Report

Process Anti-Patterns: How to Avoid the Common Traps of Business Process Modeling

Jana Koehler and Jussi Vanhatalo

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

{koe,juv}@zurich.ibm.com

Process Anti-Patterns: How to Avoid the Common Traps of Business Process Modeling

Jana Koehler **Jussi Vanhatalo**
IBM Zurich Research Laboratory
CH-8803 Rueschlikon
Switzerland
email: {koe,juv}@zurich.ibm.com
Research Report RZ-3678

Abstract

Business process modeling is gaining increasing importance with more and more people getting involved with business process modeling projects. The output of these projects are process models, which become a direct input into the software development process. Consequently, the impact of the process models on the IT systems and the operational efficiency of an enterprise is increasing. With that, the associated economic risk of using badly designed process models is growing as well. In this report, we address the problem of quality assurance for business process models. Based on hundreds of real world business process models that we reviewed over the past two years, we extracted typical modeling errors that we generalized into anti-patterns. These anti-patterns cover six common process modeling scenarios ranging from the modeling of branching and iterative behavior, over the modeling of data flow, to the reuse of process models in composite processes. For each scenario, an example illustrating typical errors is introduced and then generalized into an anti-pattern, which highlights the modeling error. Then, one or several patterns are presented that show a correct solution to the modeling scenario, followed by a summarizing recommendation.[1]

1 Introduction

Business process modeling is gaining increasing importance. Not only is the number of people modeling business processes increasing, but more and more people with different professional backgrounds are involved with business process modeling projects. Business processes are one of the most important assets of an organization. With the trend of enterprises to base their IT on a Service-Oriented Architecture [4], business processes become first-class citizens also in the supporting IT systems. Consequently, business process modeling is no longer a modeling exercise for documentation and discussion purposes, but the process models become a direct input into Business-Driven Development [14, 10] where they develop a direct economic effect on the IT systems and the processes associated with developing and maintaining these IT systems. Consequently, a significant economic risk can be associated with the design of process models and the question of how to produce process models of high quality is receiving increasing attention [2, 7, 8].

Unfortunately, no good way has been developed so far to measure the quality of business process models. The underlying processes themselves are usually measured with economic key performance indicators such as cost and profit. State-of-the-art modeling tools provide analytical capabilities to gather insights into these economic aspects of processes. However, most analytical tools do not address quality

[1]This work was published in the WebSphere Developer Technical Journal in 2007. Please refer to the official publication at http://www.ibm.com/developerworks/websphere/techjournal/0702_koehler/0702_koehler.html for citation.

requirements. Sometimes, an analytical result can even be incorrect because of hidden errors in the model that remained undetected.

When exploring related work in the scientific literature, we found only very few papers that address the measurement of quality for process models. Six *guidelines* (principles) to modeling are formulated in [2, 17]. These principles are *correctness, relevance, economic efficiency, clarity, comparability,* and *systematic design*. Each of the guidelines is informally described and it is discussed how following the guideline can improve the quality of the models. However, no quantifiable criteria are given that would allow a tool to directly measure the quality of a process model in an objective manner. In [7, 8], the ISO/IEC 9126 Software Product Quality Model is adopted and the quality of process models is measured based on the criteria of *functionality, reliability, usability,* and *maintainability*. These measures are so far estimated based on the subjective evaluation by a human expert, who, for example, has to assess whether an activity in a process model is functionally adequate for the process. An experimental study that measures the degree of misinterpretations by users caused by inconsistencies between sequence diagrams and class diagrams is described in [12]. Several types of defects are introduced, which focus on static and syntactic aspects of models, but are not applicable to behavioral, i.e., process models.

Patterns play a very important role in the workflow community, mostly due to the pioneering work on workflow patterns as a basis for the comparison of workflow engines [25]. Two initial investigations on the role of patterns in the process of *designing* behavioral models are described in [15] and [5]. While [15] focuses on using patterns to ease the implementation of business processes in a service-oriented architecture, the work described in [5] investigates how patterns can be used to capture non-functional aspects in process models such as domain-specific quality constraints. The correctness and well-formedness of business process models viewed from the perspective of Petri net theory is discussed in [24, 23, 26].

In this report, we introduce *anti-patterns* for process models that allow us to measure an important aspect of the quality of process models in an objective way. Anti-patterns capture typical design errors in a process model, which make the process model incorrect. Wikipedia even defines them as *"classes of commonly reinvented bad solutions to problems"*[1].

By *correctness* we mean how well the execution traces of the process model correspond to the user's expected behavior of the process under consideration. Incorrect models thus show unexpected and incorrect behaviors, which increase the economic risk associated with the models. Incorrect models often also exhibit a non-systematic design, their functionality is usually inadequate and they lead to reliability problems in the implementing software. Needless to say that usability, clarity, comparability, and maintainability are also affected. Thus, detecting and correcting anti-patterns in process models is an important prerequisite to improve the quality of process models and for reducing their associated economic risk.

This report captures the lessons learned from reviewing hundreds of process models created in IBM WebSphere Business Modeler (WBM) and other modeling tools such as Aris [21], Adonis [22], or MID Innovator [6] between 2004 and 2006. The models resulted from real-world projects across various industries such as banking, insurance, retail, the pharmaceutical industry, and telecommunications. When reviewing draft versions of these models, we noticed recurring modeling errors that we abstracted into anti-patterns. We describe each anti-pattern in this report, explain why the model fragment in this anti-pattern is wrong and show a corrected model. A recommendation at the end of each section summarizes the main insights in a compact form. For the presentation in this report, we have redrawn and anonymized all models in WBM, i.e., only abstract names are shown and no information about the origin of our examples nor the modeling tool originally used is provided.[2] In many cases, we can also explain why users unknowingly modeled their processes incorrectly and we point to support in the WBM tool that can help finding these errors.

[2]We want to thank all colleagues who have sent their models to us for supporting this work.

This report addresses users with some experience in business process modeling, in particular in modeling with IBM WebSphere Business Modeler. It assumes that the reader is familiar with the basics of WBM as taught in the official product tutorials or courses. Most of our anti-patterns are completely independent of the WBM tool and have correspondences in other modeling languages for behavioral models such as UML2 Activity Diagrams (UML2-AD) [16], Event-Driven Process Chains (EPC) [20], or the recent Business Process Modeling Notation (BPMN) [3].

The report is organized as follows: In Section 2, we introduce the main modeling elements for business process modeling in WBM, draw relationships to corresponding elements in UML2-AD, EPC, and BPMN, and provide the background for the subsequent sections addressing the various modeling scenarios. Section 3 discusses the branching and joining of parallel and alternative flows, while we focus on the modeling of iterative behaviors in Section 4. These two sections are also the most interesting sections for users of other modeling notations. Section 5 focuses on challenges when modeling data flow. In Section 6, we address problems around the modeling of events and triggers. Section 7 addresses the problem of how to correctly describe the termination of a process. In Section 8, we discuss hierarchical process models that consist of nested subprocesses. Section 9 concludes with a summary of our findings.

2 Background

This section reviews the main modeling elements for business process modeling in WebSphere Business Modeler and draws relationships to corresponding elements in UML2 Activity Diagrams, Event-Driven Process Chains, and the recent Business Process Modeling Notation. We then address the variability in the WBM modeling language and introduce two forms of process models: models using *gateway form* and models using *activity form*, which we use as a basis to describe solutions to modeling challenges occurring in the various modeling scenarios. We also introduce our notation for anti-patterns and patterns.

2.1 Basic Modeling Elements for Control Flow

We begin by discussing the main elements that are available for the modeling of business processes in WBM and draw relationships to other popular modeling approaches. Figure 2.1 shows a sample business process model with the main activities of the process and the modeling constructs to describe control flow.

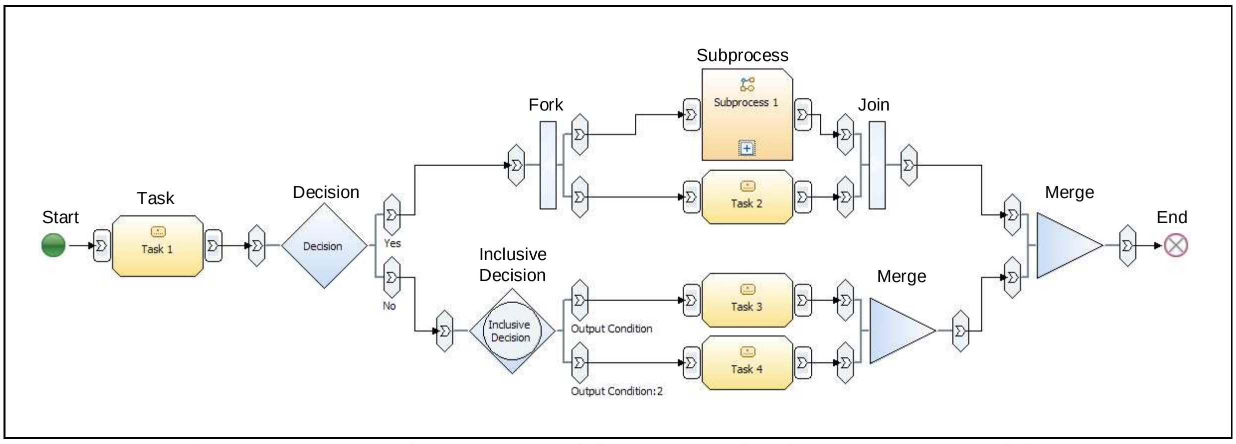


Figure 2.1: A sample process model in IBM WebSphere Business Modeler.

The process begins with a *start* node depicted as a green bullet, which is connected to *Task 1*, symbolized by a yellow rectangle with rounded corners. A task in WBM is a not further refinable (atomic) activity of the process. *Task 1* is followed by a *decision* leading to two subsequent branches. When the process executes, exactly one of the decision branches is chosen, i.e., we have an exclusive choice modeled here. The upper *Yes* branch leads to a *fork*, which captures a parallel branching in the process flow. All branches following a fork get executed, i.e., *Subprocess 1* and *Task 2* in this model. A subprocess in WBM can be further refined by another process model. Subprocesses and tasks are the two kinds of *activities* that are available in WBM. The parallel branches are joined again in a *join*, which ends the parallel execution. The lower *No* branch of the decision leads to an *inclusive decision*. An inclusive decision models the *n-out-of-m* choice from the well-known workflow patterns [25]. This means that one or more subsequent branches can be chosen for execution. In our case, either *Task 3* alone, or *Task 4* alone, or both *Tasks 3* and *4*, can execute. For the first two choices, a single sequential execution path results, while the choice of *Task 3* and *Task 4* results in both branches executing in parallel. The inclusive branches are merged again by a *merge*. The join and merge elements connect to another *merge*, which matches the initial exclusive decision. An *end* node terminates the process model. The simulation capabilities of WBM can be used to visualize and analyze the possible execution traces of a process, which show how control branches in a decision and how it leads to parallel executions after a fork.

Figure 2.2 shows the same process modeled as a UML2 Activity Diagram (UML2-AD) [16] using IBM Rational Software Architect [9].

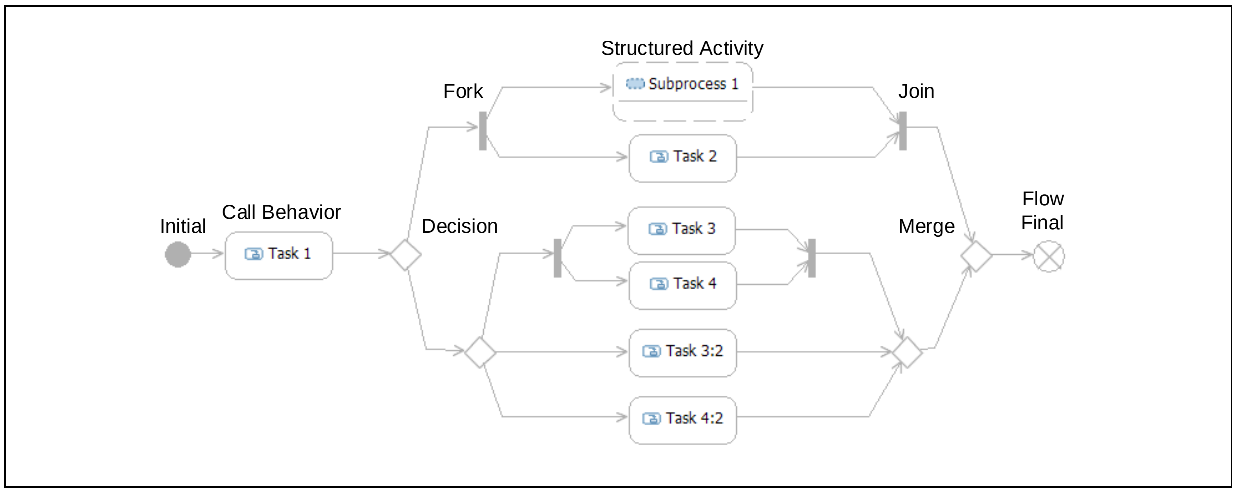


Figure 2.2: The same sample process modeled as a UML2 Activity Diagram.

Seamless integration between WBM and Rational Software Architect enables software architects to visualize business process models in UML2 notation using the *UML 2.0 Profile for Business Modeling* [9]. This profile provides interchange semantics between business process models and existing or new UML2 models. Based on this semantics, the start node in WBM corresponds to the *initial* node in UML2-AD, while the end node corresponds to the *flow final* node. A task corresponds to a *call behavior*, while *Subprocess 1* is captured as a *structured activity*, which can be further refined. Activity diagrams modeled directly by a user often also model an *action* instead of a *call behavior* and replace the *structured activity* by a *call behavior*. Fork, join, decision, and merge are identical in both modeling approaches. The inclusive decision does not exist in UML2-AD and it is mapped to the UML2 decision, which is exclusive. Thus, we explicitly added the situation to the model when *Task 3* and *Task 4* execute in parallel. We used a fork and join and placed additional copies of the tasks between them.

Figure 2.3 shows our sample process modeled in the tool Maestro [19], which implements the Business Process Modeling Notation (BPMN), a recent standard notation by the OMG [3].

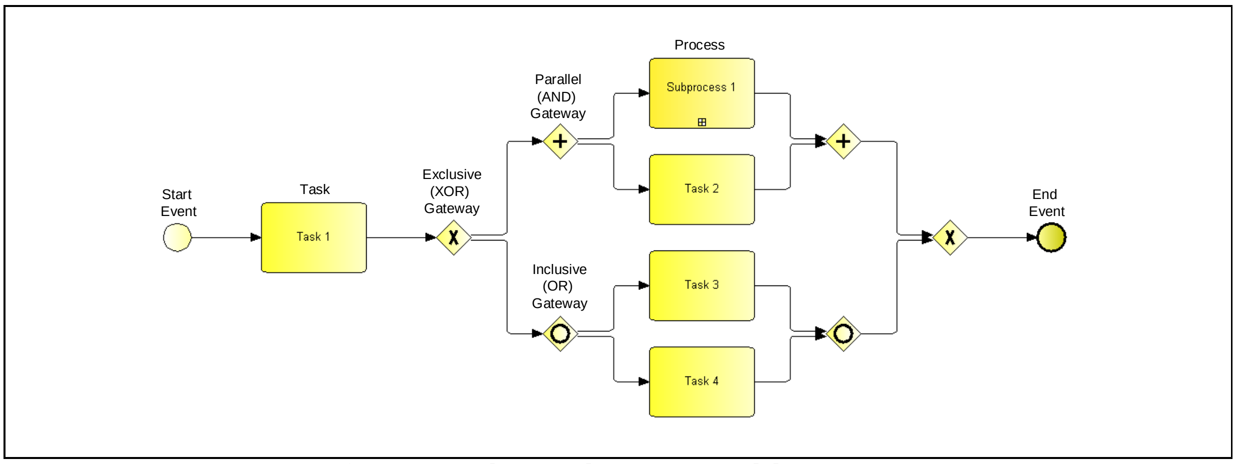


Figure 2.3: The sample process model in BPMN.

Similar to the previous modeling approaches, BPMN distinguishes *tasks* and *subprocesses* in the diagram, the latter being indicated by a ’+’ symbol in the rounded-corner rectangle. The process beginning is indicated with a *start event*, while the process end is marked with an *end event*. The control-flow behavior of a process is modeled using *gateways*. Alternative branching is modeled with the *exclusive (XOR) gateway* visualized as a diamond, optionally marked with an ’X’. Inclusive branching is modeled with the *inclusive (OR) gateway* and visualized with a diamond containing a circle. Parallel branching is modeled with the *parallel (AND) gateway* and visualized with a diamond containing a ’+’. BPMN does not provide any separate modeling elements to join or merge several branches in a process model. Instead, it uses the same symbols for branching and joining or merging of flows.

Finally, we modeled the sample process as an Event-Driven Process Chain (EPC) using Aris Business Architect [21]. In contrast to the previous approaches, EPC models are more commonly drawn in a top-down manner.

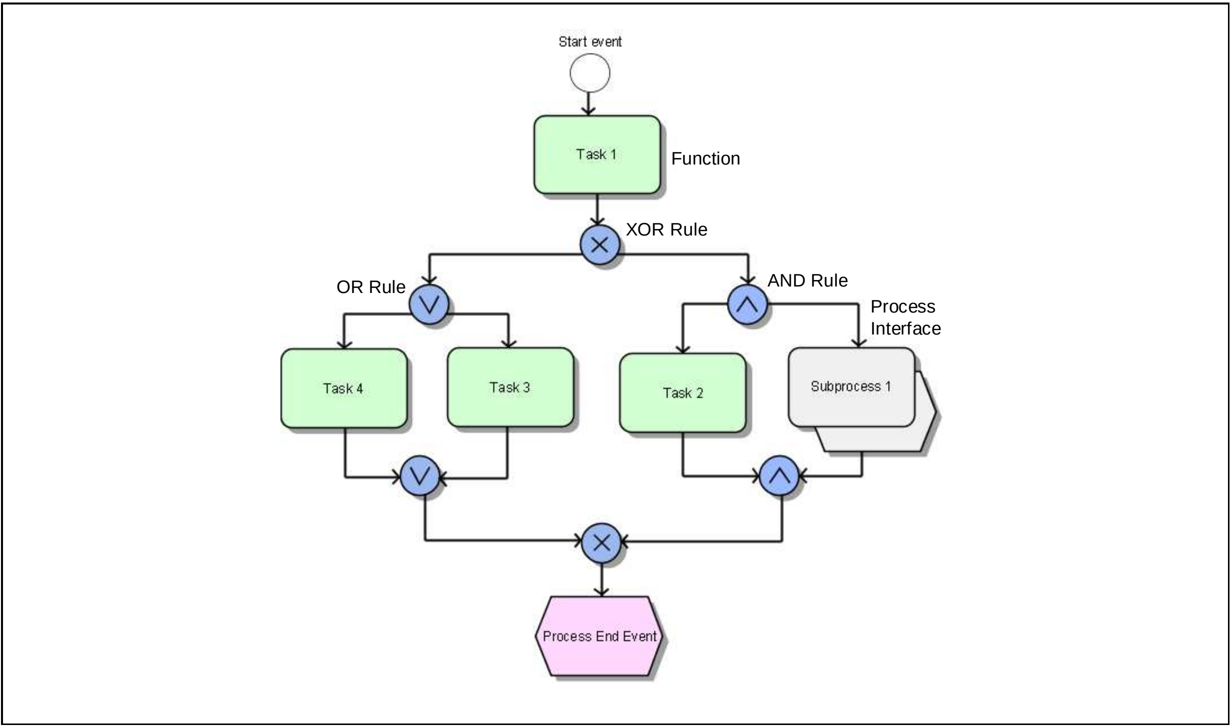


Figure 2.4: The control flow of the sample process modeled as an Event-Driven Process Chain.

The process begins with a *Start event* and ends with an event, which we named *Process End Event*. EPCs do not distinguish specific types of end events and let the user decide how to end the process. Tasks are represented as *functions*, while a subprocess is captured with a *process interface*, which is a function that can be further refined by another EPC model. We did not introduce any additional events into the model following each function. Although this would be common for the EPC approach, events are represented quite differently in the other approaches. The fork corresponds to the *AND Rule*, the inclusive decision corresponds to the *OR Rule*, and the decision corresponds to the *XOR Rule*. Similar to BPMN, EPCs use the same rule symbols to open and close branches in the process flow.

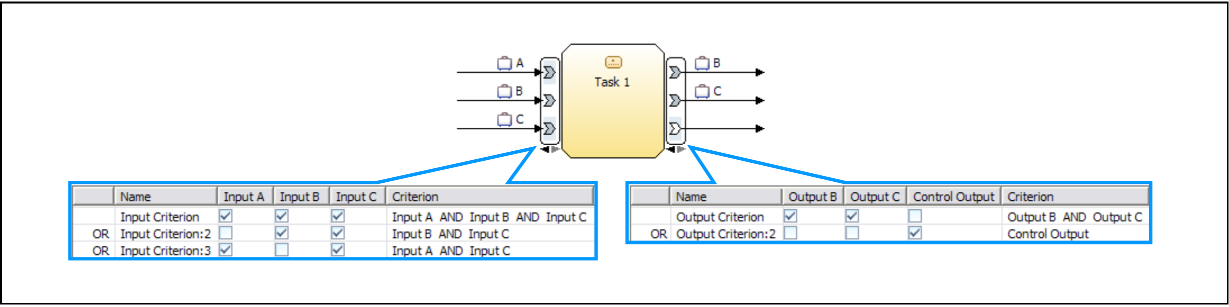
We can see that although the graphical notations differ, the main elements for modeling control flow are very similar across the different modeling approaches. Therefore, many of the following anti-patterns are also useful for users of other modeling approaches. In the following sections we focus on IBM WebSphere Business Modeler.

2.2 Gateway Form vs. Activity Form of Process Models

In the previous section, we modeled the same process using different modeling approaches. Each of these models made use of *control nodes* (WBM and UML2-AD), *rules* (EPC), or *gateways* (BPMN) to capture the control flow. Since the BPMN standard uses the term *gateways*, we call them gateways throughout this report when we refer to the decision, fork, merge, and join modeling elements in WBM.

In addition to gateways, several of these modeling languages—we only look at WBM, though—offer an alternative approach to model control and data flow. This modeling alternative is based on the inputs and outputs of the activities, i.e., of the tasks and subprocesses. The inputs and outputs can be pure control-flow links as in the models in the previous section, or can be data flow links as shown in Figure 2.5. In WBM, a control flow is changed into a data flow by associating a so-called *business item* with the flow. A business item captures a type of information or a data object that flows through the process. Users can freely choose the names of these business items and further refine their description by adding attributes. The type of business item cannot change when it is sent from one task to another, i.e., the output, the input, and the connection between them always have the same business item attached.

Figure 2.5 shows *Task 1* with three different inputs of type A, B, and C, and three different outputs: one control-flow output (white arrow) and two data-flow outputs of type B and C (gray arrow).



	Name	Input A	Input B	Input C	Criterion
	Input Criterion	✓	✓	✓	Input A AND Input B AND Input C
OR	Input Criterion:2		✓	✓	Input B AND Input C
OR	Input Criterion:3	✓		✓	Input A AND Input C

	Name	Output B	Output C	Control Output	Criterion
	Output Criterion	✓	✓		Output B AND Output C
OR	Output Criterion:2			✓	Control Output

Figure 2.5: Input and output criteria of a task.

For the inputs, three different input criteria are defined, while for the outputs, two different output criteria are defined. The criteria specify the alternative input and output sets of *Task 1*. It can execute if it either receives all three inputs at once, i.e., A, B, C or if it receives only B combined with C or only A combined with C. As output, it either provides B combined with C or only control output. Similar to gateways, these criteria thus describe branching and joining behavior.

It is important to work in the *advanced editing mode* of WBM in order to correctly define the required input and output criteria. This editing mode provides *input logic* and *output logic* tabs in the *attributes view*. The blue boxed screenshots in Figure 2.5 are extracted from these tabs. In the diagram, the gray and black backward/forward arrows below the inputs and outputs of *Task 1* indicate that several input and output criteria are defined for the task. If only one input and output criterion is defined, no arrows are shown and we usually do not show the blue boxed tabs in our figures. The basic editing mode hides all these details, including the arrows.[3]

An activity can execute if it receives all required inputs of at least one input criterion, otherwise it waits for these inputs to arrive. When an activity executes, it produces all outputs of exactly one output criterion. The output criterion it chooses can depend on which input criterion activated the activity. This dependency can be modeled in WBM by associating an output criterion with an input criterion. If no associations are modeled explicitly, the activity can produce one of the output criteria in a nondeterministic way.

We say that a process model is in *gateway form* when it is only using gateways (called control nodes or rules in other modeling languages), but only has a single input and output criterion for a task or subprocess. A process model that makes use of several input and output criteria in the activities, but does not use gateways is denoted as being in *activity form*.

Figure 2.6 shows a process model in gateway form. Gateway form makes the branching and joining points in the model more explicit. We can also more easily assign explicit decision conditions to a decision node by editing its output logic.

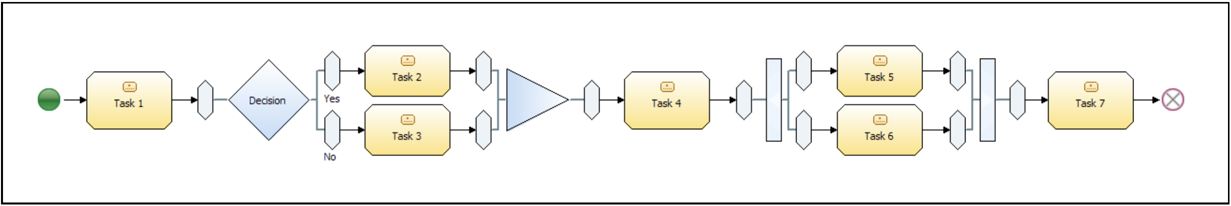
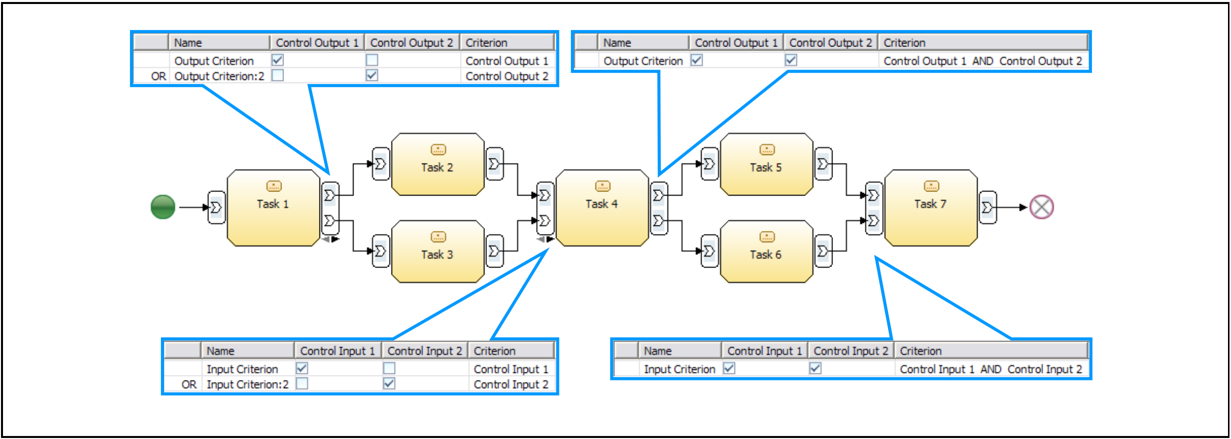


Figure 2.6: A process model showing control flow using gateway form.

The same control flow is captured in Figure 2.7 using activity form. In activity form, the decision conditions must be captured as postconditions of the output criteria. Activity form has the advantage that it makes the models more compact and reduces the modeling elements to only the functional activities in the process. However, it makes capturing the flow logic more complex, because using and defining input/output criteria is more difficult than using gateways.



	Name	Control Output 1	Control Output 2	Criterion
	Output Criterion	✓	☐	Control Output 1
OR	Output Criterion:2	☐	✓	Control Output 2

	Name	Control Output 1	Control Output 2	Criterion
	Output Criterion	✓	✓	Control Output 1 AND Control Output 2

	Name	Control Input 1	Control Input 2	Criterion
	Input Criterion	✓	☐	Control Input 1
OR	Input Criterion:2	☐	✓	Control Input 2

	Name	Control Input 1	Control Input 2	Criterion
	Input Criterion	✓	✓	Control Input 1 AND Control Input 2

Figure 2.7: The same process modeled using activity form.

In Figure 2.7, *Task 1* acts as a decision, while *Task 4* acts as a combined merge and fork. Decision and merge are always mapped to several input and output criteria—one for each branch, while fork and

[3]Note that by default each task or subprocess has at least one input and at least one output criterion, because any input must belong to at least one input criterion and any output must belong to at least one output criterion.

join correspond to a single output and input criterion, respectively.

For models that show control flow only, both forms can be used interchangeably and the models preserve their execution semantics. However, it is important to note that this only holds for control flow. In the case of data flow, there are important semantic differences that we discuss in detail in Section 5.

Recommendations 1 *Try to adopt one of the two possible forms for your models in order to reduce the number of used model elements. This leads to a more homogeneous modeling style. Gateway form using decision, fork, merge, and joins is often more readable for models that only model control flow. Activity form using input and output criteria, but no gateways, is more suitable for simplifying visualization of the process model by hiding the branching logic in the input and output criteria. Models that use activity form should be worked on in advanced editing mode in WBM to see the details of the branching flows, which otherwise remain hidden in the basic editing mode. Try to minimize mixing of both forms in a single diagram. While a mixed approach may be required for data-flow models, it is never necessary for control-flow models.*

2.3 Notation for Patterns and Anti-patterns

When describing the various modeling scenarios, we begin with an illustrative example exhibiting the error, which we then generalize into an anti-pattern. We also show a correction to the anti-pattern in the form of a pattern. To describe the patterns and anti-patterns we use process fragments as shown in Figure 2.8.

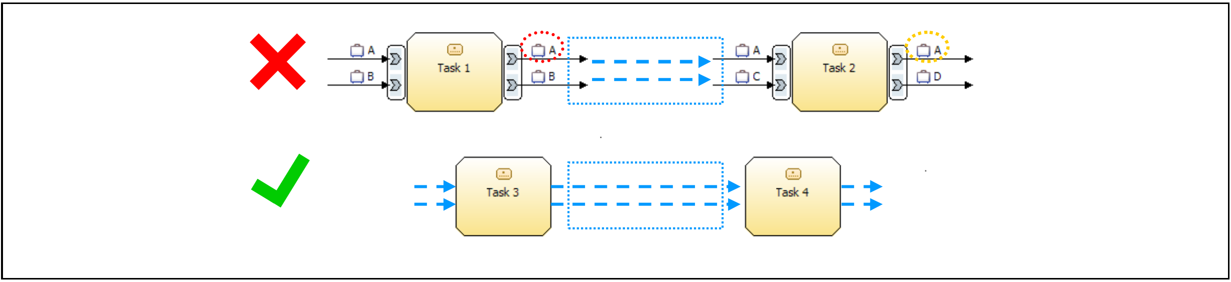


Figure 2.8: Process fragment notation.

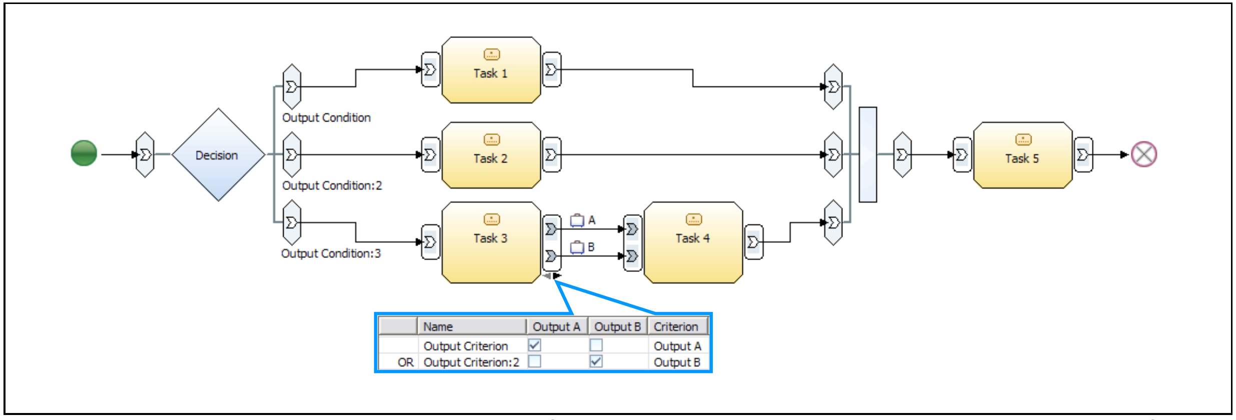
This figure shows two forms of notation that we are using. If the data or control flow in and out of activities is relevant to understand the cause of the modeling error, we use the notation in the upper half of the figure. In this process fragment, *Task 1* receives inputs A and B and provides outputs A and B, while *Task 2* receives inputs A and C and provides outputs A and D. *Task 1* precedes *Task 2* in the flow, but between both tasks, an arbitrary process fragment can be modeled. This arbitrary process fragment is denoted with the dotted box. If data attached to the flow is irrelevant, we use blue arrows to just denote the flow direction. An example of this notation is shown in the lower half of the figure involving *Task 3* and *Task 4*. Errors in a specific process model can be found by comparing a fragment of the process model to the process fragment shown in an anti-pattern. The provided patterns can be used to correct the matching process fragment. Anti-patterns are always marked with a big red cross in the upper left corner, whereas patterns are marked with a green check-off sign. In addition, we use red dotted circles to highlight a usage of modeling elements that is causing the error and orange dotted circles to highlight a usage of modeling elements that is not recommended.

3 Scenario I: Modeling Branching Behavior

Many business process models need to show how control and data flows branch and join within the process. WBM offers the gateways *decision*, *inclusive decision*, and *fork* to describe alternative and parallel (concurrent) branching and the gateways *merge* and *join* to describe alternative and parallel joining in a process. In principle, these gateways can be freely combined in a process model. However, some of these combinations lead to execution problems when the process model is simulated. In the following, we therefore systematically review the various pairings of gateways.

3.1 Deadlocks through Decision-Join Pairs

Figure 3.1 shows two typical forms of decision-join pairing that we found frequently in process models. Immediately following the start node, a decision was used to represent three alternative branches in the process flow that eventually lead to a join preceding *Task 5*. This structure leads to a *deadlock* in the process, because the decision only emits an output on one of its branches, while the join waits for input on all of its branches. This input can never come and thus the join waits forever for the missing input and does not execute—an obvious deadlock situation. A deadlock is always a modeling error: some of the desired process behaviors are missing due to non-executable activities in the model. This incorrect behavior can be easily detected using the simulation capabilities of WBM, which would show that *Task 5* never executes.



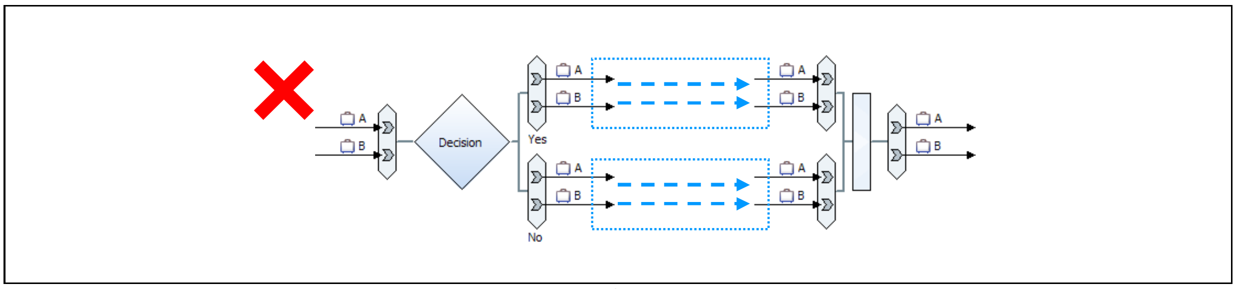
	Name	Output A	Output B	Criterion
	Output Criterion	✓	☐	Output A
OR	Output Criterion:2	☐	✓	Output B

Figure 3.1: Deadlocks caused by a decision followed by a join or by an inappropriate pairing of input and output criteria in activities.

A second deadlock occurs between *Task 3* and *Task 4* in the example. *Task 3* produces A or B as two alternative outputs in two disjoint output criteria, i.e., it behaves like an exclusive decision. *Task 4* requires both A and B as inputs in a single input criterion. In any execution of the process, both inputs are never produced and thus, *Task 4* cannot execute. Again, we have a deadlock situation that can be detected during a simulation of the process when the decision selects the lower branch for execution.

The combination of a decision with a subsequent join is one of the most frequent anti-patterns that we found in business process models. Apparently, many users are not fully aware of the semantics of these gateways and of the behaviors that result from this incorrect combination. In the above example, the deadlock is relatively easy to spot. However in large examples, the branching logic can be very complicated and comprise many different activities and gateways, which makes it much harder for a user to notice these problems. It is therefore important to organize process models in a structured manner and to decide for the use of *activity form* or *gateway form* as we described them in the previous section.

Anti-pattern 3.1 shows a generalization of the incorrect decision-join pairing that results in a deadlock. The two outgoing branches of the decision with all their connections end as incoming branches in a join. Of course, the decision could also have more than two branches and a deadlock would also occur when only a subset of the branches leaving the decision is rejoined by a join. We show some data flow here to emphasize that the anti-pattern can occur in models with control flow and/or data flow and to illustrate how the connections within the single branches connect the gateways involved in the anti-pattern.

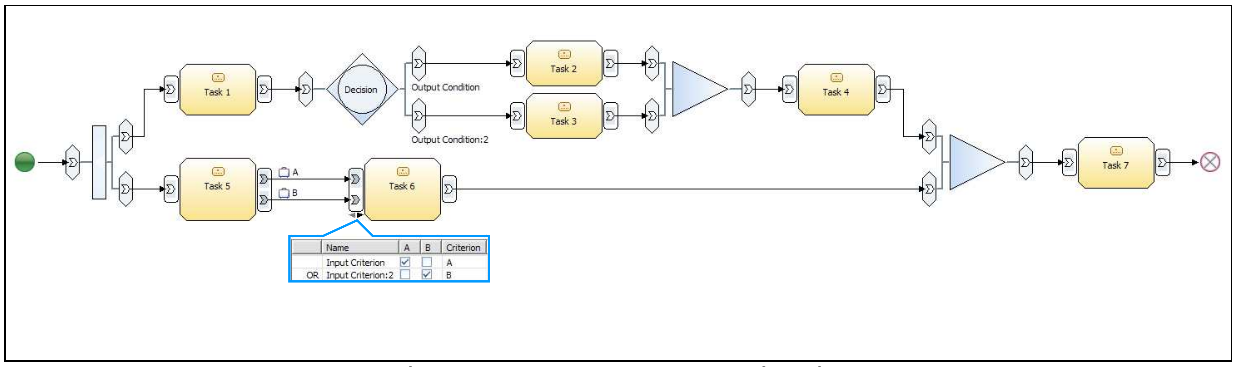


Anti-Pattern 3.1: Incorrect combination of decision and join that results in a deadlock.

A variant of this anti-pattern using *activity form* is straightforward and thus not shown.

3.2 Lack of Synchronization through Fork-Merge Pairs

The fork-merge pairing as shown in Figure 3.2 below exhibits the opposite behavior of the decision-join anti-pattern.



	Name	A	B	Criterion
	Input Criterion	✓	☐	A
OR	Input Criterion:2	☐	✓	B

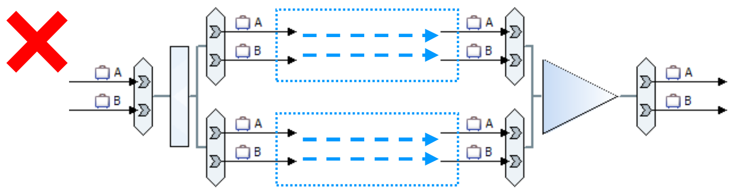
Figure 3.2: Lack of synchronization caused by a fork followed by a merge.

A fork-merge pair causes too many instances of all tasks and subprocesses following the merge to execute, because the fork emits output on all of its outgoing branches, while the merge waits for input on only one of its incoming branches. This means, for each branch of the fork, the merge executes and triggers the activities further downstream in the process model. In the workflow literature, this anti-pattern is often called *lack of synchronization* [18, 23], because for a single execution of a process fragment preceding the merge, we obtain several executions of the process fragments following the merge. Sometimes, such a behavior can be intended, but in most cases, it is unconsciously introduced into the model. A lack of synchronization can be a modeling error if more than the desired process behaviors occur, but it must not be an error unlike a deadlock.

We find three pairs that cause a lack of synchronization in Figure 3.2. The inclusive decision following *Task 1* leads to the merge preceding *Task 4*. The fork following the start node leads to the merge preceding *Task 7*. The single output criterion of *Task 5* connects to two disjoint input criteria of *Task 6*. This means, for a single execution of the process fragment comprising *Tasks 1, 2, 3, 4, 5, 6*, we obtain at least two executions of *Task 7*. For a single parallel execution of *Task 2* and *Task 3*, we obtain two executions of *Task 4*. Finally, for a single execution of *Task 5*, we obtain two executions of *Task 6*.

An inclusive decision captures an *n-out-of-m choice* [25] at an abstract level, i.e., the inclusive decision can trigger any subset of the two branches. If both branches are triggered, the inclusive decision leads to a parallel execution of *Task 2* and *Task 3* in the same way as a fork. *Task 7* following the corresponding merge would therefore execute twice—once for each triggered branch. The official recommendation in the WBM help pages is to pair an inclusive decision always with a merge. Using the merge leads to a lack of synchronization as the merge is always triggered once for each incoming branch and all tasks following the merge execute multiple times. Using a join does not solve the problem as the join always waits for all incoming connections. It therefore blocks in any simulation run in which only a subset of the branches is chosen by the inclusive decision, i.e., a deadlock occurs. [4]

Anti-pattern 3.2 captures the incorrect fork-merge pairing that is responsible for a lack of synchronization.

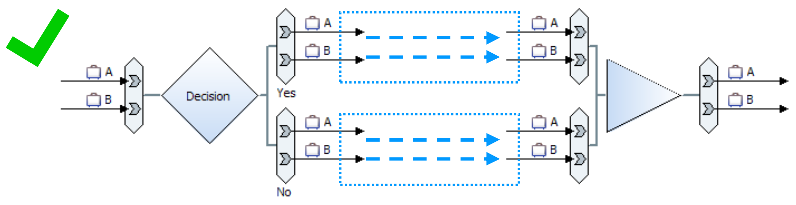


Anti-Pattern 3.2: Incorrect combination of fork and merge that results in a lack of synchronization.

Each outgoing branch of the fork with all its connections ends as an incoming branch of the merge. Again, even if only a subset of at least two branches ends in the merge, a lack of synchronization occurs. This anti-pattern is also one of the most frequent modeling errors. Spotting such combinations can be very difficult in larger process models, in particular when they occur in a more indirect way by using inclusive decisions or input and output criteria with activities.

3.3 Correct Branching through Decision-Merge and Fork-Join Pairs

The following patterns show the correct pairings of the gateways that can be used to build more well-formed process models, which are less likely to contain problems like deadlock or lack of synchronization. The two patterns represent the correct pairings of decision with merge and fork with join. Note how the two similar graphical shapes of the correctly matching gateways simplify it for the user to select the correct pairing. Pattern 3.1 presents the correct way to model alternative branching by using a matching decision-merge pair.

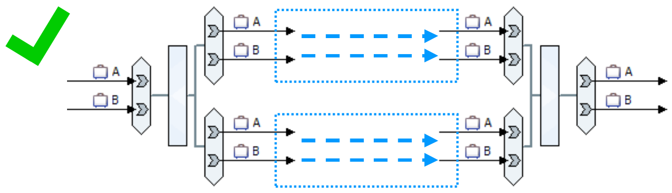


Pattern 3.1: Pairing decision with merge is the correct way to model alternative branching.

Pattern 3.2 presents the correct way to model parallel branching by using a matching fork-join pair.

Note that the decision is paired with a matching merge independent of the process fragment that is located between these two gateways. The same holds for the fork and join. There is also one matching input criterion in the merge for each output criterion in the decision, such that all the connections starting in one output criterion end in the same input criterion. Similarly, the same requirement applies to the

[4]A comprehensive treatment of an n-out-of-m choice requires techniques such as *dead-path elimination* [13], which prunes non-triggered branches of an inclusive decision and thus guarantees that the matching fork does not wait for input from any non-triggered branches.



Pattern 3.2: Pairing a fork with a join is the correct way to model parallel branching and joining.

connections between the fork and the join. All inputs of the decision and the merge, as well as of the fork and join must be connected.

We recommend to always use gateways in their corresponding pairs. This means that every decision should eventually be followed by a matching merge, which combines all the paths starting from the decision. Similarly, every fork should eventually be followed by a matching join, which combines all the paths starting from the fork. This pattern leads to a well-formed structure for the models, which avoids many modeling errors that otherwise occur when gateways are used in an arbitrary way. Sometimes, this technique can lead to many matching pairs, which add unnecessary redundancy to the process model by several merges or joins succeeding each other.

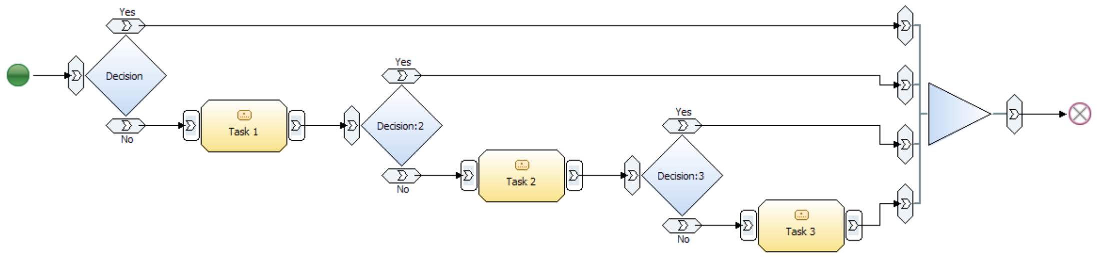


Figure 3.3: All branches correctly closed with a single matching merge.

In such a situation, closing several decisions with a single merge and closing several forks with a single join leads to a correct model. Figure 3.3 shows a typical process model that has adopted such a "short-hand" modeling notation. In this example, a single merge is used instead of having three successive merges added to the model. However, such a solution should be used with care in models that contain both types of branching, i.e., forks and decisions, to avoid suddenly closing a fork with a merge or a decision with a join.

3.4 Redundant Branches

We conclude this section with a closer look at the problem of redundant branches in a process. Figure 3.4 shows a process model with a decision of four branches that are closed by two succeeding merges.

The second and third branches both lead to *Task 2* and do not involve any activity that would occur between the decision and the first merge. It is not clear why they were distinguished from each other unless the modeler intended to separate the decision conditions. This might be required in an application that uses process monitoring. If not, this distinction appears to be redundant, because exactly the same behavior is modeled twice, i.e., the single *Task 2* is executed as a result of taking any of these two decision branches. The lowest branch represents the "do nothing" case, which can occur in a meaningful way in a decision process. In contrast to this, "do-the-same" branches may point to redundancy in the model or indicate that the modeling project is not yet complete.

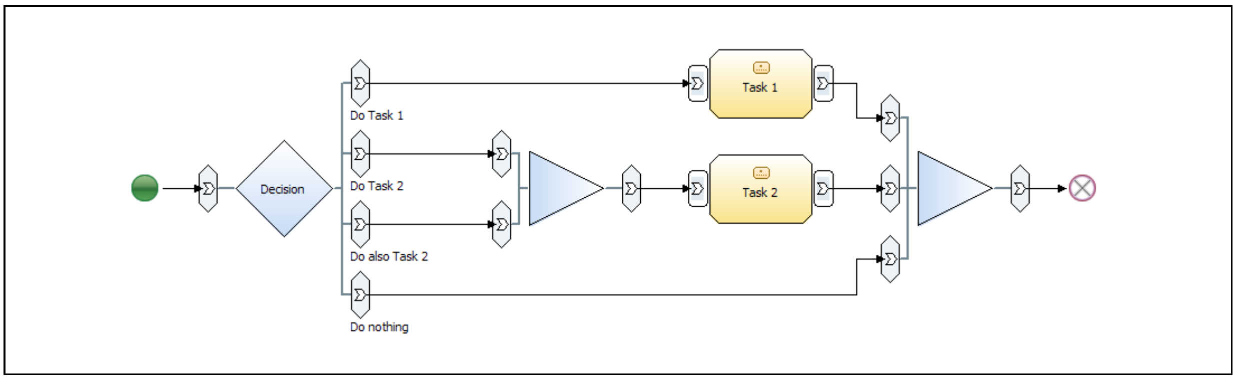


Figure 3.4: Redundant “do-the-same” branches leading to the same subsequent task execution in contrast to a “do-nothing” branch.

Recommendations 2 *Never pair a decision with a join. Never combine a fork with a merge, except when you want to introduce a lack of synchronization. Use the decision-merge and fork-join pairs always as matching pairs and make your model as well-formed as possible. When the inclusive decision is paired with a merge it can lead to a lack of synchronization when several branches execute in parallel. A safe solution is to avoid the inclusive decision and to explicitly model all possible combinations of branches using decisions and forks. If this is not possible, the inclusive decision should be used with care and the model should receive specific attention when being implemented to make sure that the control-flow logic is correctly captured in the refined IT-level model. Control or data flow connections in a process model that only connect gateways to each other without involving any activities or leading to exactly the same activity executions should be examined for redundancy.*

4 Scenario II: Modeling Cyclic Behavior

In this modeling scenario we discuss the representation of cyclic (iterative) behavior in a process model. In a realistic process, it is often necessary to repeat a previous activity based on some decision outcome. This leads to process fragments that execute several times, often until a certain exit condition is satisfied. Some process models even have to represent behaviors that execute infinitely. Cyclic behavior can be captured in WBM in four different ways, as shown in Figure 4.1.

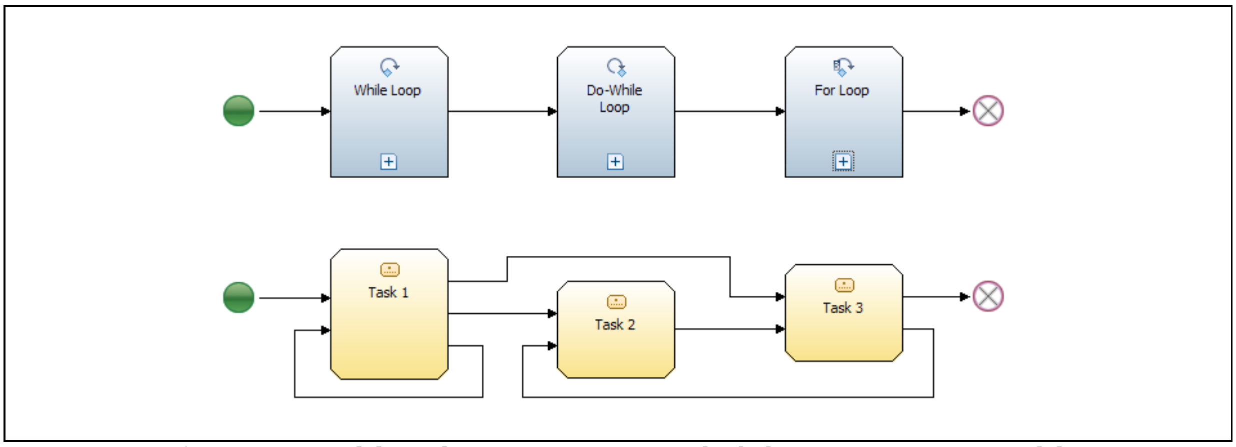


Figure 4.1: Modeling elements to capture cyclic behavior in a process model.

The upper half of this figure shows the three modeling elements to represent a *while loop*, a *do-while loop*, and a *for loop*. The repeated process fragment must be placed inside these loop modeling elements.

We consider these elements as very technical and not really as suited to business users. Instead, business users usually draw backward connections that direct a flow back to an activity further upstream as it is shown in the lower half of the figure. This process model contains one backward connection that starts at *Task 1* and leads back to the same task causing it to self-loop, and a second backward connection that starts at *Task 3* and leads back to *Task 2* causing a cyclic process fragment comprising these two tasks. These backward connections are very intuitive to use and often lead to overlapping cycles in the model, e.g., the cycle from *Task 1* to *Task 3* and back to *Task 2* overlaps with the cycle involving *Task 2* and *Task 3* only. [5]

Typical errors occur when such unstructured cycles are added to the process model. The noticeable problems relate to the correct branching and merging of the cyclic flows. Interestingly, we can easily describe these modeling errors by continuing to discuss gateways as in the preceding section. We can just change the order of these gateway pairs. In the preceding section, where we discussed pairs to describe alternative and parallel flows, a decision or fork always came before a merge or a join. Cycles are created by reversing this order, i.e., by combining a merge or join with a decision or fork. Only one of these combinations works out well, the other three lead to anti-patterns.

4.1 Cyclic Deadlocks through Join-Fork and Join-Decision Pairs

Figure 4.2 shows a process with three cycles.

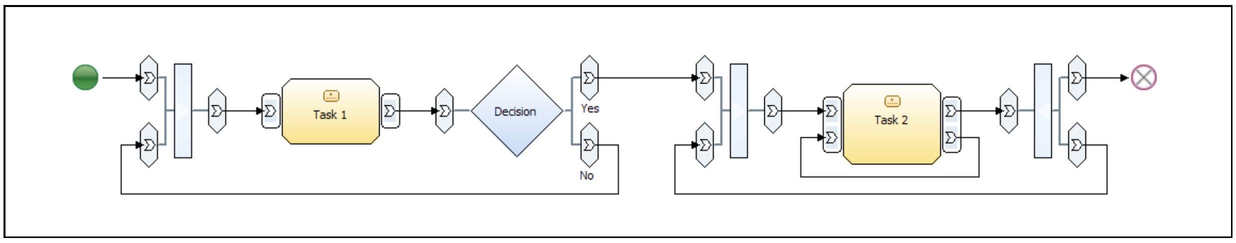


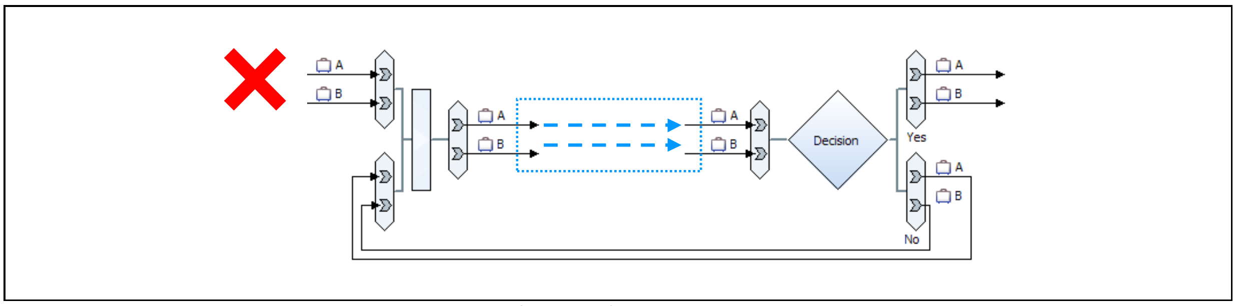
Figure 4.2: Deadlocks caused by backward connections ending in join-like modeling elements.

The first cycle involves a backward connection that starts in the decision and leads back to the join following the start node. It comprises *Task 1*. A second cycle is the self-loop of *Task 2*, which uses activity form. *Task 2* has a single input criterion, in which two connections end, acting as an implicit join. It has a single output criterion, in which two connections begin, acting as an implicit fork. As we already know from our discussion in Section 2, *Task 2* waits for all inputs in this single input criterion. However, these two inputs cannot possibly both arrive at the same time, because one of them is produced as an output of *Task 2*. The third cycle starts in the fork preceding the end node and leads back to the join following the decision. It also comprises *Task 2*. The join waits for input on <u>all</u> of its branches. However, one of its incoming branches can only receive the input <u>after</u> the join executes, because its input originates from a modeling element further downstream in the model. This cyclic dependency between the join and the decision, and the join and the fork, where the join must be executed before the decision or fork and vice versa is the reason why a deadlock occurs. These incorrectly modeled cycles are made explicit in the following two anti-patterns.

Anti-pattern 4.1 represents a join-decision pair that deadlocks. The cycle includes a join preceding a decision. At least one incoming branch of the join connects to an outgoing branch of the decision. A branch can involve several connections.

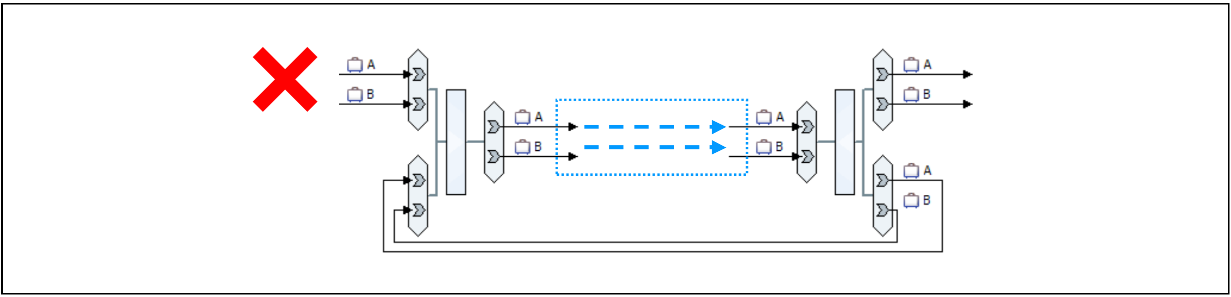
Anti-pattern 4.2 is similar to the previous anti-pattern except that it contains a fork instead of the decision. It has exactly the same kind of cyclic dependency as Anti-pattern 4.1, now between the join

[5]When exporting to BPEL, a process model in WBM must currently use only the while-loop modeling element. Unstructured cycles can often be replaced with loops by applying control-flow normalization techniques from compiler theory [11]. Plug-ins have been prototyped for WBM that implement such a normalization. Interested readers should contact the authors.



Anti-Pattern 4.1: Incorrect combination of a join followed by a decision resulting in a cyclic deadlock.

and the fork.



Anti-Pattern 4.2: An incorrect join-fork pairing that always results in a cyclic deadlock.

Variants of these anti-patterns for process models using activity form are straightforward and thus not shown. Users working in the *basic editing mode* of WBM should watch for cycles such as in Figure 4.1 where *Task 1* and *Task 2* cannot execute, because the tasks join the two incoming connections in a single input criterion, which behaves like a join. The required alternative input criteria cannot be defined in the basic editing mode, so most backward connections drawn in basic editing mode add a deadlock to the model. Deadlocking cycles can be very hard to find in a larger process model. In particular, when process models contain alternative and parallel branches combined with cycles, the risk of having modeled a deadlock is very high. In our opinion, models that combine iterative with branching behavior are the most complex models —they are not easy to draw or to understand. A well-formed nesting of cyclic and branching process fragments is a good approach to reduce the possibility of modeling errors.

4.2 Cyclic Lack of Synchronization through Merge-Fork Pairs

Lack of synchronization occurs in cyclic models when backward connections occur in branches that are executed in parallel and which are not synchronized by a join before adding the backward connection to the process model. In this situation, each of the backward connections results from the same fork or output criterion of an activity and ends in a merge or different input criterion of an activity further upstream in the model. Figure 4.3 shows an inclusive decision where two of the branches connect back to an upstream merge, i.e., each of these branches causes an independent cycle in the process as the merge executes separately for each incoming branch.

This error can lead to an explosion of uncontrolled iterations of the process—in the example for *Task 1*—which can sometimes be detected by using the WBM simulation. It requires that a simulation situation is observed where the simulation chooses more than one outgoing branch of the inclusive decision for execution and parallel cycles become visible. The upper branch of the inclusive decision connects to a third cyclic process fragment with a merge preceding a fork. The uncontrolled iteration of *Task 4* and *Task 5* becomes evident during a simulation, because for each execution of the fork with its

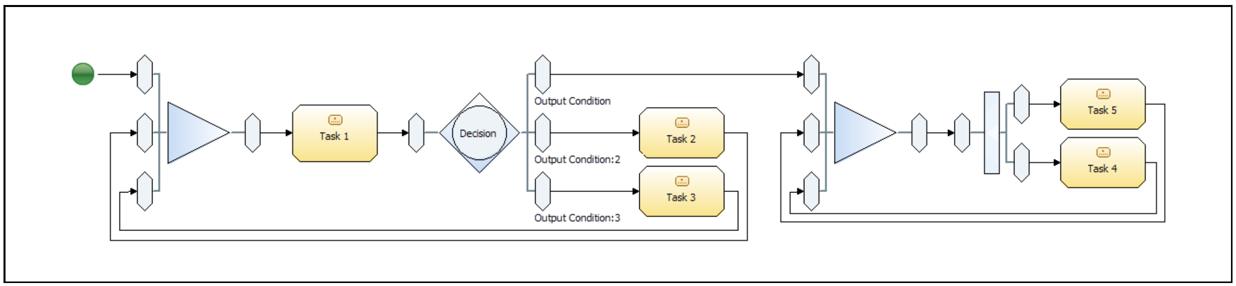
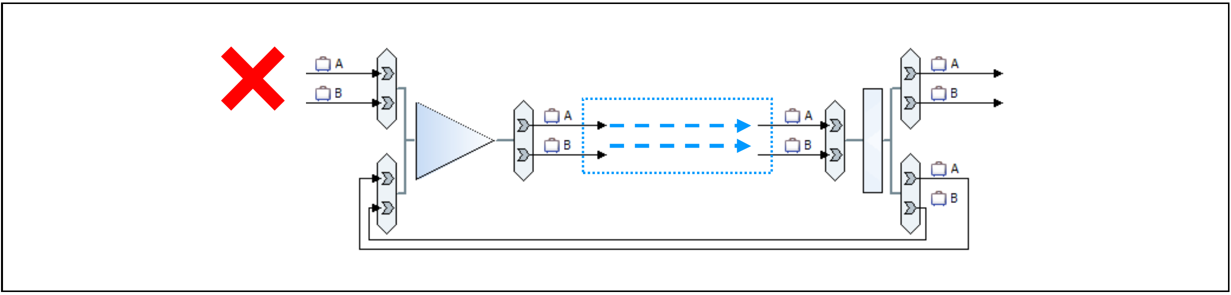


Figure 4.3: Backward connections beginning in an inclusive decision or fork cause several unsynchronized cycles when leading back to a merge without being first synchronized by a join.

two outgoing branches, the corresponding merge executes twice. The cyclic process fragment comprising *Task 4* and *Task 5* runs infinitely in parallel with the cycles involving *Tasks 1*, *2*, *3*.

Anti-Pattern 4.3 makes the lack of synchronization through unsynchronized cycles explicit by pairing a fork with a merge.

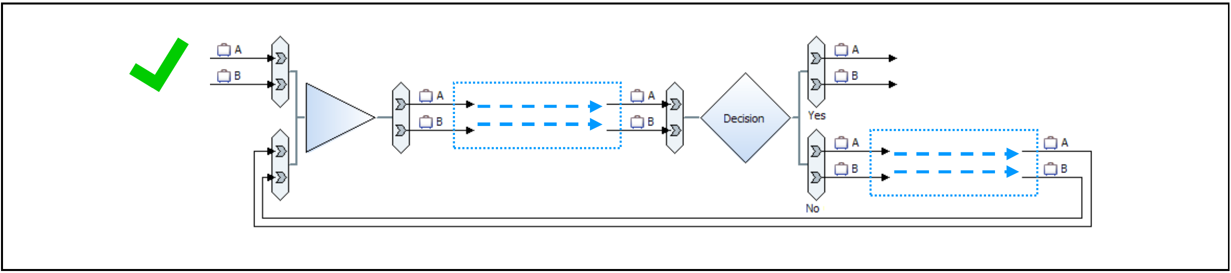


Anti-Pattern 4.3: Cyclic lack of synchronization caused by a fork connecting back to a merge.

In the first process run, it triggers the fork, which leads to two outgoing parallel branches. One branch leads to some arbitrary process fragment that is not shown. The other branch causes a cycle between the fork and the merge. When this backward connection enters the merge, the merge executes and triggers the fork again, which in turn activates its two outgoing branches. One of these branches leads to another iteration of the cycle. By replacing the fork with an inclusive decision or an activity with several outgoing connections that originate from the same output criterion, we obtain two related variants of this anti-pattern. The difference between the behavior of the fork and the inclusive decision lies in the number of unsynchronized cycles that are triggered when the fork and the inclusive decision execute. The inclusive decision only leads to a lack of synchronization if it activates more than one of its outgoing branches.

4.3 Correct Cycles with Merge-Decision Pairs

To correctly model a cycle is to use a merge followed by a decision as shown in Pattern 4.1.



Pattern 4.1: A correct way to model iterative behavior using backward connections.

In this pattern, a merge is followed by a matching decision independently of the process fragment that is located between these two gateways. At least one of the incoming branches of the merge must come from outside the cyclic process fragment, otherwise this process fragment cannot be reached from other activities. At least one of the outgoing branches of the decision must lead to a process fragment outside the cyclic process fragment, or the cycle would infinitely loop—when not intended, this causes a so-called *livelock* error. All backward connections into one branch of the merge should originate from the same branch of the decision in order to avoid implicit deadlocks or a lack of synchronization. A variant of this pattern in activity form could use activities with several disjoint input and output criteria—one criterion for each branch of the gateways.

Recommendations 3 *Cyclic processes can be correctly captured by adding backward connections to the process model that connect to an upstream activity. The backward connection should begin in a decision and lead back to a merge. Only this gateway pairing can lead to a correct cycle. Alternatively, a cycle can begin in a separate output criterion of an activity and end in a separate input criterion of this or another activity.*

5 Scenario III: Modeling Data Flow

Real-world business processes always work on data in some form. They require data, they modify and update data, and they often also derive new data by bringing various data sources together. Therefore, capturing the data flow of a process is usually an important phase in a business process modeling project. Adding this information to the process model is non-trivial and often leads to errors. In addition to containing errors, models can quickly become cluttered. Therefore, we want to focus especially on problems around the modeling of data flows in this section.

5.1 Dangling Inputs and Outputs

A phenomenon that we often observed in process models is the occurrence of *dangling* inputs and outputs, i.e., inputs and outputs of an activity or gateway that remain unconnected in the model. This phenomenon usually occurs when models are edited in the basic editing mode of WBM, which does not visualize inputs and outputs, but only shows the connecting edges between activities and gateways. Dangling inputs and outputs very often remain as residues of connections that users decide to delete or redirect. When a connection is deleted, WBM does not automatically delete the inputs and outputs that were connected, because it can still be the case that the user wants to reconnect them.

Figure 5.1 shows an example process with dangling control flow (small white arrows) and data flow (small gray arrows) inputs and outputs in a fork and a join.

Unfortunately, *dangling inputs* are often the source of simulation errors or prevent the simulation from running all, because an activity or gateway waits for some input that it can never receive. The fork and join in Figure 5.1 cannot execute due to their dangling inputs. In WBM, all branches of a gateway must currently have the same data inputs and outputs. The editor enforces this requirement, i.e., whenever the user adds an input or output to a gateway in some branch, WBM automatically adds an input or output to all the other branches as well. It is not possible to have different business items associated with different branches. Thus, if only some of the inputs and outputs are connected, this must immediately lead to dangling inputs and outputs that are not directly visible in the basic editing mode as Figure 5.2 shows. Only experienced users would notice the larger shapes for the input and output branches in the gateways that hint at the problem.

Dangling *outputs* are less severe than dangling *inputs*, because they usually do not prevent a process model from correctly executing. However, dangling data outputs show that a task or subprocess produced some data, or data was involved in the branching modeled in some gateway, but this data is not used

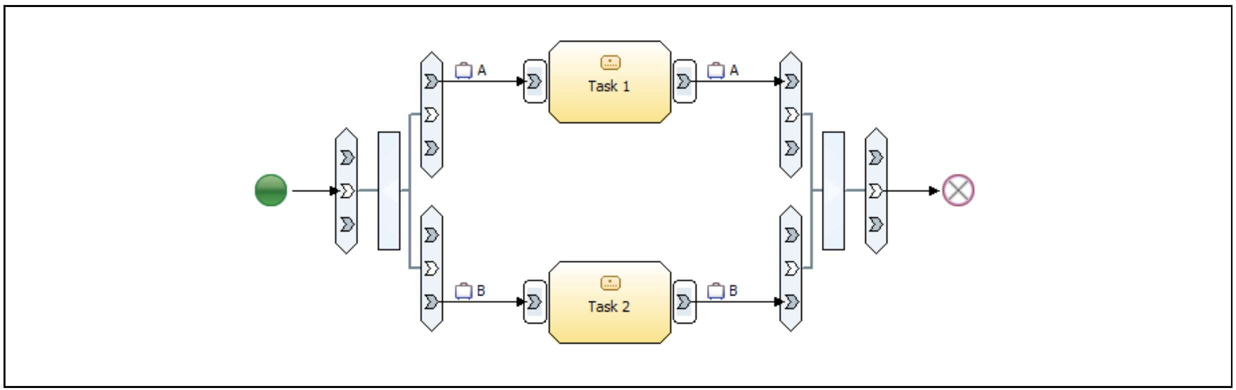


Figure 5.1: Dangling inputs and outputs in a process model are only visible in the advanced editing mode of WBM.

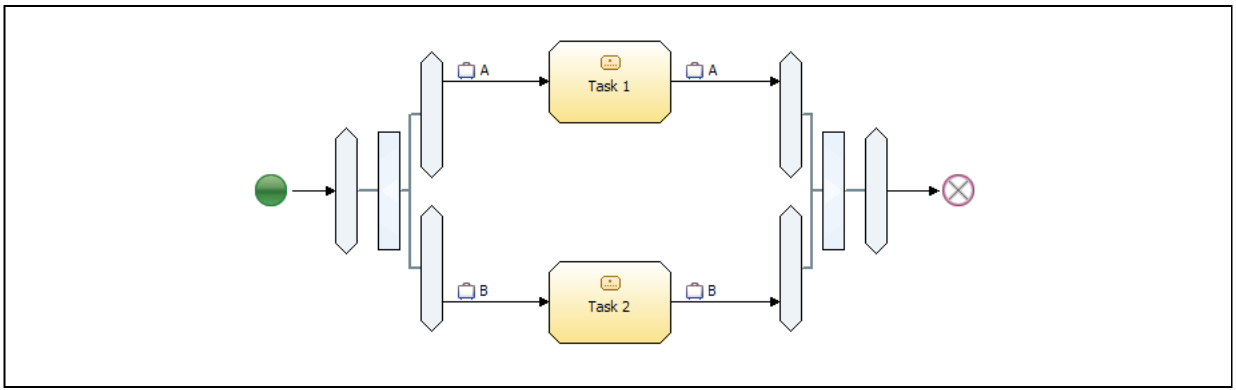
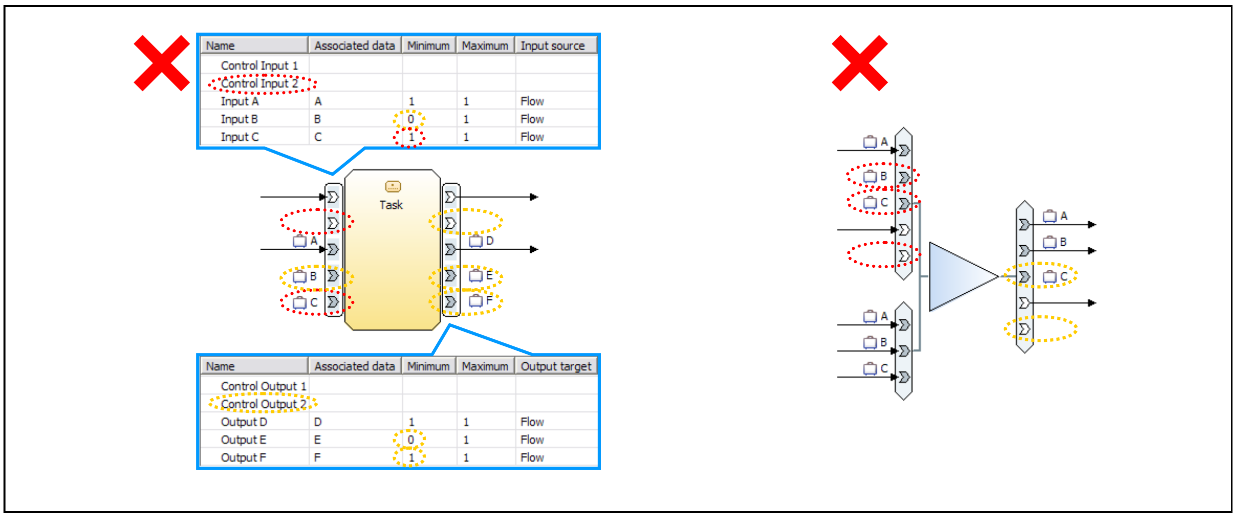


Figure 5.2: Dangling inputs and outputs are not visible in the basic editing mode of WBM.

anywhere in the process. Later in this section we discuss how to deal with such dangling outputs, because they sometimes look like an unfinished model.

Anti-pattern 5.1 summarizes the dangling inputs that must be avoided (circled in red) and dangling outputs that should be avoided (circled in orange).

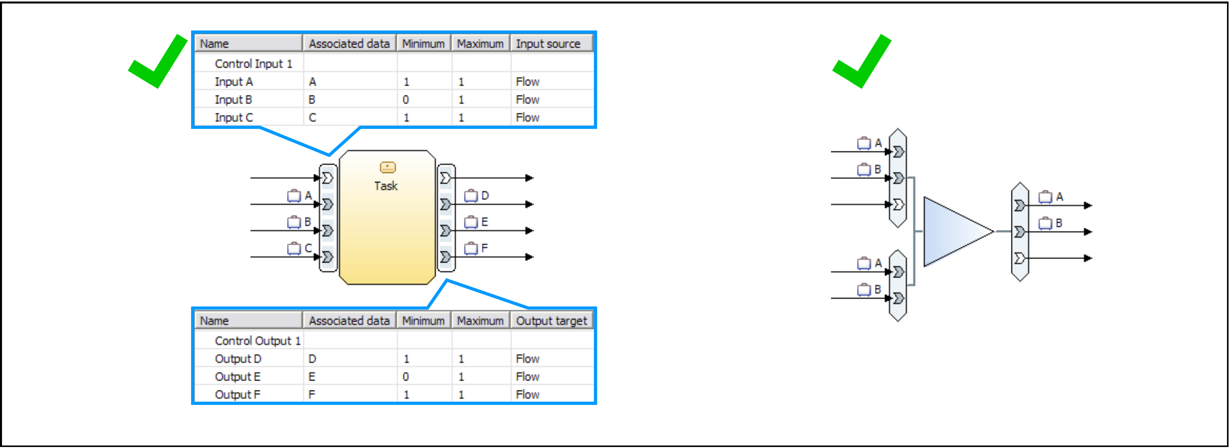


Name	Associated data	Minimum	Maximum	Input source
Control Input 1				
Control Input 2				
Input A	A	1	1	Flow
Input B	B	0	1	Flow
Input C	C	1	1	Flow

Name	Associated data	Minimum	Maximum	Output target
Control Output 1				
Control Output 2				
Output D	D	1	1	Flow
Output E	E	0	1	Flow
Output F	F	1	1	Flow

Anti-Pattern 5.1: Dangling inputs that must be avoided (circled in red) and inputs and outputs that should be avoided (circled in orange).

Dangling inputs cause deadlocks if the input is a control input of an activity or a gateway, a data input of a gateway, or a *required* data input of an activity. A data input is required if its minimum multiplicity is greater than 0. A minimum multiplicity of 0 means that the input or output is optional. The screenshots of the input logic and output logic tabs in Anti-pattern 5.1 and in Pattern 5.1 show the defined minimum and maximum of the inputs and outputs. Pattern 5.1 summarizes how to correctly model the inputs and outputs of gateways and activities with a single input and output criterion.



Name	Associated data	Minimum	Maximum	Input source
Control Input 1				
Input A	A	1	1	Flow
Input B	B	0	1	Flow
Input C	C	1	1	Flow

Name	Associated data	Minimum	Maximum	Output target
Control Output 1				
Output D	D	1	1	Flow
Output E	E	0	1	Flow
Output F	F	1	1	Flow

Pattern 5.1: Correctly defining and connecting data inputs and outputs.

All required control inputs and data inputs must be connected in order to avoid a deadlock. Control outputs and required data outputs should be connected. It is recommended, but not required, to connect all the optional data inputs and outputs, i.e., those that have their minimum multiplicity set to 0. Non-required control inputs and outputs should be deleted from an activity. Removing data inputs and outputs changes the data requirements, so it is not always possible to remove them. For example, the business item C is only connected as an input in the lower branch of the merge on the left-hand side of Anti-pattern 5.1. Since it not connected as an output, i.e., it is not used by any subsequent activity, it was removed from the merge in Pattern 5.1.

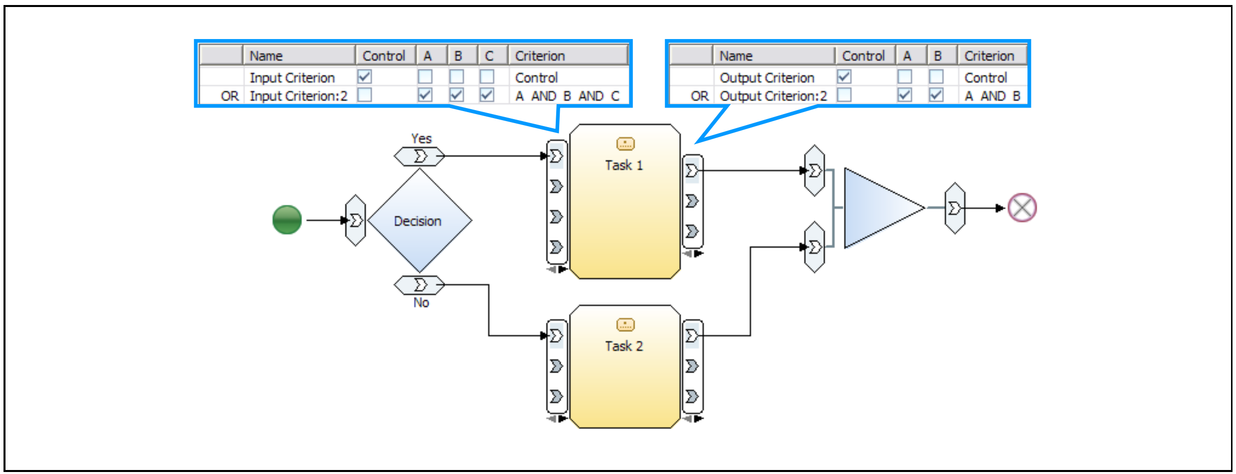
Recommendations 4 *Working in the basic editing mode speeds up editing models, especially when creating models from scratch. However, before ending a modeling session it is important to switch to the advanced editing mode in order to find and clean up dangling inputs and outputs.*

5.2 Reducing Clutter in Data-Flow Models

Can there be cases where dangling inputs and outputs make sense? Yes, because they can be a valid means to reduce clutter in models by showing only some selected flows. We see two possible modeling approaches where dangling inputs and outputs can be used safely without affecting the ability to execute the process model.[6] Figure 5.3 shows the first approach, which uses connected control flow and puts all data inputs and outputs into a separate input and output criterion.

The intuitive idea behind this approach is that data is not flowing through the process, but tasks and subprocesses access data from data sources shared among the activities. The specified control flow determines the order in which the data is accessed. The separation of the connected control flow from the disconnected data inputs and outputs into separate input and output criteria ensures that the process can correctly execute along the connected control flow. Furthermore, all gateways only involve control flow, but no data. Data inputs and outputs are visible when the model is viewed in advanced editing mode: in basic mode, only the control flow is visible.

[6]In version 6 of WebSphere Business Modeler, BPEL code can also be correctly generated from such models.

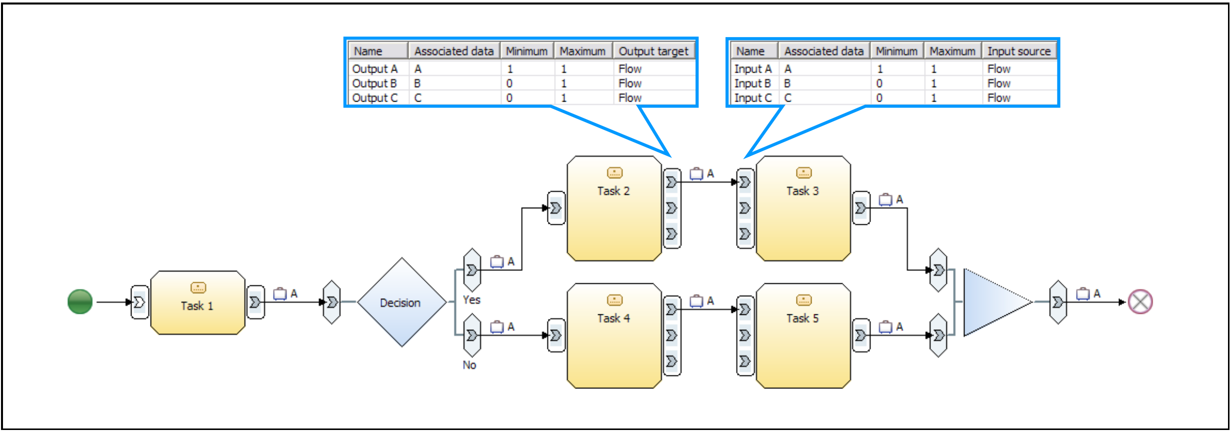


	Name	Control	A	B	C	Criterion
	Input Criterion	✓	☐	☐	☐	Control
OR	Input Criterion:2	☐	✓	✓	✓	A AND B AND C

	Name	Control	A	B	Criterion
	Output Criterion	✓	☐	☐	Control
OR	Output Criterion:2	☐	✓	✓	A AND B

Figure 5.3: Dangling inputs in a separate input criterion preserve process execution.

Figure 5.4 shows the second approach, which uses only a single input and output criterion, but sets the minimum multiplicity of all disconnected inputs and outputs to 0.



Name	Associated data	Minimum	Maximum	Output target
Output A	A	1	1	Flow
Output B	B	0	1	Flow
Output C	C	0	1	Flow

Name	Associated data	Minimum	Maximum	Input source
Input A	A	1	1	Flow
Input B	B	0	1	Flow
Input C	C	0	1	Flow

Figure 5.4: Dangling inputs with minimum multiplicity set to 0 represent optional inputs, which can remain unconnected in executable process models.

This approach focuses on only showing selected data flows in a process model. No additional control-flow connections should occur when a data-flow connection already exists between activities and gateways. Furthermore, only a single business item traverses a gateway in order to keep the data flow simple. An advantage of this presentation is that gateways involve data flow and thus the data-based branching decisions can be captured. This was not possible in the first approach, because it only showed the control flow. Using only a single input and output criterion makes editing the model easier. A slight disadvantage of the approach is that it can lead to a mixture of control flow with data flow involving different business items, which can make the models harder to understand than models that only show control flow together with disconnected inputs and outputs. To further complicate matters, different stakeholders in the modeling project sometimes disagree on which data is the most relevant and which data can be considered as optional.

Recommendations 5 *Clutter in complex data flow models can be reduced by showing disconnected data inputs and outputs. The disconnected inputs must be either put into a separate input criterion or marked as optional by setting their minimum multiplicity to 0 to allow the process to execute.*

5.3 Multiple Connections between Activities

Complex control and data flows easily lead to *multiple connections* in process models, which are another source of cluttered models. Multiple connections, for short *multi-connections* all start in the same activity or gateway and all end in one other activity or gateway. These connections lead to unnecessary redundancy if the multi-connections only involve control flow. If the multi-connections are associated with the same business item, they can easily lead to modeling errors. Figure 5.5 shows an illustrative example.

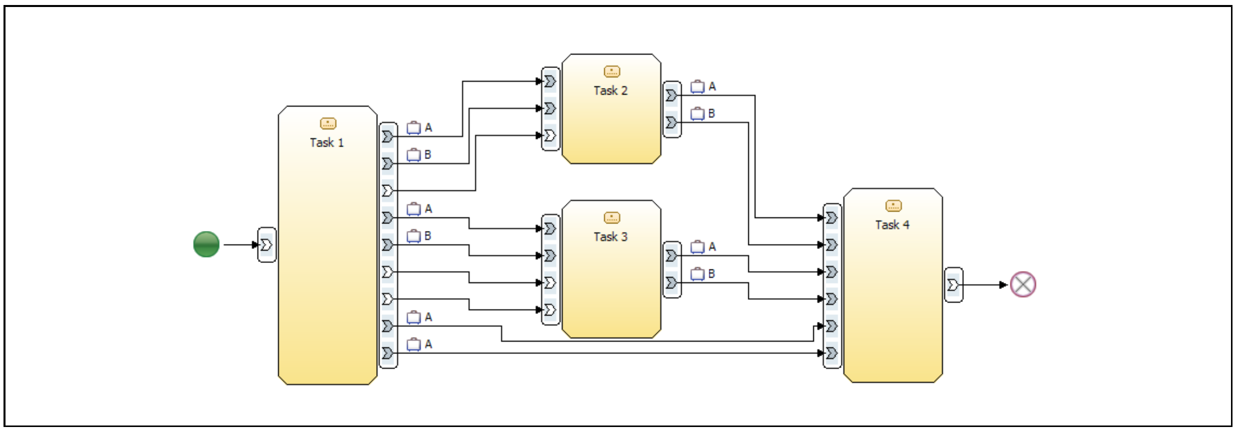


Figure 5.5: A cluttered model due to multi-connections.

The model in Figure 5.5 is very cluttered and hard to understand because of the control and data flow. Two control flow connections leave *Task 1* and end in *Task 3*. Such a control-flow multi-connection between a source and target modeling element is redundant, because it does not add any additional information to the model. Control only needs to flow once from a source element to a target element. Furthermore, if there is already a data connection drawn between the source and the target, no additional control-flow connection needs to be added, because data flow always implies control flow.

Notice also that business item A leaves *Task 1* four times, while item B leaves this task two times. Item A flows to *Task 2* and *Task 3* once, while it flows twice to task *Task 4*. Item B flows to *Task 2* and *Task 3*. Such data multi-connections usually point to a modeling problem where users either tried to pass the same item to several activities or intended to express that two different instances of the item are passed.

For example, in a negotiation process, an *offer* and a *counter offer* may be exchanged. To capture the flow of these two offers correctly, two options exist. The first option is to give meaningful different names to the inputs and outputs of the tasks using the business item. In the graphical model, WBM only shows the name of the business item. In the attributes view, we can see the names of the inputs and outputs where we can distinguish the purpose of the business item. The second option is to define a business item template and to associate several different business items with this template. For example, a template *offer* can be defined followed by two business items *initial offer* and *counter offer*, which inherit their common attributes from the *offer* template. This solution is more appropriate if indeed two different data objects flow through the process model that share a common set of attributes.

The correct passing of data along alternative and parallel flows is the subject of the remainder of this section.

Recommendations 6 *Try to avoid mixing data and control flow in a model. Decide whenever possible for a pure control-flow or a pure data-flow model. Do not use control-flow multi-connections.*

5.4 Gateway Form versus Activity Form with Data Flow

We begin by discussing the correct usage of gateway and activity form in the presence of data flow. Gateway form and activity form can be used interchangeably for process models that contain only control flow, as we discussed in Section 2. However, for process models with data flow the behavior is different depending on whether gateway form or activity form is used. In order to correctly capture complex data flows, it may even be necessary to mix both forms in a single process model.

Data-flow errors typically arise when data is flowing along several execution branches that can either capture alternative or parallel behaviors. Three different modeling scenarios can be distinguished. In the first, the same, shared data is passed along several branches in the process flow. In the second scenario, different, unshared data is passed along the branches. In the third, and most complex scenario, a mixture of unshared and shared data must be passed. We discuss these three different cases separately in the subsequent subsections.

As we pointed out earlier, WBM currently requires that all incoming and outgoing branches of gateways, i.e., of fork, join, decision, and merge must always have the same business items attached to them. It is not possible in WBM to have different business items on different branches of a gateway. Consequently, we use gateway form if we want to model how the *same, shared* information is flowing along alternative branches in case of a decision or along parallel branches in case of a fork. Figure 5.6 shows an example where items A and B are flow into *Task 2* and *Task 3*, while A and C flow out of them.

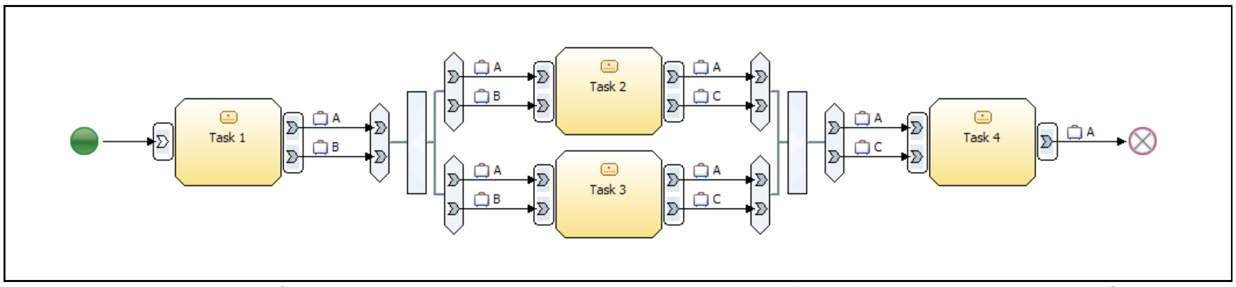


Figure 5.6: Data flow shared along several branches is correctly modeled using gateway form.

Gateway form describes how the data flow gets routed based on the *value* of attributes of business items to alternative branches in the process model. Output conditions defined for decisions can capture in detail how these values determine the branch the item flows.

If we need to model how different business items flow along the branches in a process model, we must use activity form. It is the only way to correctly capture how the data flow in a process branches based on the *type* of information, i.e., the business item. Figure 5.7 shows a model where different business items flow along several *parallel* branches.

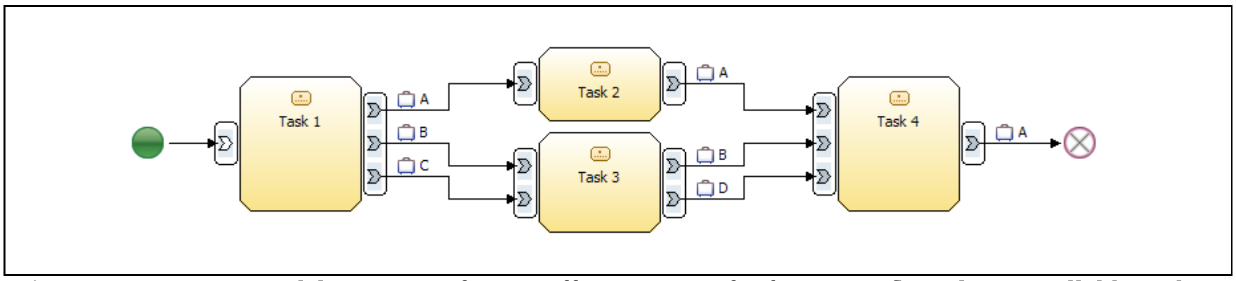
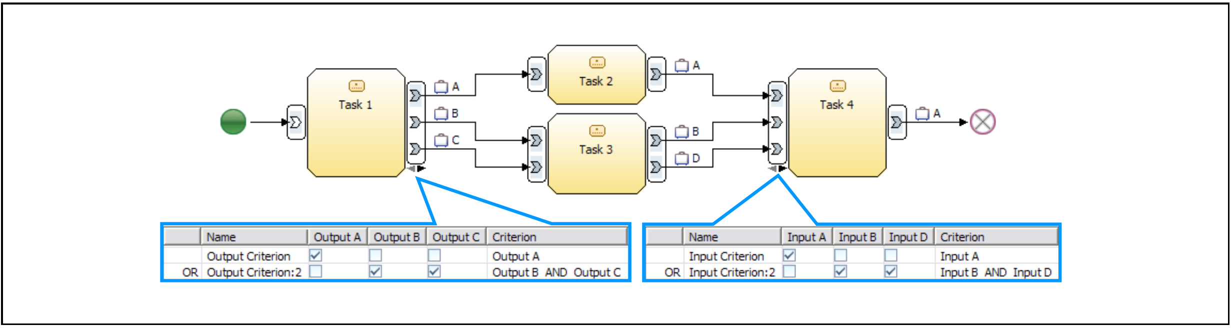


Figure 5.7: Process model in activity form. Different types of information flow along parallel branches.

In Figure 5.7, *Task 1* produces outputs A, B, and C that it routes in parallel to *Task 2* and *Task 3*. *Task 2* receives item A, while *Task 3* receives items B and C. Single output and input criteria are used (notice the absence of arrows below the inputs and outputs), because we model parallel branching where *Task 1* acts as a fork, while *Task 4* acts as a join bringing the different data flows together again.

Figure 5.8 shows the same data flow, but now flowing along alternative branches instead of parallel ones. This means that *Task 1* acts as a decision, while *Task 4* acts as a merge and two different output, respectively input criteria have to be defined for these tasks.



	Name	Output A	Output B	Output C	Criterion
	Output Criterion	✓	☐	☐	Output A
OR	Output Criterion:2	☐	✓	✓	Output B AND Output C

	Name	Input A	Input B	Input D	Criterion
	Input Criterion	✓	☐	☐	Input A
OR	Input Criterion:2	☐	✓	✓	Input B AND Input D

Figure 5.8: Process model in activity form. Different types of information flow along alternative branches.

By defining different output criteria, we model that *Task 1* provides alternative outputs, namely either item A or items B and C. Based on the output, either *Task 2* or *Task 3* execute. Note that again, this flow logic can only be viewed correctly when working in the advanced editing mode in WBM. In the basic editing mode, Figures 5.7 and 5.8 are visualized in the same way.

Now we can show the corrected model for the process in Figure 5.5. The most likely interpretation of this model is that the user wanted to show how shared information is passed on to several tasks. It is rather unlikely that *Task 1* produces several copies of the item A as output. Consequently, the process should have been modeled using gateway form as Figure 5.9 shows. We removed the redundant control-flow connections and added a new business item *A-prime* in order to distinguish the two different purposes of the item A.

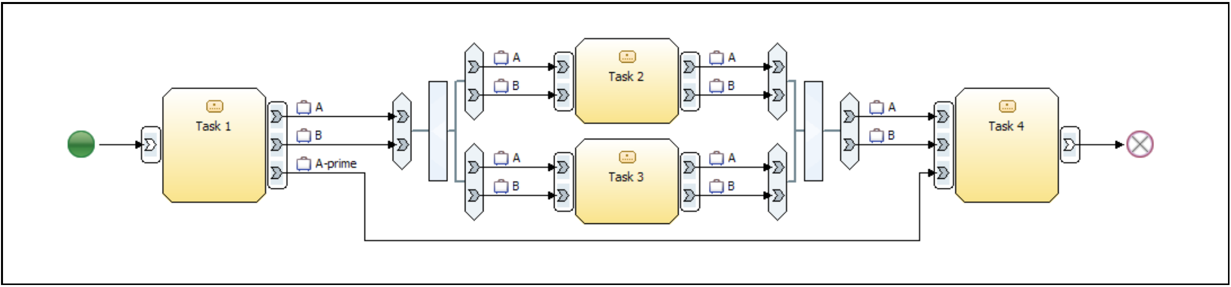


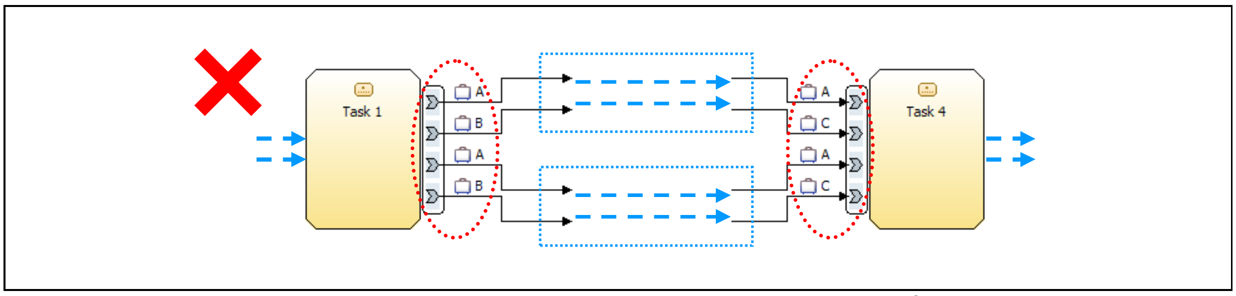
Figure 5.9: A corrected model for the process from Figure 5.5 using gateway form.

Recommendations 7 *Gateway form using decision, fork, merge, and join is appropriate to model how shared information flows along several branches in the process and where branching takes place based on the value of attributes of business items. Activity form using input and output criteria, but no gateways, must be used in models where data flow branches based on the type of the information and different business items travel along different branches. Try to separate process fragments where data flow branches based on the business item from process fragments where data flow branches based on an item attribute in order to avoid the need to mix both forms.*

5.5 Passing Shared Data along Several Branches

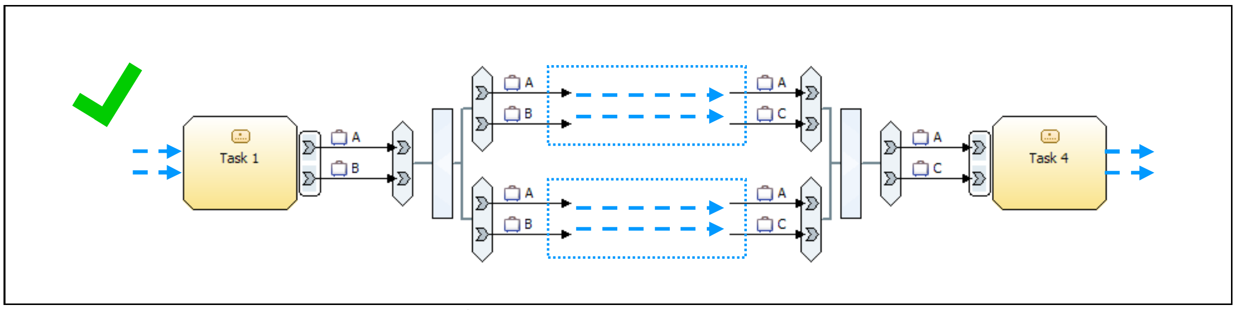
Understanding the difference in the behavior of data-flow models using activity form or gateway form provides the foundation to investigate the above mentioned three data-flow modeling scenarios in detail and to discuss the typical modeling errors that can occur. In most of the cases, identical data-flow anti-patterns and patterns apply to parallel and alternative branching. Therefore, we usually concentrate on parallel branching and discuss alternative branching flows only if there is an interesting difference. We first discuss the scenario where shared data must be routed along several branches.

Anti-pattern 5.2 shows a frequent error that we observed when using activity form to model that shared business items flow along several branches.



Anti-Pattern 5.2: Activities provide the same data outputs multiple times for using the same, shared data on parallel branches.

Using activity form leads to a duplication of data in the inputs and outputs of activities. *Task 1* would normally produce business items A and B as outputs. To route the items in parallel to two process fragments (visualized by the blue boxed areas), the outputs of *Task 1* are duplicated. Similarly, the inputs of *Task 4* are duplicated. This is not only a bad modeling practice, it also changes the semantic meaning of the model, because it adds additional business items and duplicates the inputs and outputs of activities. When activities are designed for reuse, such a duplication is a strong limitation, because any reusing process must provide two As and two Cs as input to *Task 4*, or the task cannot execute. Pattern 5.2 shows the correct modeling solution for this scenario.



Pattern 5.2: Use gateways for branching when using shared data on several paths.

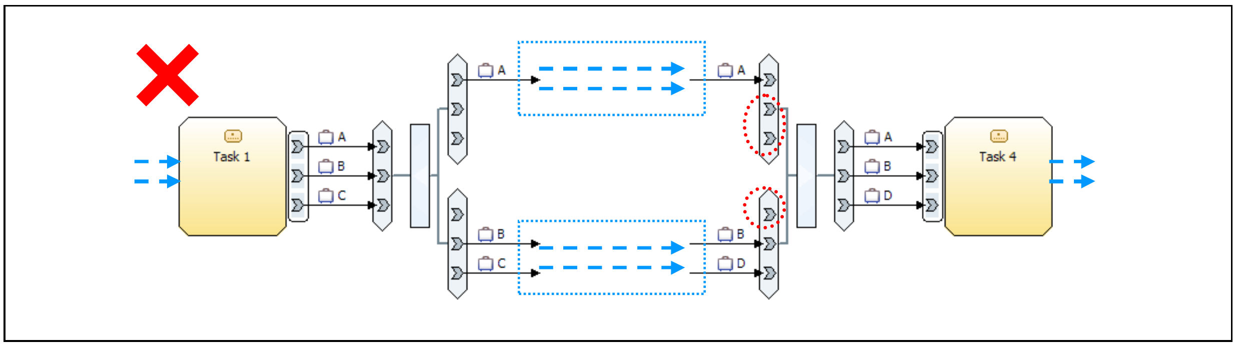
For alternative branching, an identical pattern results where the fork is replaced by a decision and the join is replaced by a merge. Gateway form is used for branching when shared data is used and produced along several branches. The process fragments that comprise these branches can of course modify the data. We can see in the pattern that item B enters both branches, but does not leave the blue process fragment anymore. Instead, an item C is provided on both branches as input to the join. Note that again the same item C must be provided on both branches for the join to be able to correctly execute.

The simulation in WBM 6.0.2 currently shows that two As and two Cs leave the join in Pattern 5.2, which cause *Task 4* to execute twice. This means, the simulation implements a semantics where items get multiplied by a fork and the join does not behave symmetrically, i.e., it does not undo the multiplication.

The join behaves more like a merge on data flow models, so a lack of synchronization can occur, unless the multiple execution of the process fragment following the join is intended. In Pattern 5.2, the lack of synchronization can be prevented by setting the minimum and maximum multiplicity of the inputs A and C to 2, because two occurrences of each item have to be synchronized by *Task 4*. Unfortunately, this solution makes reusing *Task 4* in other process models more difficult. Furthermore, computing the right multiplicities can be challenging in models with more complex fork-join structures.

5.6 Passing Unshared Data along Several Branches

When unshared data flows along different alternative or parallel branches, many users are tempted to use gateway form. This often leads to dangling inputs in a join or merge, which cause a deadlock. Anti-pattern 5.3 illustrates this error.

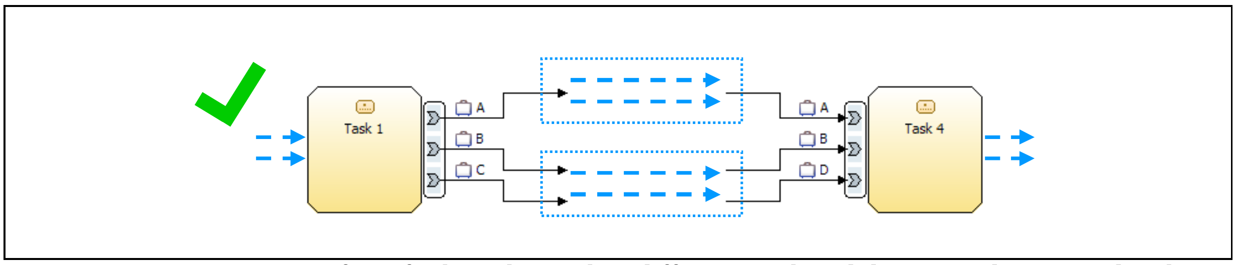


Anti-Pattern 5.3: Passing unshared data along parallel branches using gateways causes deadlocks due to dangling inputs.

The dangling inputs in the join prevent it from executing and thus block all activities succeeding it. This means, *Task 4* in the anti-pattern cannot execute, although the task has all its inputs connected. The dangling outputs of the fork are not causing an execution problem, but they lead to a process model where certain outputs are not used. Additional dangling inputs can occur in scenarios where process fragments within the blue frame produce additional data. For example, a new business item D is provided as output of some task within this process fragment and it replaces input item C.

Very often, this error is corrected by wiring all business items through all tasks, although these tasks do not need to access these business items. For example, items B and C would additionally flow through the upper parallel branch. There are several good reasons to not wire unnecessary data through activities. First, it makes reusing activities more difficult as they require additional input and output that not all reusing processes may be able to provide. Secondly, it exposes information to activities that do not need it, which can later cause security and performance problems when designing the implementing IT solution by closely following the process model.

Pattern 5.3 shows the correct solution using activity form.

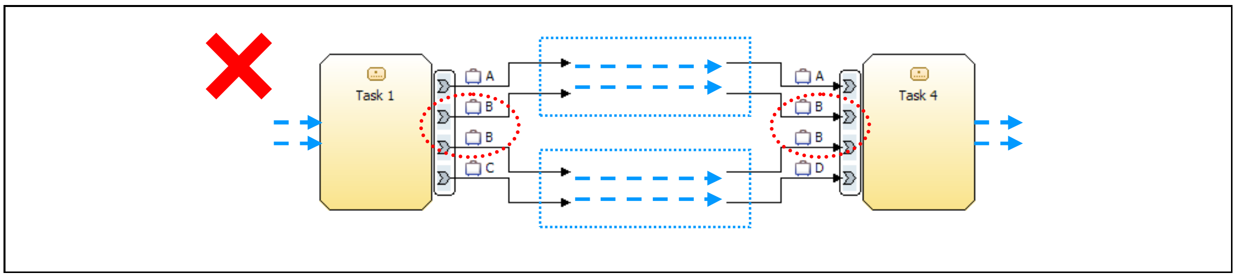


Pattern 5.3: Use activity form for branching when different, unshared data is used on several paths.

This pattern shows the case of parallel branches. When modeling alternative branching flows that work on unshared data it is important to correctly define the input and output criteria of the activities. In particular, the input criteria of activities that act as implicit merges must exactly match the alternative branches. Otherwise, deadlocks or a lack of synchronization can occur. If we change Pattern 5.3 to show two alternative branches, we need to define one input criterion for *Task 4* that includes item A and a second input criterion that includes items B and D.

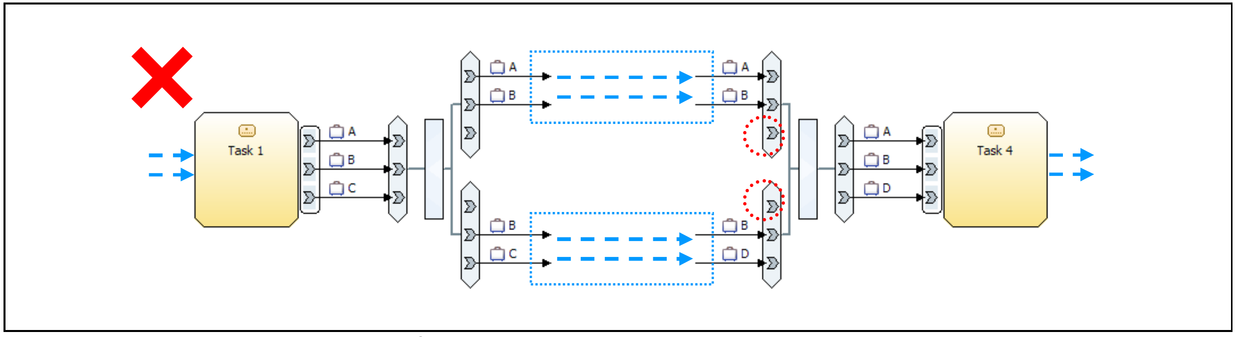
5.7 Passing the Shared and Unshared Data along Several Branches

The third scenario covers the case where a subset of the data is shared on all the branches, but an individual branch works on data that is specific to this branch. Using only activity or gateway form cannot lead to a correct model. Anti-pattern 5.4 shows a scenario where the upper branch works on item A, the lower branch receives item C and produces item D, but both branches work on item B. Activity form was used to model this process. We notice immediately the problem of the duplicated item B in the output of *Task 1* and the input of *Task 4*.



Anti-Pattern 5.4: Using activity form alone to pass shared and unshared data along several paths leads to a duplication of inputs and outputs.

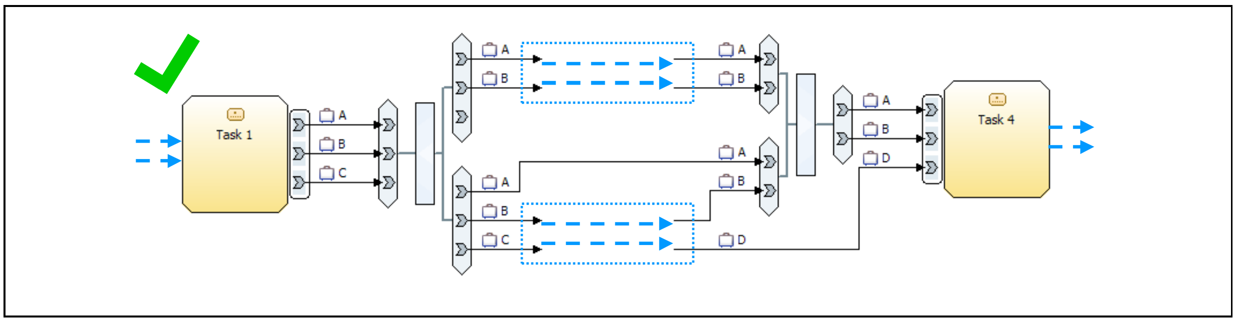
Anti-pattern 5.5 shows that gateway form leads to dangling inputs in the join that cause a deadlock.



Anti-Pattern 5.5: Using gateway form alone to pass shared and unshared data along several paths leads to deadlocks caused by dangling inputs.

The only solution is to mix both forms. Gateways are used to route business items that the branches share, while input and output criteria are used to route business items that are specific to a branch. Pattern 5.4 shows a solution that works for parallel branches.

The shared items A and B branch through the fork and rejoin in the join. It is important that the join expects no other input in order to avoid dangling inputs. Item C is only needed as input for the lower branch. It is passed through the fork, where it creates a dangling output in one of the branches. This is not an ideal modeling solution, but at least it allows the process to execute. Alternatively, C could bypass the fork and enter the blue process fragment directly. D must, however, bypass the join and enter *Task 4* directly to avoid the deadlock. Note that A flows through the fork and join, but it bypasses the process fragment in the lower branch of the fork, because it is not required by activities inside this fragment. A could bypass the fork and join, but it still needs to enter and leave the process fragment in

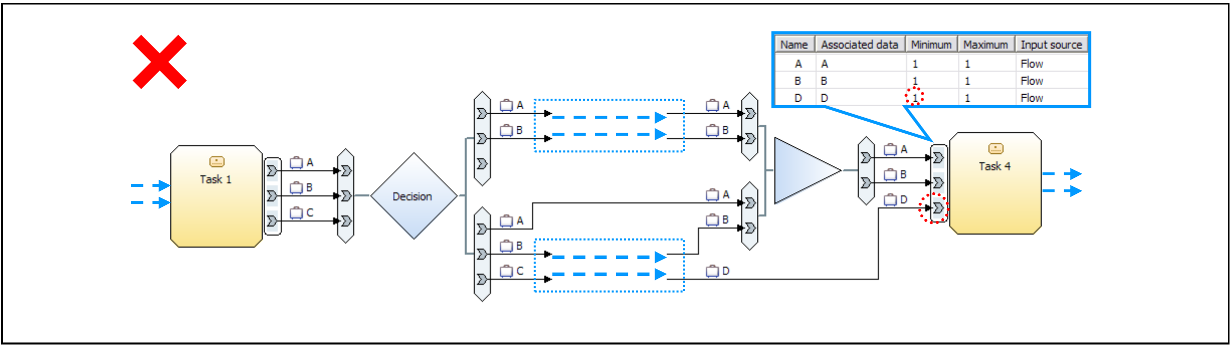


Pattern 5.4: Bypassing gateways is possible in parallel flows.

the upper branch of the fork. It also needs to enter *Task 4*. We show in the pattern a selected variant of bypassing that eliminates the critical deadlock. Additional flows that bypass gateways are possible, but such a solution quickly leads to a cluttered diagram.

Bypassing forks and joins in parallel flows does not cause a deadlock, because all branches execute in parallel and thus, all business items always arrive through the connections, i.e., all tasks receive their inputs as specified. For example, *Task 1* produces all its outputs and they are therefore available for *Task 4*. In the case of alternative flows, the availability of inputs is not guaranteed for an activity that expects these inputs from several alternative branches.

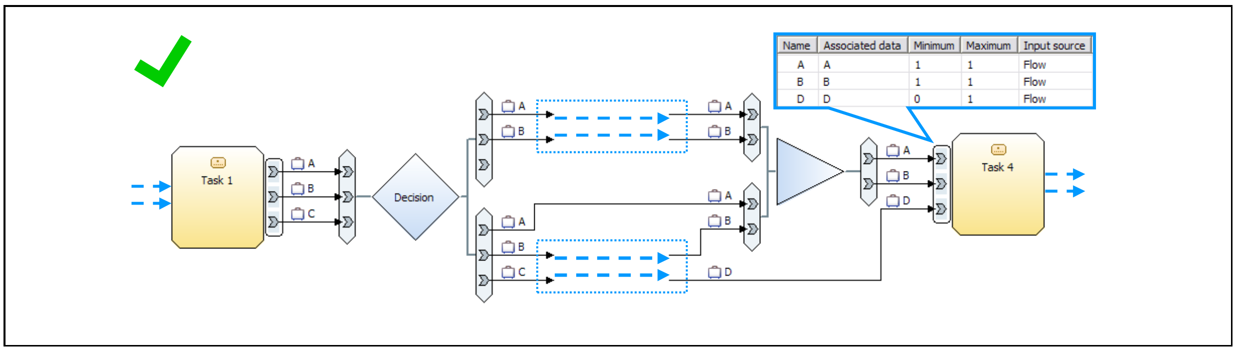
Anti-pattern 5.6 shows a process model with alternative branches where item D, which is only produced by the lower branch, bypasses the merge to enter *Task 4* directly. D is a required input of *Task 4*.



Name	Associated data	Minimum	Maximum	Input source
A	A	1	1	Flow
B	B	1	1	Flow
D	D	1	1	Flow

Anti-Pattern 5.6: Bypassing gateways of alternative branches can cause deadlocks.

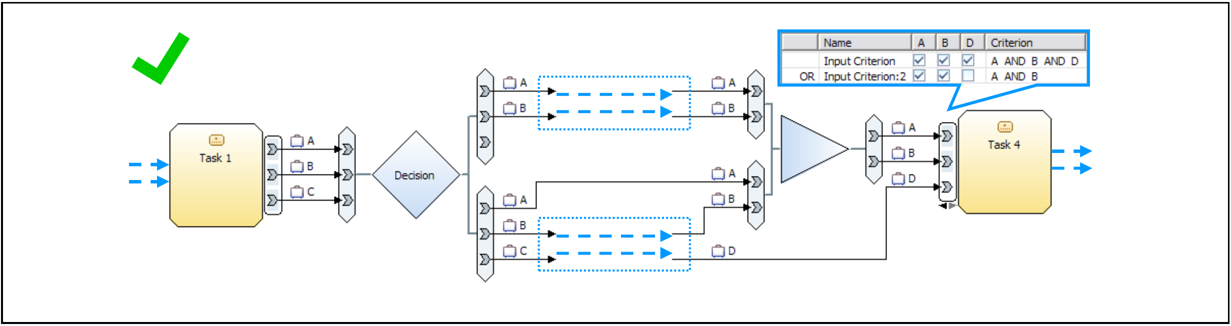
Unfortunately, D is not available if the upper branch executes. A required input must always be provided by all alternative branches. If an input is not required, but optional, two possible solutions exist to correct the input behavior of *Task 4*. Pattern 5.5 shows the first solution where the minimum multiplicity of item D as input of *Task 4* is set to 0.



Name	Associated data	Minimum	Maximum	Input source
A	A	1	1	Flow
B	B	1	1	Flow
D	D	0	1	Flow

Pattern 5.5: Inputs provided along only some of the alternative paths must be optional.

Pattern 5.6 shows the second solution by defining several input and output criteria for *Task 4* that correctly match the alternative branches. *Task 4* has an input criterion requiring only business items A and B for the upper branch, while for the lower branch it has a separate criterion that includes business items A, B, and D.



	Name	A	B	D	Criterion
	Input Criterion	✓	✓	✓	A AND B AND D
OR	Input Criterion:2	✓	✓	☐	A AND B

Pattern 5.6: Input criteria must precisely match the data that flows along alternative paths.

Both solutions enable *Task 4* to execute correctly and independently of the branching decision that was taken in the decision gateway.

Recommendations 8 *When modeling complex data flows, take a systematic approach based on scenarios that distinguish whether shared or unshared data has to be passed along several flows. Then determine whether these flows occur as parallel or alternative branches in the process model. You can use activity form to capture the flows, but when branches share data, it leads to duplicated inputs and outputs of activities that should be avoided. Gateway form is a better solution for this scenario. When modeling alternative branches, make sure you pay attention to data that is only available on one of the branches. This data must be an optional input for any activity following the merging of the alternative branches.*

6 Scenario IV: Modeling Events and Triggers

Very often, users want to model events and triggers in a process model.[7] WBM supports events in the context of *business measures* that are used to define *key performance indicators* for process monitoring, but not directly as first-class modeling elements in the process model itself. Currently, it has support for events in the form of a *notification*, which can be received via a *notification receiver* and broadcasted via a *notification broadcaster*, i.e., an abstract modeling of publish-subscribe communication is supported. A point-to-point event-based communication cannot be modeled so far and it also cannot be captured how events flow through a process model.

6.1 Events as Control Flow?

We observed that very frequently control flow is used to capture events. However, the semantics of control flow in WBM assumes that control flow is something that occurs and that defines an order of execution between activities—it does not carry any information as events usually do. Consequently, a modeling practice that captures events with control flow leads to various semantic problems. Figure 6.1 shows an example using control flow to capture the logic behind several initial and final events that occur in a process.

[7] In our own understanding, we make no semantic difference between events and triggers and only speak of events in the following.

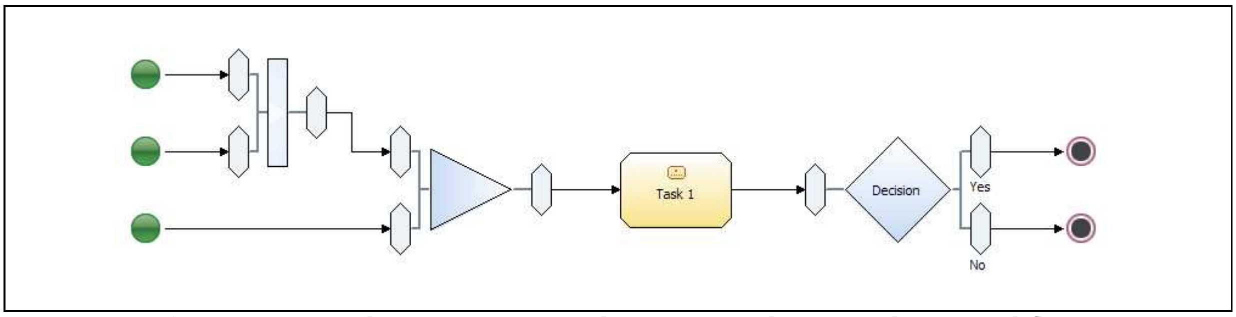
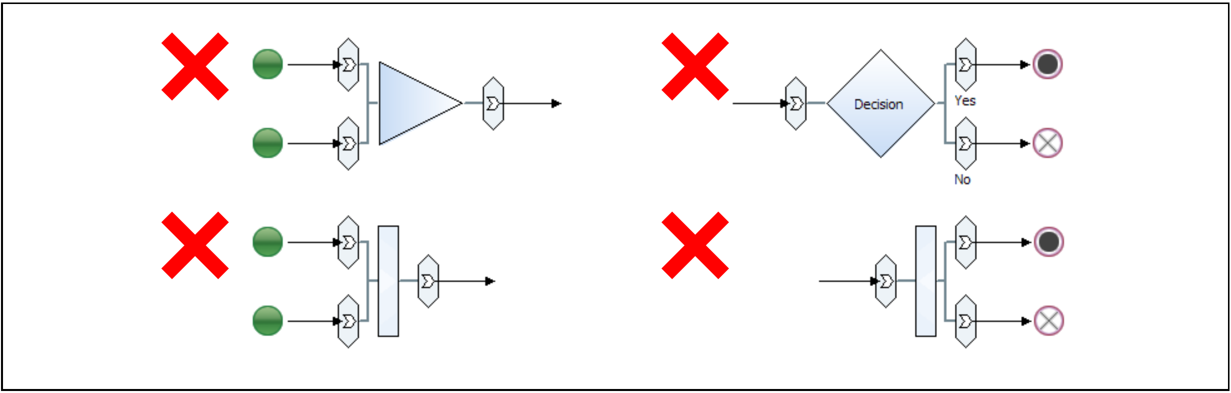


Figure 6.1: Complex event triggering logic incorrectly captured as control flow.

In this example, three start nodes are used to represent three different initial events that can initiate execution of the process. The control connections from these start nodes to the subsequent gateways were named with events, however, these names are not visualized and can only be seen by clicking on the connection or opening the attributes view. This leads to a model where essential information about the events is not directly visible in the graphical representation. A merge and a join are used to capture the event logic. The user's intention here was to describe a process that is triggered by a single event or by two events that must jointly occur. To express the event logic *(event1 AND event2)*, the user introduced the upper two start nodes and connected them to a join. The lower start node represents the third event. This start node and the join are both connected to a merge in order to represent the event logic *(event1 AND event2) OR event3*.

However, the semantics of start nodes in WBM specifies that always all start nodes of a process model execute *at once*. In case of the example, all three event-representing control connections are triggered immediately and then the join executes. The merge executes twice, once when it receives control from the lower start node, and again when it receives control from the join. Consequently, *Task 1* executes twice in all executions of this process. This behavior can be observed in the WBM simulation. So the three alternative events that the user tried to capture with the three start nodes always occur together and are by no means alternative triggers for the subsequent task. The two final events that the task emits after successful execution have been captured in a decision with two outgoing branches that directly end in a stop node. Stop and end nodes cannot pass any event information outside the process—they only stop the control flow. Therefore using control flow to depict events is not a good idea.



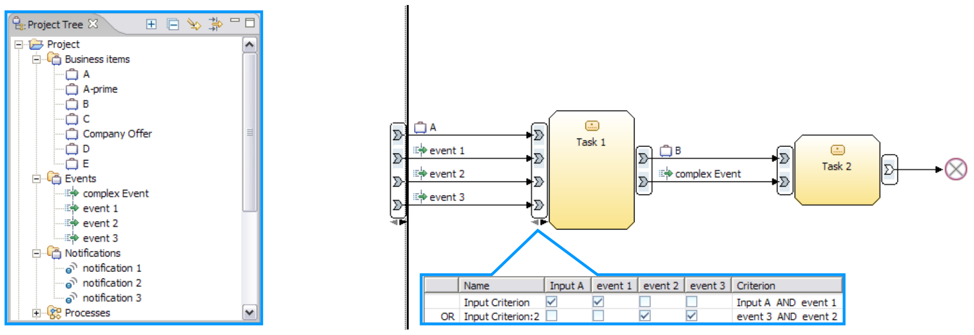
Anti-Pattern 6.1: A merge or join only preceded by start nodes, and a decision or merge only followed by and end or stop nodes is an error.

Anti-pattern 6.1 generalizes this insight. Multiple start nodes directly linked to gateways are not suitable to capture any triggering logic of a process. Connecting several start nodes directly to a merge causes a lack of synchronization. A join that only has start nodes has input can be replaced by a single start node that should directly connect to a task or subprocess. Connecting all outgoing branches of a decision or fork directly to end or stop nodes shows that the decision or fork is unnecessary. A gateway should lead to branches that contain tasks or subprocesses. Usually, only one of the branches should capture a “do-nothing” case and directly link to an end or stop node. We deal with the difference in the semantics of the end and stop node in Section 7.

6.2 Events as Data Flow

In our own practice, we use two alternative solutions to capture events. In process models, which are not intended to be exported to the IT level and where we want to capture start and end events of the process, but do not need to show the event flow, we use *receivers* and *notification broadcasters*. Information on how to correctly use these modeling elements can be found in the WBM documentation. In process models, where we want to describe how information received through events flows between the activities in a process, we use business items and data-flow connections.

Figure 6.2 shows on the left three different catalogues of business items to distinguish *events*, *notifications*, and “normal” *business items* from each other.

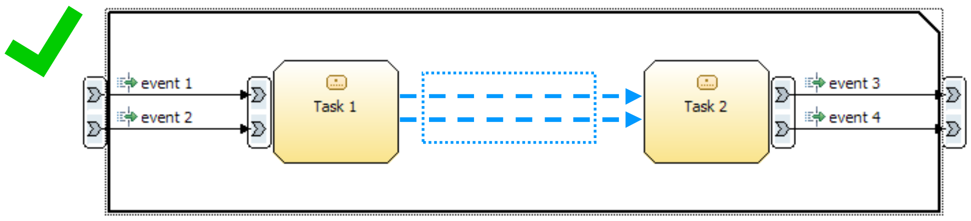


	Name	Input A	event 1	event 2	event 3	Criterion
	Input Criterion	✓	✓			Input A AND event 1
OR	Input Criterion:2			✓	✓	event 3 AND event 2

Figure 6.2: Event flow represented as a specific kind of business item flow.

Each kind of information is associated with a different icon, which can be easily customized in WBM. Notifications and business items are available as predefined catalogues, while the *Events* subcatalogue is user-defined. The process model fragment in this figure shows an example where *Task 1* can execute if it either receives a business item A together with some *event 3* or if *events 1* and *events 2* occur. *Task 1* sends business item B to *Task 2* together with a *complex event*, which now flows through the process similar to other business items. As events are modeled as specific kinds of business items, attributes can be defined for them in order to capture in more detail what information they carry. This information is also accessible when modeling decision conditions.

Pattern 6.1 illustrates the approach of using data flow to represent events. Initial events are received via the process input interface, while final events leave the process via the process output interface.



Pattern 6.1: Events can be modeled as business items.

Recommendations 9 *Do not use control flow to model events and triggers in a process model. Either use the modeling elements that are provided to capture notifications, or represent events as a specific kind of business item flow. Do not connect all inputs or outputs of a gateway or activity directly to start, end, and stop nodes.*

7 Scenario V: Correct Termination of a Process

WBM offers two types of nodes to terminate a process, which are called the *end* and the *stop* node. The end node is visualized with a circle containing a cross, while the stop node is visualized with a circle containing a black dot. A stop node stops all activities and flows in the process model. This means, the stop node has a termination effect on *all* executing branches within the whole process model, i.e., it leads to a "global shutdown" of the entire process.

If more than one branch executes, e.g., if the model contains some parallelism, the stop node always terminates all parallel branches. In contrast, the end node only has a local effect: it only ends the single branch through which it was reached. Given the more global effect of the stop node on the entire process, we have to be careful putting a stop node in process models that can have several branches executing in parallel.

From the perspective of the semantics, we can use both nodes interchangeably in models that for sure only have a single sequential execution, e.g., those models that do not use forks, inclusive decisions, cyclic connections, and branching output criteria. From a tool perspective, at least one stop node is required within every process, subprocess, and loop in WBM 6.0.2. The simulation in WBM 6.0.2 requires that every path in the process model ends in a stop node, i.e., the end node should be rarely used. The stop node is particularly important when simulating data flow models, because it is required to release the data. This means, that a parent process can only receive data from a subprocess when a stop node is reached or when the *advanced output logic* (see the tab of the same name) of the subprocess is set to *streaming*, i.e., the subprocess releases data while still running.

In the following discussion, we explore the semantic difference between the end and the stop node in more detail. In particular, we investigate the global shutdown effect of the stop node.

7.1 The Stop Node in Parallel Execution Branches

Very often, users are not aware of the global effect of a stop node and use it to end each of the individual branches of a process. Figure 7.1 shows a typical example.

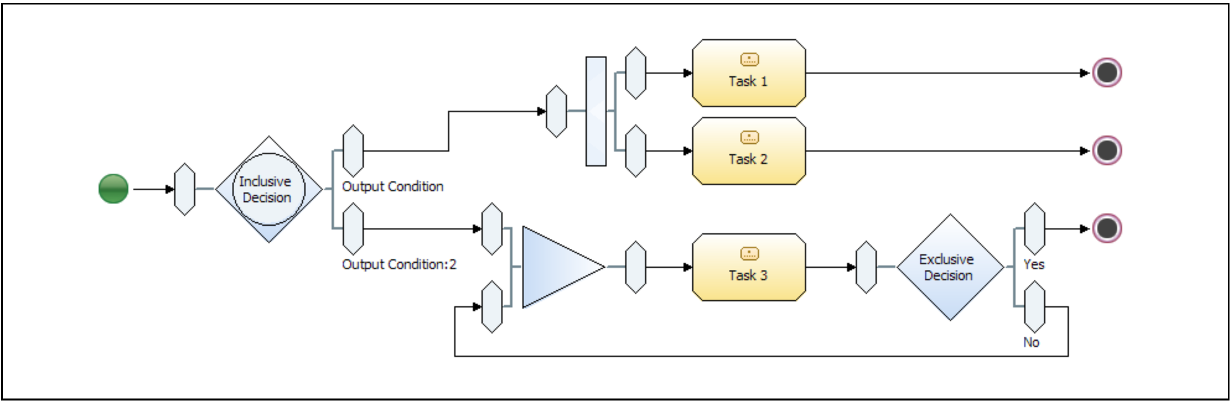
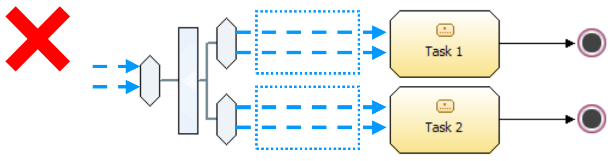


Figure 7.1: Stop nodes used in a process model with parallel execution branches.

Immediately following the start node, we see an inclusive decision with two branches. The upper branch leads to a fork that causes *Task 1* and *Task 2* to execute in parallel. Both tasks connect directly to a stop node. The lower branch of the inclusive decision leads to a cyclic process fragment, where *Task 3* iterates until a decision condition is satisfied. The 'yes' branch of this second decision is directly connected to a stop node.

As soon as one of the branches reaches the stop node, the entire process terminates, even if tasks have not finished their execution. Such a "global shutdown" can sometimes be intended and this happens during the simulation. However if we think of the IT implementation, this is most likely not the intended process behavior: instead, branches running in parallel should end individually after the tasks on these branches finished correctly.

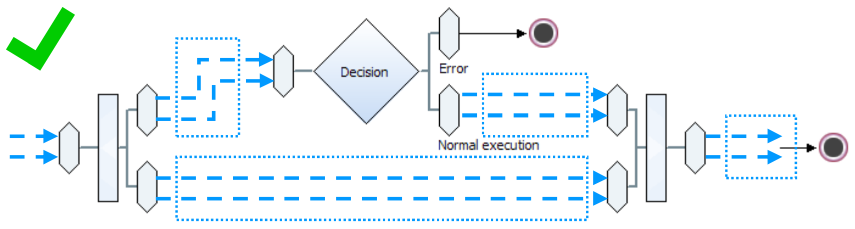
A problem occurs if the initial decision is indeed inclusive and activates both branches. If only the lower branch is activated, no problem occurs, because the cyclic process fragment only contains a sequential loop, i.e., *Task 3* is executed repeatedly, but several instances of the task do not run at the same time. If the decision only activates the upper branch, there can still be a problem in the fork if *Task 1* and *Task 2* represent activities of different duration. When one of the tasks finishes, it would also cause the immediate termination of the other task. When the inclusive decision activates both of its outgoing branches and *Task 1* or *Task 2* only have a very short execution duration, this could in principle also cause the cycle to never execute, because as soon as one of the upper stop nodes is reached, it terminates the whole process including the cycle. Anti-pattern 7.1 alerts the user of stop nodes used in parallel branches.



Anti-Pattern 7.1: Stop nodes ending parallel branches always terminate the whole process even if they are only meant to end a single execution branch.

If we replace the inclusive decision and fork with two exclusive decisions, only one sequential execution path results for the process. In this case, end and stop nodes can be used interchangeably. Whithout parallelism, the stop node has exactly the same effect as the end node, i.e., it ends the single branch through which it was reached.

Given the current requirements of the simulation, we recommend to prefer the stop node in models. Users should be aware, however, that some parallel paths may not have finished execution when the stop node is reached during a simulation run. The BPEL export in WBM maps the end and the stop node to an implicit end of the BPEL process. It does not generate an explicit global termination behavior in the BPEL for a stop node, e.g., via a BPEL *terminate* activity.



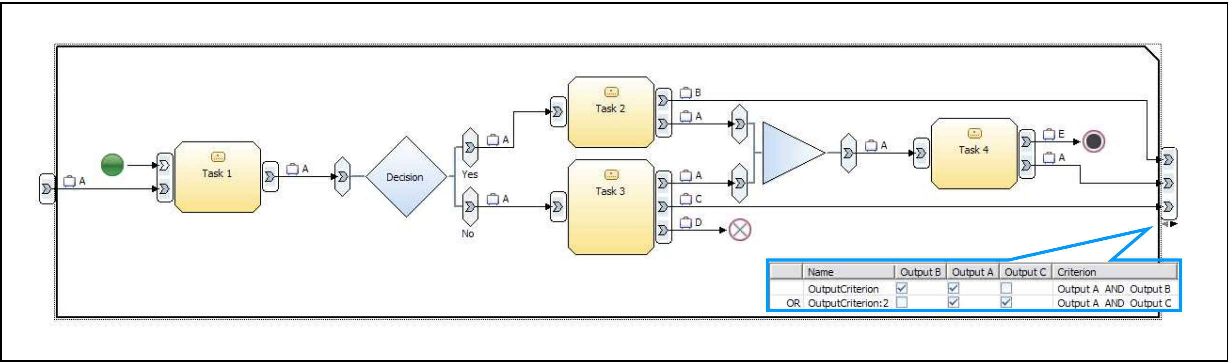
Pattern 7.1: Stop nodes model a global shutdown behavior. They can be used safely when this behavior is intended, when only a single sequential execution occurs, or when parallel branches have been rejoined before reaching a single stop node.

Pattern 7.1 summarizes our discussion and recommends to rejoin parallel branches before adding a stop node to terminate the process.

Recommendations 10 *Use the stop node when you want to model a "global shutdown" of a process. Rejoin parallel branches with a join node and then place a single end or stop node, instead of ending parallel branches individually.*

7.2 Data Output upon Termination of a Process

Finally, we want to discuss the process boundaries and the inputs and outputs of a process. Very often, processes receive data as inputs and produce data as outputs similar to activities inside the process model. Consequently, input and output criteria can be defined for the process to represent the process interface. Figure 7.2 shows an example of a process for which inputs and outputs are defined.



	Name	Output B	Output A	Output C	Criterion
	OutputCriterion	✓	✓	☐	Output A AND Output B
OR	OutputCriterion:2	☐	✓	✓	Output A AND Output C

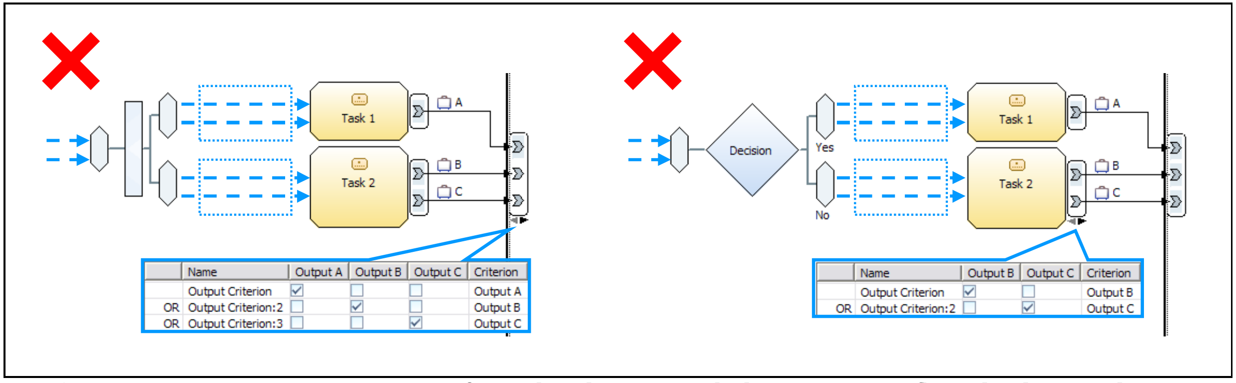
Figure 7.2: Input and output data of a process, combined with start and stop nodes.

The process receives item A as input, which it passes on to *Task 1*. The task also has a start node, however, this node is optional and does not change the execution semantics of the task, which can only begin execution when the input is available. The additional start node visualizes the process beginning more clearly. The process produces items A and B, or A and C as outputs. A is an output of *Task 4*, B is an output of *Task 2*, and C is an output of *Task 3*. Item D is an output of *Task 3*, and is not used by any other task and not provided as an output of the process. It is therefore connected to an end node. Using a stop node instead, would not be a correct solution, because it would immediately end the whole process and prevent execution of *Task 4*. Item E is an output of *Task 4* and connected to a stop node, because it is not used by any other activity in the process. This use of the stop node is correct, because *Task 4* is the last executing task of this process and no other activities execute in parallel. The process interface shows two alternative output criteria in order to match the alternative outputs.

Defining a process interface is done in the same way as defining the inputs and outputs of activities within the process. The definition needs to take into account the different execution branches that can occur, which can lead to different combinations of business items that can reach or leave an activity. We discussed this in detail in Section 5. A possible source of errors is the mismatch between the input and output criteria of the process and its possible execution branches. It is important to link alternative branches to alternative input and output criteria.

Anti-pattern 7.2 summarizes typical problems of process interfaces that do not match their incoming flows. It shows two parallel branches on the left that always provide outputs A, B, and C in parallel. The process output interface only expects a single output in each output criterion, which means that the process internally decides which data to release via its output interface, and this data can be different any time the process runs. A reusing process must therefore be able to handle any of the possible outputs. On the right, we have two alternative branches that either provide A or B or C as output of the process. The two alternative branches result from the decision. In addition, *Task 2* has B or C as alternative outputs.

The output criterion always expects all three business items in a single output criterion. In this situation, a reusing process would never receive all the data that was specified in the output interface of the reused process, because the process fragment shown in the left of the anti-pattern is deadlocking, because it cannot release all the required data. The correct solution is to match the single output criterion on the right with the process fragment on the left and vice versa. Note also that a stop node was not provided in the anti-pattern, which is currently required to release the data in a simulation run.

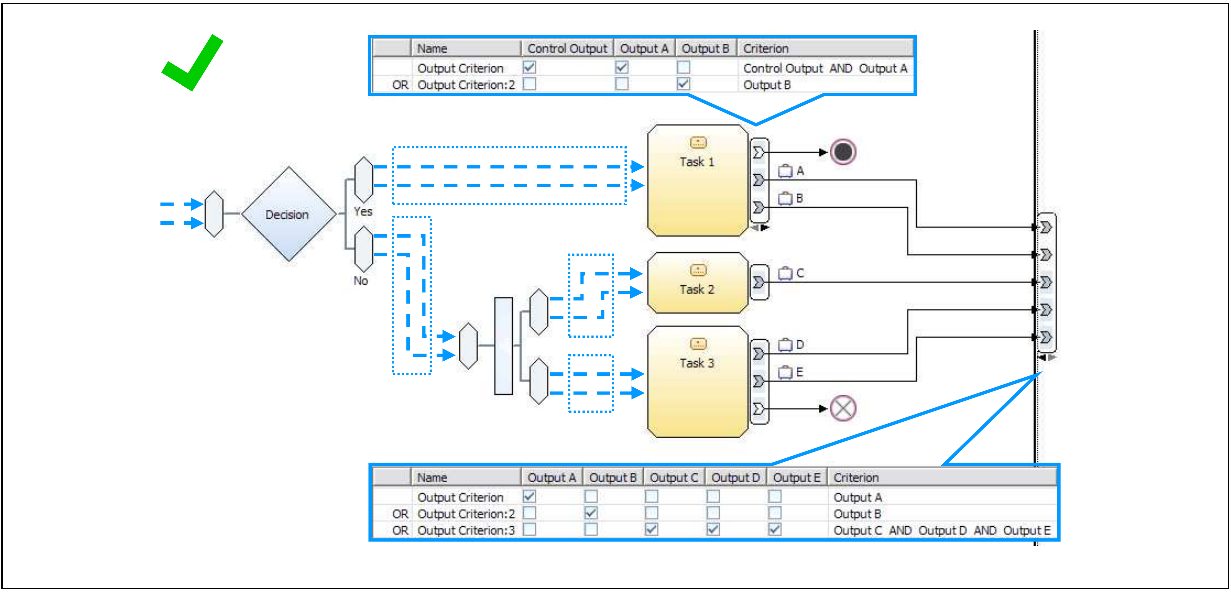


	Name	Output A	Output B	Output C	Criterion
	Output Criterion	✓	☐	☐	Output A
OR	Output Criterion:2	☐	✓	☐	Output B
OR	Output Criterion:3	☐	☐	✓	Output C

	Name	Output B	Output C	Criterion
	Output Criterion	✓	☐	Output B
OR	Output Criterion:2	☐	✓	Output C

Anti-Pattern 7.2: Process output interfaces that do not match their incoming flows lead to nondeterminism (on the left) or to a deadlock (on the right).

Pattern 7.2 shows correctly matching process output interfaces and a correctly placed stop node that does not lead to an unintended global shutdown. However, the process does not release its data in the simulation, because the parallel branches involving *Task 2* and *Task 3* do not end with a stop node.



	Name	Control Output	Output A	Output B	Criterion
	Output Criterion	✓	✓	☐	Control Output AND Output A
OR	Output Criterion:2	☐	☐	✓	Output B

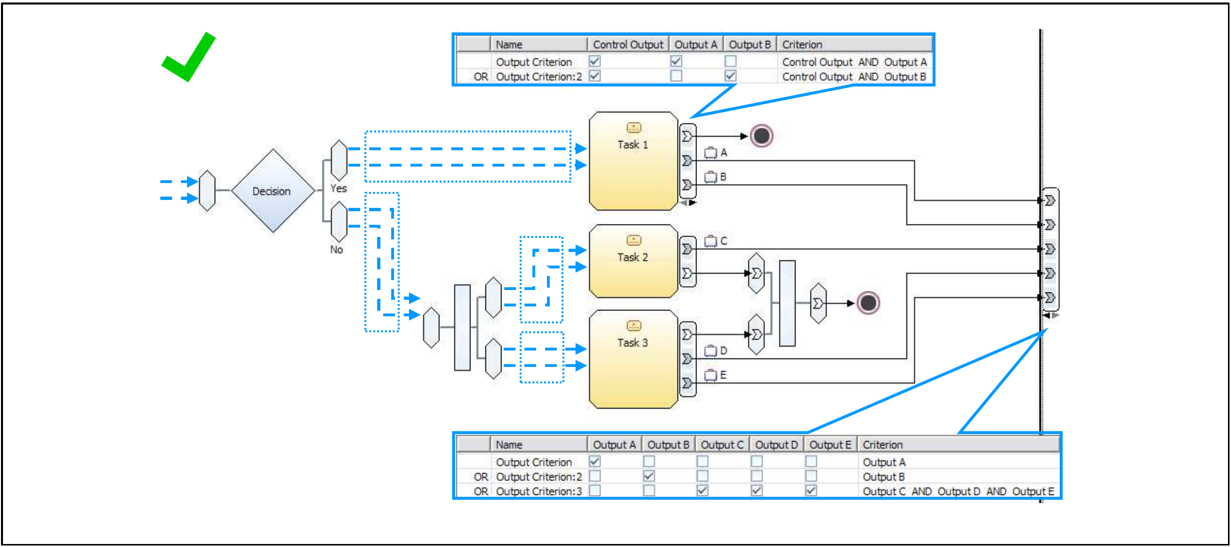
	Name	Output A	Output B	Output C	Output D	Output E	Criterion
	Output Criterion	✓	☐	☐	☐	☐	Output A
OR	Output Criterion:2	☐	✓	☐	☐	☐	Output B
OR	Output Criterion:3	☐	☐	✓	✓	✓	Output C AND Output D AND Output E

Pattern 7.2: Process output interfaces must correctly match their incoming flows.

Each branch in a process must begin with a start node or by receiving data as input. It is also a good modeling practice to end each branch of a process either with a termination node or by providing data as output. To increase the well-formedness of process models, we recommend to rejoin branches opened by forks and decisions using joins and merges when they use the some shared data. We discussed this in detail in Section 5.

Sometimes, activities within a process output data that is not part of the output of the process. This output often remains unconnected and becomes a dangling output. An alternative is to connect this output to an end or stop node in order to emphasize that the process has been completely modeled. For this purpose, WBM provides an asymmetry between the start and the end and stop nodes with respect to data flow. A start node can only issue control flow, while stop and end nodes can also receive data flow.

Pattern 7.3 shows the further improved model. Note that a join was added to rejoin the two parallel branches so that they can lead to a single stop node. The end node, to which *Task 3* previously connected, is no longer needed. No unintended shutdown can occur and all data output is correctly released.



	Name	Control Output	Output A	Output B	Criterion
	Output Criterion	✓	✓	☐	Control Output AND Output A
OR	Output Criterion:2	✓	☐	✓	Control Output AND Output B

	Name	Output A	Output B	Output C	Output D	Output E	Criterion
	Output Criterion	✓	☐	☐	☐	☐	Output A
OR	Output Criterion:2	☐	✓	☐	☐	☐	Output B
OR	Output Criterion:3	☐	☐	✓	✓	✓	Output C AND Output D AND Output E

Pattern 7.3: Process output interfaces must correctly match their incoming flows and parallel branches rejoin before reaching a stop node to release the data and avoid an unintended global shutdown.

Recommendations 11 *Process interfaces must correctly match the data flows of the process. Each process branch should be terminated with an end or stop node. Branches that release data must be ended with a stop node. They should be rejoined before adding the stop node in order to avoid an unintended global shutdown. Data output of activities, which is not released to the process output interface, should be connected to an end or stop node in to avoid dangling outputs.*

8 Scenario VI: Reuse of Activities in Hierarchical Process Models

So far we discussed subprocesses and tasks from the perspective of a single process model, i.e., from a local point of view. However, different processes can share the same activities and increasing their reuse in a process model is very desirable. To help with reuse, WBM offers the possibility to define tasks and subprocesses as *global* modeling elements directly in the project tree from where they can be dragged and dropped into other process models for reuse.

Figure 8.1 shows a project tree on the left and a compact view of the hierarchical levels of a composed process on the right. The project tree contains the definitions of the global processes and global tasks available for reuse. The compact view on the decomposition hierarchy of the *Main Process* lets us to immediately see where processes and tasks are located on the hierarchy levels.[8] Whenever possible, each activity should only be defined once and thus, should also be implemented only once. The definition of this activity can subsequently be reused by any other process.

[8]Plugins have been implemented for WBM to visualize the decomposition hierarchy and the reuse of activities. Interested readers should contact the authors to inquire about the availability of these plugins.

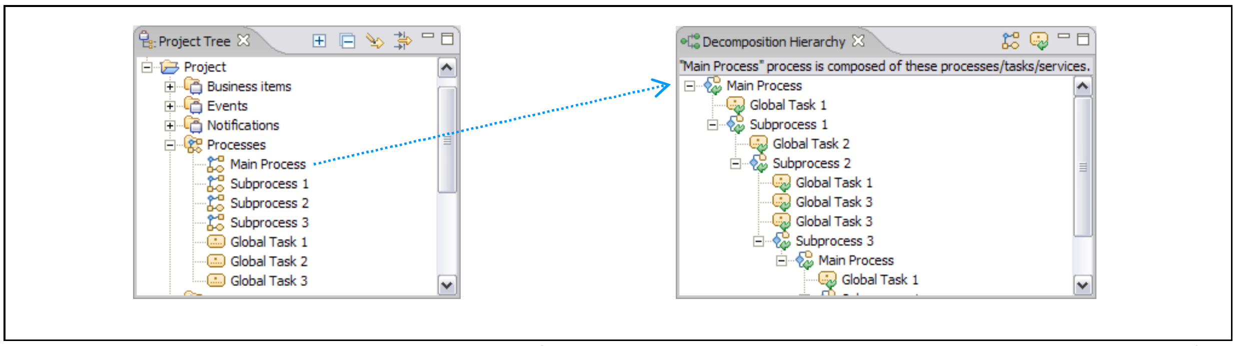


Figure 8.1: Global tasks and processes defined in the project tree and reused at several levels of a process.

Figure 8.2 shows the composed process in more detail. The example process is built from four levels as both figures show. On the top level, we find the *Main Process*, which reuses *Subprocess 1* and *Global Task 1*. *Subprocess 1* reuses *Global Task 2* and *Subprocess 2*. *Subprocess 2* reuses *Subprocess 3*. The reused subprocess is followed by a decision with two branches. Both branches reuse *Global Task 3*. As this task occurs twice in *Subprocess 2*, the tool distinguishes its two occurrences by adding a ":2" to the second occurrence of this task. The upper branch of the decision reuses *Global Task 1*, *Main Process* also reuses. Finally, *Subprocess 3* invokes the *Main Process* again.

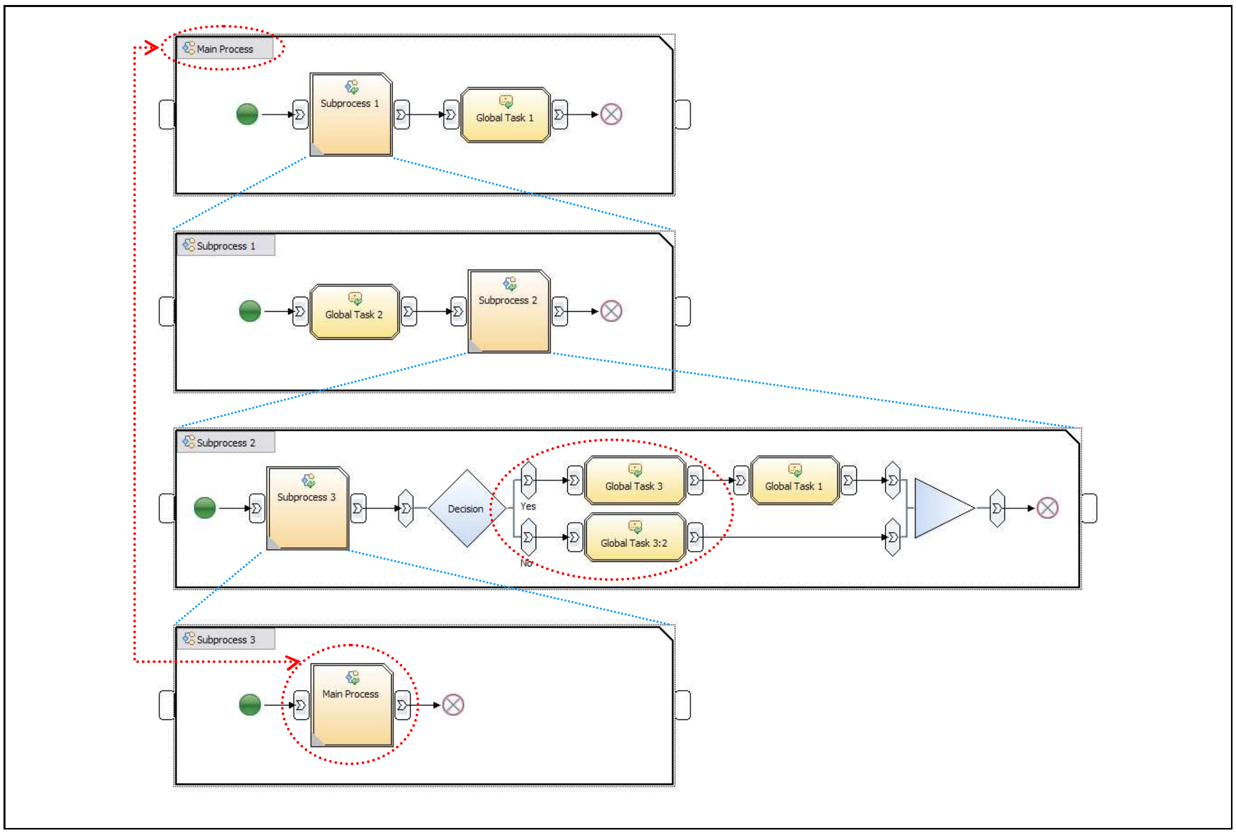


Figure 8.2: A detailed view of a hierarchical process model.

Since we focus on common modeling errors in this report, we do not want to discuss aspects of a good hierarchical decomposition, but highlight what can go wrong when a process is composed by reusing subprocesses and global tasks. The example above illustrates two sources of errors. First, we can see that *Main Process* occurs twice in the hierarchical composition, which means that it reoccurs again in its own decomposition. Such an occurrence of a process within its own decomposition hierarchy leads to a recursive refinement. In case of the example process, the recursion is infinite: *Subprocess 3* always invokes *Main Process* again. In a proper recursive process definition, *Subprocess 3* would contain a decision with two alternative branches where one branch covers the recursive invocation, while the other leads to a stop or end node.

Infinite decompositions can easily occur in larger composite processes, where maintaining a global overview of the model becomes difficult. Usually, users do not intentionally create recursive process models. They result from dragging and dropping processes or tasks on different hierarchical levels of the process. Very easily, reuse chains as in Figure 8.2 can result where a lower-level process suddenly reuses one of its “parent” processes. To avoid such recursive refinements, reusable processes should be grouped into several abstraction levels. Processes at a higher abstraction level should only be refined with processes from a lower abstraction level and the lowest level should only contain global and local tasks.

A second modeling error that we observed, results from the reuse of the same activity within a process model. For example, *Global Task 3* occurs twice in the refinement of *Subprocess 2*. Multiple occurrences of activities should always be carefully examined, because they can indicate that the control flow was not adequately captured. It can be argued that the decision should be placed after *Global Task 3* and that this task should only occur once in the process model. Figure 8.3 shows the corrected *Subprocess 2*.

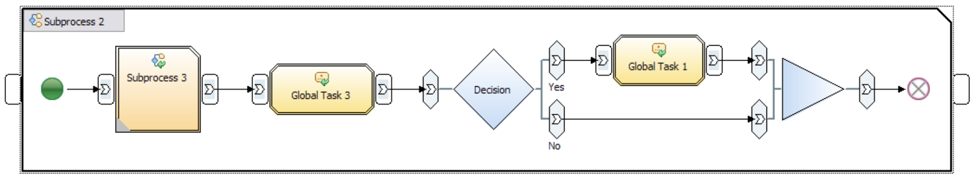


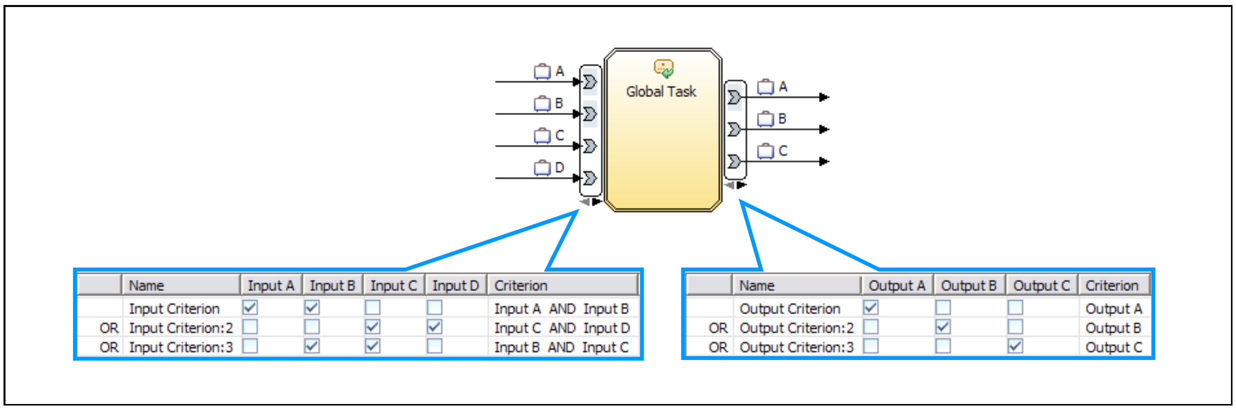
Figure 8.3: The corrected subprocess with only a single occurrence of the reused *Global Task 3*.

Multiple occurrences of an activity do not necessarily lead to a process execution error, but they can affect the readability of the model. A frequent example of redundancy occurs when users approximate iterative behavior. Instead of drawing a cycle, a decision is added and repeated activities are placed multiple times in the process model. This technique enables a business analyst to separate repeated paths in a process and it facilitates the analysis of the process model in a business scenario. However, in an implementation scenario, the alternative sequential paths do no correctly capture the behavior to implement, which makes it impossible to generate meaningful BPEL out of these models. A better solution in case of the latter scenario would be to place the repeating activities within a cycle, as we discussed it in Section 4. The cycle can be created either by using a merge followed by a decision, or by using the loop modeling elements available in WBM.

Finally, reusing activities in process models with data flow requires a careful examination of the inputs and outputs of the reusable activities. Figure 8.4 shows a *Global Task* that provides alternative input and output interfaces for reuse.

The task executes when it receives either business items A and B, or C and D, or B and C as inputs. We defined these possible combinations in three different input criteria. As an output, the task provides business item A, or B, or C. Each single business item is thus placed in a separate output criterion. The definitions of the input and output criteria are part of the activity definition: they cannot be changed in any process model that reuses this task. Only additional control flow inputs and outputs can be added to an activity after dragging and dropping it into a reusing process model.

To correctly reuse a task or subprocess, each reusing process must provide the inputs for at least one of the input criteria, which means that these inputs must be connected to other activities in the reusing process model. The reusing process should also be able to handle the possible outputs of the reused



	Name	Input A	Input B	Input C	Input D	Criterion
	Input Criterion	✓	✓	☐	☐	Input A AND Input B
OR	Input Criterion:2	☐	☐	✓	✓	Input C AND Input D
OR	Input Criterion:3	☐	✓	✓	☐	Input B AND Input C

	Name	Output A	Output B	Output C	Criterion
	Output Criterion	✓	☐	☐	Output A
OR	Output Criterion:2	☐	✓	☐	Output B
OR	Output Criterion:3	☐	☐	✓	Output C

Figure 8.4: Input and output criteria of a global task that specify alternative task interfaces for reuse.

activity. Ideally, all outputs of all output criteria should be connected. If an output cannot be connected, because it cannot be used by the reusing process, this output should be connected to an end or stop node. It must be ensured that the reusing process is able to execute despite the non-usable output. This can also require to add an additional control-flow connection from the reused activity to other activities further downstream in the process model.

In the example, the reusing process must be able to handle the alternative outputs A, B, or C. There should be subsequent process fragments in the reusing process to receive these business items as separate inputs. If a specific output occurs depending on a specific input, for example, when business item A is produced as an output only if A and B are provided as an input, then the reusing process only needs to be able to handle those outputs.

Recommendations 12 *Hierarchical models become more readable if reusable subprocesses are grouped based on identical abstraction levels and processes at one abstraction level are only refined with processes from a lower abstraction level. A hierarchical process decomposition should not contain the same process in different level unless an infinite recursive refinement of the process has to be modeled. In this case, a reachable exit branch that stops the recursive invocation should be added to the model.*

Multiple occurrences of the same reused activity should be examined for redundancy and possible improvements of the control flow of the process. When reusing an activity in a process model containing data flow, the inputs and outputs of the activity must match the possible data flows that can involve the reused activity within the reusing process model.

9 Conclusion

In this report, we investigate typical modeling errors that we extracted from hundreds of real-world process models drawn in different business process modeling tools over the last two years. The modeling errors are grouped into six common modeling scenarios. In the report, we address the following scenarios: the modeling of branching and iterative behavior, the modeling of data flow, the modeling of events and triggers, the correct termination of a process, and the reuse of activities in hierarchical process models. Each scenario is motivated by discussing an example process model containing errors. Then, the errors are generalized into anti-patterns that allow us to systematically describe incorrect modeling solutions. The correct solution is presented in form of a pattern. A summarizing recommendation concludes each scenario.

The scenarios enable readers to easily match the material provided in this report with the modeling challenges they are facing. The anti-patterns and patterns are based on a systematic approach that shows the modeling elements in combination—they make it easy for users to understand which combinations work and which fail and for what reason. For example, when discussing branching and iterative behavior, we examine the possible pairings of gateways and show that out of the eight possible combinations only three lead to correct models. In case of data-flow modeling, we provide users with a systematic approach where they evaluate whether the shared or unshared data is passed along several alternative or parallel branches.

The process anti-patterns not only help users to learn how to create correct process models, but they also provide the basis for quality assurance support that could be added to modeling tools in the future.

Acknowledgment

We would like to thank our colleagues Thomas Gschwind, Jochen Küster, Cesare Pautasso, Ksenia Ryndina, Michael Wahler, Olaf Zimmermann, and IBM practitioners for their valuable comments on draft versions of this report.

References

[1] Anti-pattern, 2007. en.wikipedia.org/wiki/anti-pattern.

[2] J. Becker, M. Rosemann, and C. von Uthmann. Guidelines of business process modeling. In W. Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *LNCS*, pages 30–49. Springer, 2000.

[3] *Business Process Modeling Notation Specification*, 2006. Version 1.0 dtc/06-02-01.

[4] T. Erl. *Service-Oriented Architecture: Concept, Technology, and Design*. Prentice Hall, 2005.

[5] A. Foerster, G. Engels, and T. Schattkowsky. Activity diagram patterns for modeling quality constraints in business processes. In *Proceedings of the MoDELS Conference*, volume 3713 of *LNCS*, pages 2–16. Springer, 2005.

[6] MID GmbH. MID Innovator, 2007.

[7] A. Selcuk Guceglioglu and O. Demirors. A process based model for measuring process quality attributes. In *12th European Conference on Software Process Improvement (EuroSPI)*, volume 3792 of *LNCS*, pages 118–129. Springer, 2005.

[8] A. Selcuk Guceglioglu and O. Demirors. Using software quality characteristics to measure process quality. In *3rd International Conference on Business Process Management*, volume 3649 of *LNCS*, pages 374–379. Springer, 2005.

[9] IBM. Rational Software Architect. IBM, 2007.

[10] J. Koehler, R. Hauser, J. Küster, K. Ryndina, J. Vanhatalo, and M. Wahler. The role of visual modeling and model transformations in business-driven development. In *Proceedings of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques*, pages 1–12. Elsevier, 2006.

[11] J. Koehler, R. Hauser, S. Sendall, and M. Wahler. Declarative techniques for model-driven business process integration. *IBM Systems Journal*, 44(1):47–65, 2005.

[12] C. Lange, B. Dubois, M. Chaudron, and S. Demeyer. An experimental investigation of UML modeling conventions. In *Proceedings of the MoDELS Conference*, volume 4199 of *LNCS*, pages 24–41. Springer, 2006.

[13] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, 1999.

[14] T. Mitra. Business-driven development. IBM developerWorks article, http://www.ibm.com/developerworks/webservices/library/ws-bdd, IBM, 2005.

[15] J. Novatnack and J. Koehler. Using patterns in the design of inter-organisational systems - an experience report. In *Proceedings of the Workshop on Modeling Inter-Organisational Systems*, volume 3292 of *LNCS*, pages 444–455. Springer, 2004.

[16] Object Management Group (OMG). *Unified Modeling Language: Superstructure*, 2005. Version 2.0 formal/05-07-04.

[17] M. Rosemann. *Komplexitätsmanagement in Prozessmodellen*. Gabler, 1996.

[18] W. Sadiq and M. Orlowska. Applying graph reduction techniques for identifying structural conflicts in process models. In *Proceedings of the 11th International Conference on Advanced Information Systems Engineering (CAiSE '99)*, volume 1626 of *LNCS*, pages 195–209. Springer, 1999.

[19] SAP Research. Maestro Business Process Modeling Tool, 2007.

[20] A. W. Scheer, F. Abolhassan, W. Jost, and M. Kirchner. *Business Process Excellence - ARIS in Practice*. Springer, 2002.

[21] IDS Scheer. Aris Business Architect, 2007.

[22] BOC Information Systems. ADONIS Business Process Management, 2007.

[23] W.M.P. van der Aalst. Challenges in business process management: Verification of business processes using Petri nets. *Bulletin of the EATCS*, 80:174–198, 2003.

[24] W.M.P. van der Aalst. Business process management demystified: A tutorial on models, systems and standards for workflow management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 1–65. Springer, 2004.

[25] 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.

[26] W.M.P. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.