9 Patterns of Process Modeling

WIL M.P. VAN DER AALST[1,2], ARTHUR H.M. TER HOFSTEDE[2], MARLON DUMAS[2]

[1] Eindhoven University of Technology, The Netherlands
[2] Queensland University of Technology, Australia

9.1 INTRODUCTION

The previous chapters have presented different languages and approaches to process modeling. In this chapter, we review some issues in process modeling from a more language-independent perspective. To this end, we rely on the concept of pattern: an "abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts" [18]. The use of patterns is a proven practice in the context of object-oriented design, as evidenced by the impact made by the design patterns of Gamma et al. [10].

Process Aware Information Systems (PAISs) address a number of perspectives. Jablonski and Bussler [11] identify several such perspectives in the context of workflow management. These include the process perspective (describing the control-flow), organization perspective (structuring of resources), data/information perspective (to structure data elements), operation perspective (to describe the atomic process elements) and integration perspective (to "glue" things together). [1] In a typical workflow management system, the process perspective is described in terms of some graphical model, e.g. a variant of Petri nets (see Chapter 7), the organization perspective is described by specifying and populating roles and organizational units, the data/information perspective is described by associating data elements to workflow instances (these may be typed and have a scope), the operation perspective is described by some scripting language used to launch external applications, and the integration perspective is described by some hierarchy of processes and activities. In principle, it is possible to define design patterns for each of these perspectives. However, the focus of this chapter is on patterns restricted to the process (i.e. control-flow) perspective. This perspective is the best understood as well as the dominant perspective of workflow. Patterns for the data perspective have been reported in [20],

[1] Note that in [11] different terms are used.

while patterns for e.g. the resource perspective are currently under investigation. In addition it is worthwhile mentioning that the control-flow patterns do not pay too much attention to issues related to exception handling. A systematic treatment of exception handling is planned for however, though whether this will take the form of a collection of patterns remains open at this stage.

The control-flow perspective is concerned with enforcing control-flow dependencies between tasks[2], e.g. sometimes tasks need to be performed in order, sometimes they can be performed in parallel, sometimes a choice needs to be made as to which task to perform, etc. There is an abundance of approaches towards the specification of control-flow in PAISs in general, and workflow management systems in particular. Many commercial workflow management systems and academic prototypes use languages with fundamental differences. Concepts with similar names may have significantly different behavior, different languages may impose different restrictions (e.g. with respect to cycles), and some concepts are supported by a select number of languages only. The reader is referred to [12, 13, 14] for a fundamental discussion of some of these issues. This work identifies a number of classes of workflow languages, which are abstractions of approaches used in practice, and examines their relative expressive power. The workflow patterns initiative took a more pragmatic approach focusing on suitability.

The workflow patterns initiative, which this chapter reviews, started in 1999. It aimed at providing a systematic and practical approach to dealing with the diversity of languages for control-flow specification. The initiative took the state-of-the-art in workflow management systems as a starting point and documented a collection of 20 patterns predominantly derived from constructs supported by these systems. The patterns provided abstractions of these constructs as they were presented in a language independent format. The patterns consist of a description of the essence of the control-flow dependency to be captured, possible synonyms, examples of concrete business scenarios requiring the control-flow dependency, and for the more complex ones, typical realization problems and (partial) solutions to these problems.

There are a number of applications of the workflow patterns. The patterns can be used for the selection of a workflow management system. In that case one would analyze the problem domain in the context of which the future workflow management system is to be used, i.e. analyze the needs in terms of required support for various workflow patterns and subsequently match the requirements with the capabilities of various workflow management systems (this could be termed a suitability analysis). Additionally, the patterns can be used for benchmarking purposes, examining relative strengths and weaknesses of workflow products. Such examinations may be the basis for language development and adaptations of workflow management systems. Another use of the patterns can be found in the context where a particular workflow tool is prescribed and certain patterns need to be captured. Here the workflow patterns collection acts as a resource for descriptions of typical workarounds and realization approaches for patterns in different workflow systems.

[2]In this chapter, the terms “task” and “activity” are used interchangeably.

The first paper related to the workflow patterns initiative[3] appeared in the CoopIS conference in 2000 (see [2]). The main paper appeared in 2003 in the Distributed and Parallel Databases Journal (see [5]). Apart from a description of the complete set of patterns, this latter paper contains an analysis of 13 commercial workflow management systems and two academic prototypes in terms of their support for the patterns. The patterns have been used for analyses of UML Activity Diagrams version 1.4 (see [8]), BML (Business Modeling Language) an approach used in the context of Enterprise Application Integration (see [24]), and various approaches and proposed standards in the area of web service composition such as BPEL4WS (see [23] for this evaluation and Chapter 14 for an introduction to BPEL4WS). The workflow patterns formed the starting point for the development of YAWL[4] (Yet Another Workflow Language). This language extends Petri nets with constructs for dealing with some of the patterns in a more straightforward manner. Though based on Petri nets, its formal semantics is described as a transition system (see [4]) and YAWL should not be seen as a collection of macros defined on top of Petri nets. A first description of the design and implementation of the YAWL environment can be found in [1]. In this chapter the YAWL notation will be used to explain various patterns.

The goal of this chapter is to take an in-depth look at a selection of the patterns as presented in [5] from a more didactic perspective. The organization of this chapter is as follows. A classification of the patterns is discussed which is followed by a detailed discussion of a selection of patterns organized according to this classification. The chapter concludes with a brief summary and outlook. Note that whenever products are referred to in this chapter their version corresponds to the version which was used for the evaluation in [5] unless stated otherwise.

9.2 CLASSIFICATION OF PATTERNS

As mentioned earlier, the patterns initiative has led to a set of 20 control-flow patterns. These patterns range from very simple patterns such as sequential routing (Pattern 1) to complex patterns involving complex synchronizations such as the discriminator pattern (Pattern 9). These patterns can be classified into six categories:

1. *Basic control-flow patterns.* These are the basic constructs present in most workflow languages to model sequential, parallel and conditional routing.

2. *Advanced branching and synchronization patterns.* These patterns transcend the basic patterns to allow for more advanced types of splitting and joining behavior. An example is the Synchronizing merge (Pattern 7) which behaves like an AND-join, XOR-join, or combination thereof, depending on the context.

3. *Structural patterns.* In programming languages a block structure which clearly identifies entry and exit points is quite natural. In graphical languages allowing

[3] www.workflowpatterns.com
[4] www.yawl-system.com

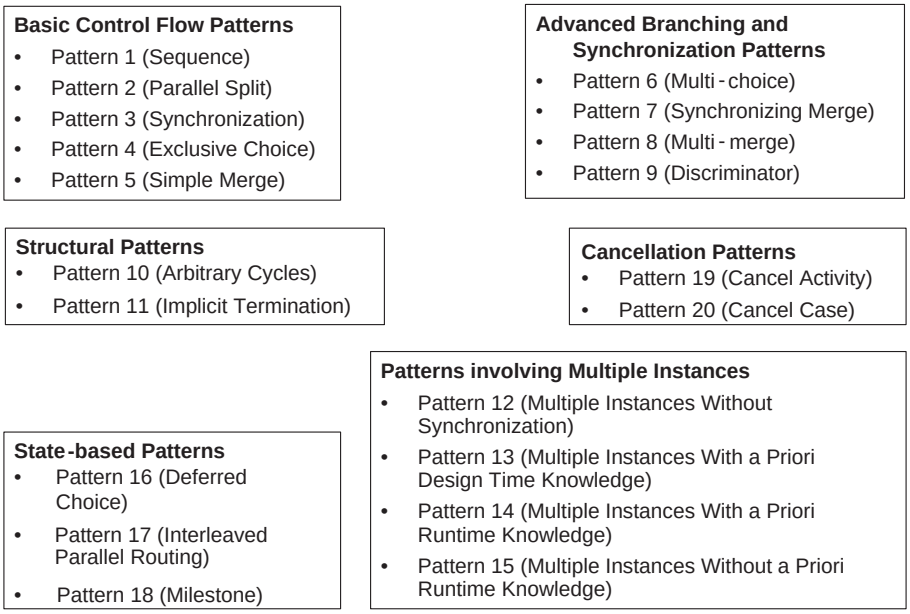

Fig. 9.1 Overview of the 20 workflow patterns described in [5].

for parallelism such a requirement is often considered to be too restrictive. Therefore, we have identified patterns that allow for a less rigid structure.

4. *Patterns involving multiple instances.* Within the context of a single case (i.e. workflow instance) sometimes parts of the process need to be instantiated multiple times, e.g. within the context of an insurance claim, multiple witness statements need to be processed.

5. *State-based patterns.* Typical workflow systems focus only on activities and events and not on states. This limits the expressiveness of the workflow language because it is not possible to have state dependent patterns such as the Milestone pattern (Pattern 18).

6. *Cancelation patterns.* The occurrence of an event (e.g. a customer canceling an order) may lead to the cancelation of activities. In some scenarios such events can even cause the withdrawal of the whole case.

Figure 9.1 shows an overview of the 20 patterns grouped into the six categories. This classification is used in the next section to highlight some of the patterns.

9.3 EXAMPLES OF CONTROL-FLOW PATTERNS

This section presents a selection of the various control-flow patterns using the classification shown in Figure 9.1.

9.3.1 Basic control-flow patterns

The basic control flow patterns essentially correspond to control-flow constructs as described by the Workflow Management Coalition[5] (see e.g. [9] or [22]). These patterns are typically supported by workflow management systems and as such do not cause any specific realization difficulties. It should be pointed out though that the behavior of the corresponding constructs in these systems can be fundamentally different. In this section the sequence pattern is discussed in terms of the format used for capturing patterns, while the other four patterns are discussed in a less structured manner.

Pattern 1 (Sequence)
Description An activity should await the completion of another activity within the same case before it can be scheduled.
Synonyms Sequential routing, serial routing
Examples

- After the expenditure is approved, the order can be placed.

- The activity *select_winner* is followed by the activity *notify_outcome*.

Implementation

- This pattern captures direct causal connections between activities and is supported by all workflow management systems. Graphically, this pattern is typically represented through a directed arc without an associated condition.

There are four other basic control-flow patterns, two of which correspond to splits and two of which correspond to joins.

The *XOR-split* corresponds to the notion of an *Exclusive choice* (Pattern 4). Out of two or more outgoing branches, one branch is chosen. Such a choice is typically determined by workflow control data or input provided by users of the system. In some systems (e.g. Staffware) there is explicit support for XOR-splits, while in some other systems (e.g. MQSeries/Worfklow) the designer has to guarantee that only one outgoing branch will be chosen at runtime by providing mutually exclusive conditions. In the YAWL environment, conditions specified for outgoing arcs of an XOR-split may overlap. In case multiple conditions evaluate to true, the arc with the highest preference (which is specified at design time) is selected. If all conditions evaluate to false, the default arc is chosen. This solution is similar to Eastman's solution (see [21], p. 144-145) where a rule list can be specified for an activity, and these rules are processed after completion of that activity. Among others, this list may contain rules for passing control to subsequent activities. Control is passed to the first activity occurring in such a rule, whose associated condition evaluates to true. A default rule can be specified which does not have an associated condition and

[5] www.wfmc.org

therefore should be last in such a list ([21], p. 145). As an example of an XOR-split consider the case where purchase requests exceeding $10,000 are to be approved by head office, while purchase requests not exceeding this amount of money can be approved by the regional offices.

The converse of the XOR-split is the *XOR-join* or *Simple merge* (Pattern 5). An XOR-join is enabled when one of its preceding branches completes. The definition by the Workflow Management Coalition (WfMC) requires that the XOR-join (called OR-join by the WfMC) is not preceded by parallelism, i.e. no two or more preceding branches of the XOR-join run in parallel at any point in time. The pattern incorporates this requirement, which can be seen as a context assumption. Without this assumption substantial differences between various workflow products would become apparent, but we will treat this as a different pattern, the multi-merge, to be discussed in the next section. In some cases, XOR-joins should have corresponding XOR-splits (e.g. Visual WorkFlo), which, combined with other similar restrictions, may guarantee that no parallelism occurs in branches preceding XOR-joins. Such structured workflows are discussed in [12, 14]. As an example of an XOR-join consider the activity *report_outcome* which is to be executed after activity *finalize_rejection* or activity *finalize_approval* completes. It is assumed that these two latter activities never run in parallel.

The *AND-split* can be used to initiate parallel execution of two or more branches (Pattern 2). It should be remarked that the description in [5], which was adapted from the original formulation by the WfMC, left open the possibility that no true parallelism takes place, so that these branches could be executed in an interleaved manner. This interpretation is convenient, as it can be used in contexts where other constraints (e.g. on resources) do not permit activities of different branches to be executed at the same time, but where the specification should be flexible enough to allow the execution environment to decide itself which activity of which branch should be scheduled next (hence one should not be forced to make an arbitrary decision at design time). An example of an AND-split could be in the context of an application process where after short-listing of candidates, referee reports need to be obtained and interviews need to be held. For a particular candidate, these activities could be done in parallel or in any order. In terms of implementation, sometimes the AND-join is supported in an implicit manner (e.g. MQSeries/Workflow), where conditions are required to be specified for all outgoing branches. In those cases the AND-join can be realized by specifying the condition represented by the Boolean constant *true* for all these branches.

The *AND-join* is the converse of the AND-split and synchronizes its incoming branches (Pattern 3). All incoming branches of an AND-join need to be completed before the AND-join can proceed the flow. E.g. a decision for a particular candidate can only be made once their referee reports have been received and they have been interviewed. Again, there is a context assumption for this pattern. It should not be possible that a branch signals its completion more than once before all other branches have completed. So the AND-join only needs to remember whether a particular branch has completed or not. Note that it is possible that a completion signal is never received for a particular branch in which case the AND-join causes a deadlock.

Figure 9.2 provides an overview of the graphical constructs in YAWL that can be used to capture the basic control-flow patterns. In this figure, boxes denote "tasks" while circles denote "conditions". These correspond to the notions of transition and place in Petri nets respectively (See Chapter 7).[6] These YAWL constructs do not require the context assumptions of some of the patterns presented in this section and as such have a broader interpretation than these patterns (e.g. the simple merge can be realized by the YAWL XOR-join, but this XOR-join also realizes the Multi-merge pattern discussed in the next section).

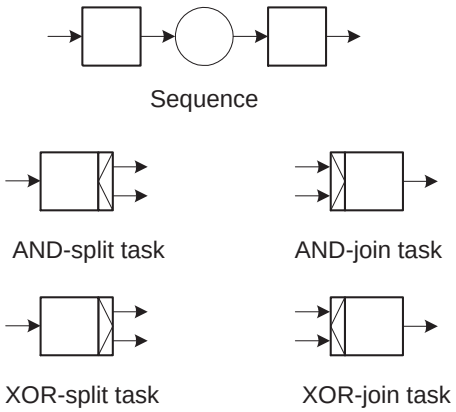

Fig. 9.2 Some basic symbols used in YAWL.

9.3.2 Advanced branching and synchronization patterns

The patterns presented in this section deal with less straightforward split and join behavior and, though not uncommon in practice, pose more difficulties in terms of their support by contemporary workflow management systems. The Multi-choice (Pattern 6) and the Synchronizing merge (Pattern 7) will be looked at in-depth, the other patterns will be discussed more briefly.

While only one outgoing branch is chosen in case of an XOR-split, the *OR-split* or *Multi-choice* allows a choice for an arbitrary number of branches.

Pattern 6 (Multi-choice)
Description Out of several branches, a number of branches are chosen based on user input or data accessible by the workflow management system.
Synonyms Conditional routing, selection, OR-split.
Examples

[6]Note that in the YAWL representation of the Sequence (which appears in the figure), the condition does not need to be explicitly shown.

- After the execution of activity *determine_teaching_evaluation*, execution of activity *organize_student_evaluation* may commence as well as execution of activity *organize_peer_review*. At least one of these two activities is executed, possibly both.

Problem Workflow management systems that allow for the specification of conditions on transitions support this pattern directly. Sometimes however the multi-merge needs to be realized in terms of the basic patterns (e.g. Staffware supports AND-splits and XOR-splits, but nothing "in between").

Implementation

- As mentioned above, this pattern is directly supported by systems that allow for conditions to be specified for transitions (as is e.g. the case for Verve, Fortté Conductor, and MQSeries/Workflow). As stated in the previous section, such systems provide implicit support for XOR-splits and AND-splits. However, their approach also allows for splits that are neither (selection of more than one branch, but not all). The multi-merge may be considered a generalization of the XOR-split and the AND-split.

- An OR-split in YAWL is shown in Figure 9.3. It should be noted that in the YAWL environment, at least one outgoing transition needs to be chosen, which makes its OR-split slightly less general than the pattern. In YAWL, the selection of at least one branch is guaranteed by the specification of a default branch which is chosen if none of the conditions evaluate to true (including the condition associated with the default branch!).

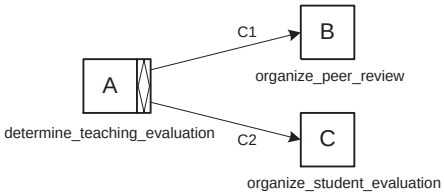

Fig. 9.3 Multi-choice in YAWL.

- For those languages that only support the basic XOR-splits and AND-splits there are two solutions:

  - Transform the n-way multi-choice into an AND-split followed by n binary XOR-splits, each of which checking whether the condition of the corresponding branch in the multi-choice is true or not. If a condition evaluates to true, the corresponding activity needs to be executed, otherwise no action is required.
  - Transform the n-way multi-choice into an XOR-split with 2^n outgoing branches. Each of these outgoing branches corresponds to a particular subset of outgoing branches that may be chosen as part of the multi-choice

(AND-splits would be used for those subsets that consist of at least two outgoing branches). The associated condition should capture that the corresponding conditions of these branches are true, but that this doesn't hold for any of the conditions associated with the other branches (note that this solution indeed guarantees mutual exclusion of the outgoing branches of the XOR-split). This solution is exponential in terms of the number of outgoing transitions as opposed to the previous solution.

These solutions are illustrated in YAWL in Figure 9.4, where it should be noted that contrary to YAWL's semantics we assume that the multi-choice of Figure 9.3 also allows none of the branches to be executed (to be in line with the description of the pattern).

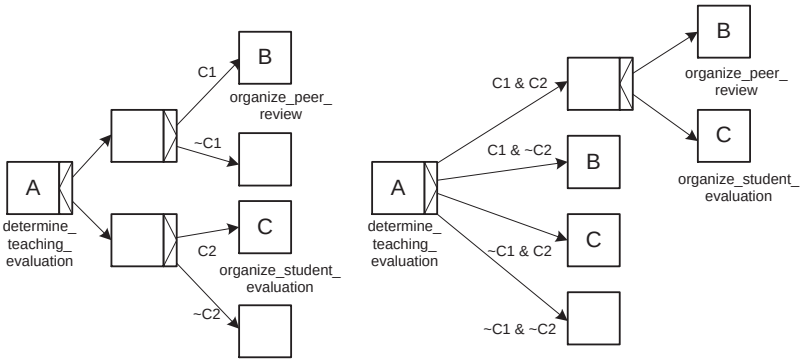

Fig. 9.4 Expanding a multi-choice in terms of simple choice and parallel split (illustrated in YAWL). "$\sim$ C" stands for "not C".

While the multi-choice does not cause too many problems in contemporary workflow management systems, the same cannot be said for the so-called *Synchronizing merge* (Pattern 7) which in a structured context can be seen as its converse. Consider the YAWL schema of Figure 9.5. After activity A (*determine_teaching_evaluation*) either B (*organize_peer_review*) or C (*organize_student_evaluation*) or both B and C will be executed (note that as mentioned before, in YAWL the multi-choice has to choose at least one of the outgoing branches). The synchronization prior to the execution of activity D (*interpret_results*) should now be such that only active threads should be waited upon. In particular, if activity B is not triggered after the OR-split, then the OR-join should not wait for it and same for activity A. This is achieved through the use of an *OR-join* in YAWL. The use of an AND-join here may lead to a deadlock in case either B or C was chosen (but not both), while the use of a synchronization construct which executes D upon completion of any of the incoming branches (multi-merge) may lead to executing this activity twice in case both B and C were chosen.

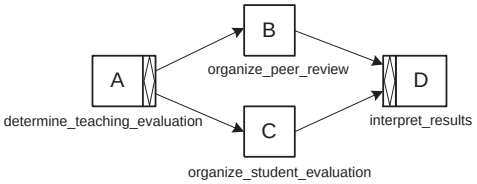

Fig. 9.5 Illustration of the synchronizing merge in YAWL.

In the description of the synchronizing merge a more general approach will be taken, in line with the formalization in YAWL, than the one described in [5].

Pattern 7 (Synchronizing Merge)
Description A form of synchronization where execution can proceed if and only if one of the incoming branches has completed and from the current state of the workflow it is not possible to reach a state where any of the other branches has completed.
Synonyms Synchronizing join, OR-join.
Examples

- Consider again the example presented in Pattern 6 (Multi-choice). After activities *organize_student_evaluation* and *organize_peer_review* have finished, activity *interpret_results* could be scheduled. This activity should only await completion of those activities that were actually executed and itself be performed once.

Problem The main challenge of achieving this form of synchronization is to be able to determine when more completions of incoming branches are to be expected. In the general case, this may require an expensive state analysis.
Implementation

- In workflow systems such as MQSeries/Workflow and InConcert the synchronizing merge is supported directly because of the evaluation strategy used. In MQSeries/Workflow activities have to await signals from all incoming branches. Such a signal may indicate that a certain branch completed and that the associated condition evaluated to true, or it indicates that a certain branch was bypassed or that the associated condition evaluated to false. Depending on the particular combination of signals received and the evaluation of the join condition the activity is or is not executed and a corresponding signal is propagated. In InConcert, activities just await signals from all incoming branches and the evaluation of a precondition (not the value of these signals) will determine whether they themselves will be executed or not. In either case, no deadlock will occur as neither MQSeries/Workflow nor InConcert allows cycles.

- The interpretation of the OR-join in YAWL (as formalized in [4]) is such that it is enabled if and only if an incoming branch has signaled completion and from the current state it is not possible to reach a state (without executing any

OR-join) where another incoming branch signals completion. While this can handle workflows of a structured nature it can also handle workflows such as the one displayed in Figure 9.6. As a possible scenario consider the situation where after completion of activity A both activities B and C are scheduled. If activity C completes, and activity B has not completed then activity D can not be executed as it is possible that activity F will be chosen after completion of B. In this case, if after completion of activity B, activity E is chosen, activity D can be scheduled for execution as it is not possible to reach a state where activity F will be scheduled. So the OR-join guarantees that activity D has to await completion of activity C if it was scheduled, and if activity B was scheduled, activity D has to at least await the outcome of the decision after completion of activity B. If activity E was subsequently chosen it does not need to wait for completion of activity F, but if activity F was chosen it will have to await completion of that activity. Current work with respect to the OR-join in YAWL is reconsidering the treatment of other OR-joins in the reachability analysis required for determining whether a certain OR-join is enabled and is examining algorithmic solutions for this analysis.

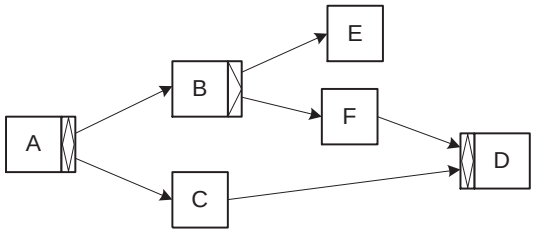

Fig. 9.6 Another illustration of the synchronizing merge in YAWL.

- As it is stated in [21] (p. 109) that "non-parallel work items routed to Join worksteps bypass Join Processing" an XOR-split followed by an AND-join in Eastman does not necessarily lead to a deadlock. Hence joins have information about the number of active threads to be expected. The situation captured by the YAWL workflow in Figure 9.5 would not cause any problems in Eastman as the OR-join would know whether to expect parallel execution of B and C or not. While this solution works fine in a structured context where information about active threads initiated after a split can be passed on to a corresponding join, this does not work so well in a context where workflows are not fully structured. Consider again the YAWL workflow depicted in Figure 9.6. In Eastman this specification leads to a deadlock if B was chosen and after completion of B a choice for activity E was made. The OR-join would then keep waiting for the completion of activity F.

- In the context of EPCs (see Chapter 6), there exists a body of research examining possible interpretations of the OR-join [3, 7, 15, 19].

- In case there is no direct support for the OR-join, it may require some work to capture its behavior (and without the use of the data perspective and under a certain notion of equivalence, it may not always be possible, see [13, 12]). In a structured context a multi-choice could be replaced as indicated in the two solutions in its pattern description and synchronization can then be achieved in a straightforward manner.

The *Multi-merge* (Pattern 8) does not make the context assumption specified for the XOR-join in the previous section. It will execute the activity involved as many times as its incoming branches signal completion. This interpretation allows these incoming branches to be executing in parallel. The YAWL XOR-join corresponds to the multi-merge.

The *Discriminator*[7] (Pattern 9) provides a form of synchronization for an activity where out of a number of incoming branches executing in parallel, the first branch to complete initiates the activity. When the other branches complete they do not cause another invocation of the activity. After all branches have completed the activity is ready to be triggered again (in order for it to be usable in the context of loops). In YAWL, one of the ways to capture the discriminator involves the usage of cancelation regions (cf. [4]). The discriminator is specified with a multi-merge and a cancelation region encompassing the incoming branches of the activity. In this realization, the first branch to complete starts the activity involved, which then cancels the other executing incoming branches. This is not in exact conformance with the original definition of the pattern (as it actually cancels the other branches), but this choice is motivated by the fact that it is clear in this approach what the region is that is in the sphere of the discriminator giving it a clearer semantics. The discriminator is a special case of the n-out-of-m join (sometimes referred to as partial join [6]) as it corresponds to a 1-out-of-m join.

9.3.3 Structural patterns

This section briefly examines two so-called structural patterns: arbitrary cycles and implicit termination. Structural patterns deal with syntactic restrictions that some languages impose on workflow specifications.

Some workflow systems only allow the specification of loops with unique entry and exit points. *Arbitrary cycles*, where there are multiple ways of exiting from the loop or multiple ways of entering the loop, are not allowed. Sometimes this is enforced in an explicit manner, e.g. in languages that are structured (see e.g. [14]), while sometimes the restriction comes about through the fact that iterative behavior can only be specified through postconditions on decompositions (as e.g. in MQSeries/Workflow). The process specified in the decomposition is to be repeated till the postcondition evaluates to true. In the case of MQSeries/Workflow, this more implicit way of specifying loops is a direct consequence of the evaluation strategy used, where incoming

[7] The term Discriminator originates from Verve Workflow.

signals are expected from all incoming branches of an activity. Obviously, cycles in a specification would then cause a deadlock. YAWL allows for the specification of arbitrary cycles. In [12, 14], expressiveness issues in relation to structured workflows are investigated. It is shown that not all arbitrary cycles can be converted to structured cycles (in the context of a given equivalence notion and without considering the data perspective).

At least two different termination strategies can be distinguished for workflows. In one approach, a workflow execution is considered completed when no activity can be scheduled anymore (and the workflow is not in a deadlock). This is referred to as *implicit termination* (e.g. supported by MQSeries/Workflow, Staffware). In the other approach the workflow is considered completed if a designated end point is reached. Though other activities may still be executing, they are terminated when this happens. While the two approaches are different, in some cases workflows following one approach may be converted to workflows conforming to the other approach. In [13] it is shown how so-called standard workflows which do not contain a deadlock and do not have multiple concurrent instances of the same activity at any stage, can be transformed into equivalent standard workflows with a unique ending point so that when this ending point is reached, no other part of the workflow is still active. YAWL does not support implicit termination as to force workflow designers to carefully think about workflow termination.

9.3.4 Patterns involving multiple instances

The patterns in this section involve a phenomenon that we will refer to as *multiple instances*. As an example, consider the reviewing process of a paper for a conference. Typically, there are multiple reviews for one paper and some activities are at the level of the whole paper (e.g. accept/reject) while others are at the level of a single review (e.g. send paper to reviewer). This means that inside a case (i.e. the workflow instance, in this example a paper) there are sub-instances (i.e. the reviews) that need to be dealt with in parallel (i.e. in parallel multiple reviewers may be reviewing the same paper). Multiple-instance patterns are concerned with the embedding of sub-instances in cases (i.e. workflow instances). From a theoretical point of view the concept is relatively simple and corresponds to multiple threads of execution referring to a shared definition. From a practical point of view it means that an activity in a workflow graph can have more than one running, active instance at the same time. As we will see, such behavior may be required in certain situations. The fundamental problem with the implementation of these patterns is that due to design constraints and lack of anticipation for this requirement most of the workflow engines do not allow for more than one instance of the same activity to be active at the same time (in the context of a single case).

When considering multiple instances there are two types of requirements. The first requirements has to do with the ability to launch multiple instances of an activity or a subprocess. The second requirement has to do with the ability to synchronize these instances and continue after all instances have been handled. Each of the

patterns needs to satisfy the first requirement. However, the second requirement may be dropped by assuming that no synchronization of the instances launched is needed.

If the instances need to be synchronized, the number of instances is highly relevant. If this number is fixed and known at design time, then synchronization is rather straightforward. If, however, the number of instances is determined at run-time or may even change while handling the instances, synchronization becomes very difficult. Therefore, Figure 9.1 names three patterns with synchronization. If no synchronization is needed, the number of instances is less relevant: Any facility to create instances within the context of a case will do. Therefore, Figure 9.1 names only one pattern for multiple instances without synchronization.

In this section we highlight only one of the four multiple instance patterns. This is the most complex of these patterns since it requires synchronization while the number of instances can even vary at runtime.

Pattern 15 (Multiple Instances Without a Priori Runtime Knowledge)
Description For one case an activity is enabled multiple times. The number of instances of a given activity for a given case is not known during design time, nor is it known at any stage during runtime, before the instances of that activity have to be created. Once all instances are completed some other activity needs to be started. It is important to note that even while some of the instances are being executed or already completed, new ones can be created.
Examples

- Consider the reviewing process of a paper for a conference where there are are multiple reviews for one paper. The number of reviewers may be fixed initially, say 3, but may be increased when there are conflicting reviews or missing reviews. For example, initially three reviewers are appointed to review a paper. However, halfway the reviewing period a reviewer indicates that he will not be able to complete the review. As a result a fourth reviewer (i.e. the fourth instance) is appointed. At the end of the review period only two reviews are returned. Moreover, the two reviews are conflicting (strong accept versus strong reject). As a result, the PC chair appoints a fifth reviewer (i.e. the fifth instance).

- The requisition of 100 computers involves an unknown number of deliveries. The number of computers per delivery is unknown and therefore the total number of deliveries is not known in advance. After each delivery, it can be determined whether a next delivery is to come by comparing the total number of delivered goods so far with the number of the goods requested. After processing all deliveries, the requisition has to be closed.

- For the processing of an insurance claim, zero or more eyewitness reports should be handled. The number of eyewitness reports may vary. Even when processing eyewitness reports for a given insurance claim, new eyewitnesses may surface and the number of instances may change.

Problem Some workflow engines provide support for generating multiple instances only if the number of instances is known at some stage of the process. This can be

compared to a “for” loop in procedural languages. However, these constructs are of no help to processes requiring “while” loop functionality. Note that the comparison with the “while” construct may be misleading since all instances may run in parallel.

Implementation

- YAWL directly supports multiple instances. Figure 9.7 shows a process where there may be multiple witnesses processed in parallel. Composite activity B (*process_witness_statements*) consists of three steps (D, E, F) which are executed for each witness. Figure 9.7 does not show that the number of instances of B may be changed at any point in time, i.e. even when the processing of witness statements has already started. This is a setting of B.

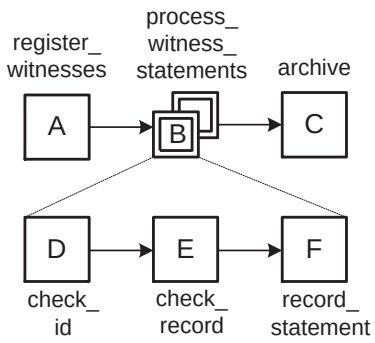

Fig. 9.7 Illustration of the multiple instances pattern in YAWL.

- FLOWer (see Chapter 19) is one of the few commercial systems directly supporting this pattern. In FLOWer it is possible to have dynamic subplans. The number of instances of each subplan can be changed at any time (unless specified otherwise).

- If the pattern is not supported directly, typical implementation strategies involve a counter indicating the number of active instances. The counter is incremented each time an instance is created and decremented each time an instance is completed. If, after activation, the counter returns to 0, then the construct completes and the flow continues. Consider for example, Figure 9.7. In activity A the counter is set to some value indicating the initial number of witnesses. While executing instances of composite activity B, new instances may be created. For each created instance, the counter is incremented by 1. Each time F is executed for some instance (i.e. B completes), the counter is decremented. If the value of the counter equals 0, no more witnesses can be added and C gets enabled. Note that the counter only takes care of the synchronization problem. In addition to the counter, the implementation should allow for multiple instances running in parallel.

Multiple instances are not only interesting from a control-flow point of view. Note that each instance will have its own data elements but at the same time it may be necessary to aggregate data. For example, in Figure 9.7 each witness may have an address, i.e. each instance of B has a case attribute indicating the address of the witness. However, in C it may be interesting to determine the number of witnesses living at the same address. This implies that it is possible to query the data of each instance to do some calculations.

9.3.5 State-based patterns

In [17] supporting evidence from a number of sources is collected with respect to the time spent waiting as a percentage of the total execution time (i.e. cycle/flow time) averaged over workflow instances in areas dealing with insurance and pension claims, and tax returns. In the five sources mentioned, this average percentage is at least 95% (and in three of these sources at least 99%), which implies that workflow instances are typically in a state awaiting processing rather than being processed. Many computer scientists, however, seem to have a frame of mind, typically derived from programming, where the notion of state is interpreted in a narrower fashion and is essentially reduced to the concept of data or a position in the queue of some activity. As this section will illustrate, there are real differences between work processes and computing and there are business scenarios where an explicit notion of state is required.

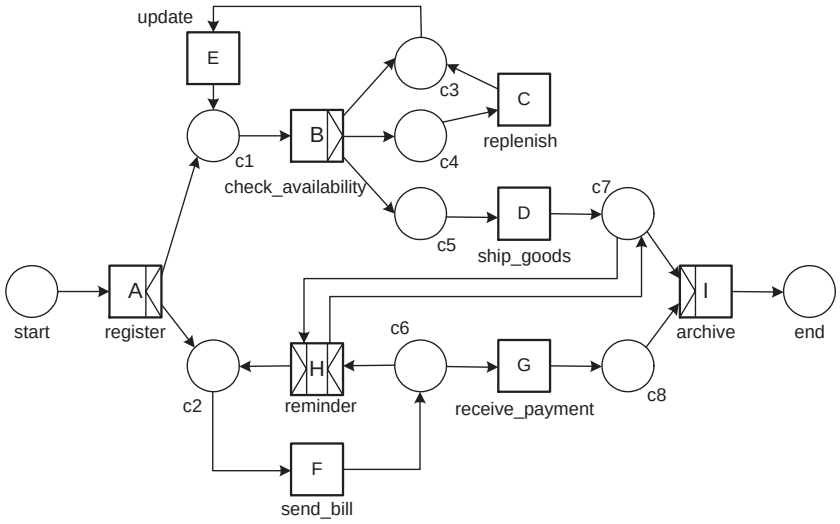

Fig. 9.8 A example illustrating state-based patterns.

To illustrate two of the state-based patterns, consider Figure 9.8. This is again a YAWL diagram. However, in contrast to the earlier diagrams, Figure 9.8 explicitly shows the states in-between activities. Note that YAWL uses a Petri-net-like notation to model states (i.e. places). The initial state of the case (i.e. process instance) is modeled by *start*. The final state is modeled by *end*. If there is a token in *start*, the first activity A can be executed. The last activity I will put a token in *end*. In-between A (register) and I (archive) two parallel processes are executed. The upper part models the logistical subprocess while the lower parts models the financial subprocess. In the logistical subprocess there is an Exclusive choice (Pattern 4) modeled by B. If the ordered goods are available, B wil put a token in $c5$. If the ordered goods are not available, a replenishment order is planned (token in place *c4*) or the missing goods have already been ordered (token in place *c3*). In the financial subprocess (i.e. lower part) there is also a choice. After sending the bill (F) there is a choice between the decision to send a reminder (activity H) and the receipt of the payment (activity G). Note that this decision is not made by F, i.e. after completing activity F the choice between H and G is not fixed but depends on external circumstances, e.g. H may be triggered by a clock (e.g. after four weeks) while G is triggered by the customer actually paying for the ordered goods. Note that the choice modeled by *c6* is different from the choice modeled by B: in *c6* there is a "race" between two activities H and G while after executing B the next activity is fixed by putting a token in *c3* (E), *c4* (C), or *c5* (D). As indicated, the construct involving B corresponds to the traditional *Exclusive choice* (Pattern 4) supported by most systems and languages. The construct involving *c6* corresponds to the *Deferred choice* (Pattern 16) described in this section. Figure 9.8 also shows another state-based pattern: the *milestone* (Pattern 18). This is the construct involving H and *c7*. Note that H is an AND-split/AND-join and therefore it can only occur if there is a token in *c7*. This implies that it is only possible to send a reminder if the goods have been shipped. The purpose of the milestone pattern is to be able to test the state in another parallel branch. If H would always occur exactly once, this construct would not be needed, i.e. the two arcs representing the milestone could be replaced by a new place connecting D to H. However, for some cases H is not executed at all (i.e. the customer is eager to pay) while for other cases H is executed multiple times (i.e. the customer is reluctant to pay). Therefore, this alternative solution does not work properly. Note that in this case the deferred choice and milestone are connected. In general this is not the case since both patterns can occur independently. There is a third state-based pattern (Pattern 17: interleaved parallel routing). The pattern is used to enforce mutual exclusion without enforcing a fixed order. A discussion of this pattern is beyond the scope of this chapter. Instead we restrict our attention to the deferred choice.

Pattern 16 (Deferred Choice)
Description A point in the workflow process where one of several branches is chosen. In contrast to the XOR-split, the choice is not made explicitly (e.g. based on data or a decision) but several alternatives are offered to the environment. However, in contrast to the OR-split, only one of the alternatives is executed. This means that once the environment activates one of the branches the other alternative branches are

withdrawn. It is important to note that the choice is delayed until the processing in one of the alternative branches is actually started, i.e. the moment of choice is as late as possible.

Synonyms External choice, implicit choice, deferred XOR-split.

Examples

- See the YAWL diagram shown in Figure 9.8. After sending the bill (F) there is a choice between the decision to send a reminder (activity H) and the receipt of the payment (activity G). This decision is not made by F but is resolved by the "race" between H and G, i.e. a race between a time trigger (end of a four week period) and an external trigger (the receipt of the payment).

- At certain points during the processing of insurance claims, quality assurance audits are undertaken at random by a unit external to those processing the claim. The occurrence of an audit depends on the availability of resources to undertake the audit, and not on any knowledge related to the insurance claim. Deferred choices can be used at points where an audit might be undertaken. The choice is then between the audit and the next activity in the processing chain. The activity capturing the audit triggers the next activity to preserve the processing chain.

Problem Many workflow management systems support the XOR-split described in Pattern 4 (Exclusive choice) but do not support the deferred choice. Since both types of choices are desirable (see examples), the absence of the deferred choice is a real problem. The essence of the problem is the moment of choice as illustrated by Figure 9.9. In Figure 9.9(a) the choice is as late a possible (i.e. when B or C occurs) while in Figure 9.9(b) the choice is resolved when completing A.

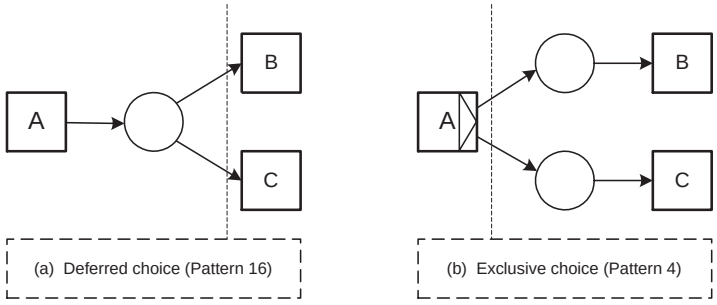

Fig. 9.9 The moment of choice in the Patterns 4 (Exclusive choice) and 16 (Deferred choice).

Implementation

- COSA is one of the few systems that directly supports the deferred choice. Since COSA is based on Petri nets it is possible to model implicit choices as indicated in Figure 9.9(a). YAWL is also based on Petri nets and therefore also supports the deferred choice. Some systems offer partial support for this pattern by offering special constructs for a deferred choice between a user action and a time out (e.g. Staffware) or two user actions (e.g. FLOWer).

- Although many workflow management systems have problems dealing with the deferred choice, emerging standards in the web services composition domain have no problems supporting the patterns. For example, BPEL offers a construct called *pick* which directly captures this pattern.

- Assume that the workflow language being used supports cancelation of activities (Pattern 19) through either a special transition (e.g. Staffware) or through an API (most other engines). Cancelation of an activity means that the activity is being removed from the designated worklist as long as it has not been started yet. The deferred choice can be realized by enabling all alternatives via an AND-split. Once the processing of one of the alternatives is started, all other alternatives are canceled. Consider the deferred choice between B and C in Figure 9.9(a). This could be implemented using cancelation of activities in the following way. After A, both B and C are enabled. Once B is selected/executed, activity C is canceled. Once C is selected/executed, activity B is canceled. Note that the solution does not always work because B and C can be selected/executed concurrently.

- Another solution to the problem is to replace the deferred choice by an explicit XOR-split, i.e. an additional activity is added. All triggers activating the alternative branches are redirected to the added activity. Assuming that the activity can distinguish between triggers, it can activate the proper branch. Note that the solution moves part of the routing to the application or task level. Moreover, this solution assumes that the choice is made based on the type of trigger.

9.3.6 Cancelation patterns

When discussing possible solutions for the deferred choice pattern, we already mentioned the pattern *Cancel activity* (Pattern 19). This is one of two cancelation patterns. The other pattern is *Cancel case* (Pattern 20). The cancel activity pattern disables an enabled activity, i.e. a thread waiting for the execution of an activity is removed. The cancel case pattern completely removes a case, i.e. workflow instance, (i.e. even if parts of the process are instantiated multiple times, all descendants are removed). Both constructs are supported by YAWL through a more generic construct which removes all tokens from a given region. A more detailed discussion of the cancelation patterns is outside the scope of this chapter.

9.4 CONCLUSION

This chapter introduced several control-flow patterns which can be used to support modeling efforts, to train workflow designers, and to assist in the selection of workflow management systems. Although inspired by workflow management systems

the patterns are not limited to workflow technology but applicable to Process Aware Information Systems (PAIS) ranging from EAI platforms and web services composition languages to case-handling systems and groupware. For example, the patterns have not only been used to evaluate several commercial and academic workflow management systems [5, 4] but also several standards including UML [8], BML [24], and BPEL4WS [23]. For more information on these evaluations and interactive animations for each of the patterns we refer to www.workflowpatterns.com. Throughout this chapter we used YAWL diagrams to illustrate the patterns. YAWL [4] demonstrates that it is possible to support the patterns in a direct and intuitive manner. YAWL is an open-source initiative and supporting tools can be downloaded from www.yawl-system.com. Current research aims at further developing YAWL and to develop patterns and pattern languages for other perspectives than control-flow (notably the resource perspective; a collection of data patterns was recently reported in [20]).

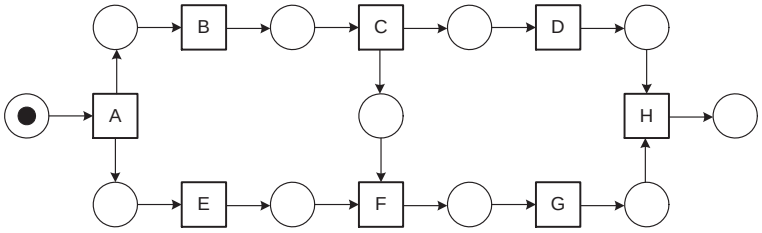

Fig. 9.10 How to model this in terms of Pi calculus?

Note that the 20 patterns mentioned in this chapter are not complete. Therefore, we invite users, researchers and practitioners to contribute. Moreover, some systems and languages have limitations not adequately addressed by the patterns. For example, in Staffware it is not allowed to connect a condition (i.e. an exclusive choice/XOR-split) to a wait step (i.e. synchronization/AND-join). Moreover, a condition in Staffware is always binary, i.e. to model a choice involving three alternatives two condition elements are needed. For most of these limitations there are simple workarounds. However, this is not always the case as is illustrated by the following example. Consider the Petri net shown in Figure 9.10. This model shows a simple classical Petri net with 8 transitions. First A is executed followed by B and E in parallel. B is followed by C, however, F has to wait for the completion of both E and C, etc. Finally, H is executed and all transitions have been executed exactly once. Although the Petri net is very simple (e.g. it does not model any choices, only parallelism), process algebras like Pi calculus [16] have problems modeling this example. To understand the problem consider the Petri net shown in Figure 9.10 without the connection between C and F. In that case the sequences $B.C.D$ and $E.F.G$ are executed in parallel in-between A and H. In terms of Pi calculus (or any other process algebra) this is denoted as $A.(B.C.D|E.F.G).H$. In this notation the "." is used to denote sequence and the "|" denotes parallelism. Indeed this notation is elegant and allows for computer manipulation. Unfortunately, such a simple notation

is not possible if the connection between C and F is restored. The linear language does not allow for this while for a graph based language like Petri nets this is not a problem. Note that the claim is *not* that Pi calculus cannot model the process shown in Figure 9.10. However, it illustrates that even powerful languages like Pi calculus have problems supporting certain patterns. This is particularly relevant in the domain of web services where Pi calculus is being put forward as a starting point for developing (future versions of) languages for describing service-based processes.

Acknowledgements

We would like to thank Bartek Kiepuszewski, Alistair Barros, and Petia Wohed for their contribution to the research involving workflow patterns, and Lachlan Aldred for his contribution to the YAWL initiative.

9.5 EXERCISES

Exercise 9.1 (Identification of patterns in an informal description)
Consider the following informal description of a process for insurance claim handling.

> When a claim is received, it is first registered. After registration, the claim is classified leading to two possible outcomes: simple or complex. If the claim is simple, the insurance is checked. For complex claims, both the insurance and the damage are checked independently. After the check(s), an assessment is performed which may lead to two possible outcomes: positive or negative. If the assessment is positive, the garage is phoned to authorize the repairs and the payment is scheduled (in this order). In any case (whether the outcome is positive or negative), a letter is sent to the customer and the process is considered to be complete. At any moment after the registration and before the end of the process, the customer may call to modify the details of the claim. If a modification occurs before the payment is scheduled, then the claim is classified again, and the process is repeated from that point on. If a modification occurs after the payment is scheduled and before the letter is sent, a "stop payment" task is performed and the process is repeated starting with the classification of the claim.

Which tasks can be identified in this scenario, and which workflow patterns link these tasks?

Exercise 9.2 (Identification of patterns in an existing model)
Consider the YAWL specification in Figure 9.11. Which patterns occur in this specification and where? For example, the "AND-split" pattern can be found between tasks "register", "send_form", and "evaluate".

Exercise 9.3 (Identification of patterns in an existing model)
Consider the UML activity diagram shown in Figure 5.1 (Chapter 5). Which patterns occur in this model and where? Same question for the ARIS function flow in Figure 6.2 (Chapter 6).

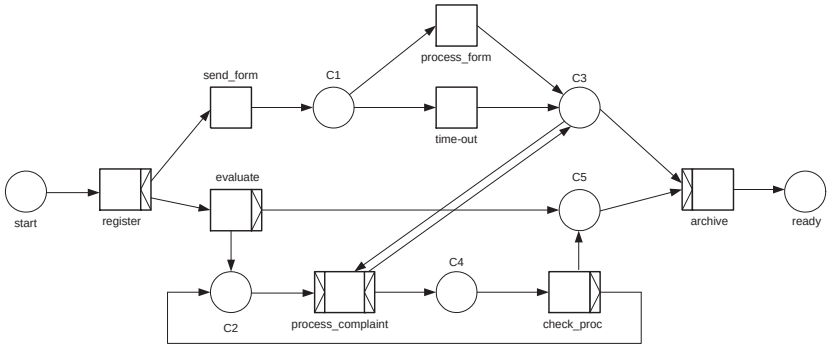

Fig. 9.11 YAWL specification for exercise 9.2.

Exercise 9.4 (Pattern implementation)
Figures 9.5 and 9.6 feature two YAWL specifications illustrating the multi-choice (OR-split) and the synchonizing merge (OR-join) patterns. Translate these YAWL specifications into classical Petri nets (see Chapter 7). In other words, expand the YAWL OR-split and OR-join constructs in terms of places and transitions (possibly labeled with empty tasks).

Exercise 9.5 (Pattern implementation)
Figure 9.12 contains a YAWL specification in which the edges are labeled with boolean expressions $C1$, $C2$, $C3$ and their negations.

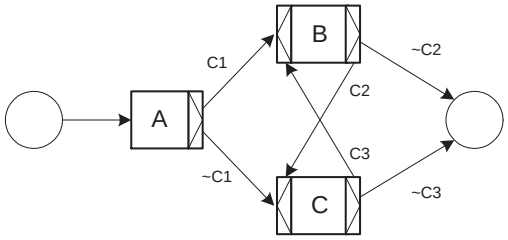

Fig. 9.12 YAWL specification for Exercise 9.5.

This specification contains an arbitrary loop in which tasks B and C can be repeated multiple times in alternation until the process completes. A possible execution of this process specification is AB, i.e. task A is executed, then condition $C1$ evaluates to true so B is executed after which condition $C2$ evaluates to false so the process terminates. Other possible executions include AC, ABC, ACB, $ABCB$, $ACBC$, etc.

Some process modeling or process execution languages only provide constructs for structured loops (e.g. constructs of the form `while (condition) { fragment of process to be repeated }`) like in contemporary imperative programming

languages such as C and Java.[8]. How could the specification in Figure 9.12 be expressed in a language that provides "while" loops, conditional statements of the form `if (condition) { fragment of process }`, and simple sequencing between tasks (which can be denoted using a semicolon ';'), but does not support arbitrary loops. Consider each of the following two cases:

1. The language in question supports "break" statements allowing to exit a "while" loop in the middle of its body like in contemporary imperative programming languages such as C and Java.

2. The language in question does not support "break" statements. Hint: You may introduce one or several auxiliary boolean variable(s). When a condition is evaluated, it can be assigned to a boolean variable and this variable can be used in the condition parts of "if" and "while" statements.

Exercise 9.6 (Evaluation of PAIS development platforms)
Select a tool for PAIS development (see for example the tools mentioned in Chapters 1–4 and 17). Evaluate the selected tool in terms of the patterns. The evaluation should state, for each of the patterns presented in this chapter, whether the tool provides "direct support" for that pattern or not. If the answer to this question is positive for a given pattern, briefly explain how the pattern is supported. Otherwise, provide (if possible) a workaround solution to capture the pattern in question. Some sample evaluations of tools can be found in [5].

REFERENCES

1. W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and Implementation of the YAWL System. In A. Persson and J. Stirna, editors, *Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159, Riga, Latvia, June 2004. Springer.

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

3. W.M.P. van der Aalst, J. Desel, and E. Kindler. On the Semantics of EPCs: A Vicious Circle. In M. Nüttgens and F.J. Rump, editors, *Proceedings of the EPK 2002: Business Process Management using EPCs*, pages 71–80, Trier, Germany, November 2002. Gesellschaft für Informatik, Bonn, Germany.

[8]This is the case for example of BPEL as discussed in Chapter 14

4. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. Accepted for publication in *Information Systems*, and also available as QUT Technical report FIT-TR-2003-04, Queensland University of Technology, Brisbane, Australia, 2003.

5. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

6. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual Modelling of Workflows. In M.P. Papazoglou, editor, *Proceedings of the 14th International Object-Oriented and Entity-Relationship Modelling Conference (OOER)*, volume 1021 of *Lecture Notes in Computer Science*, pages 341–354, Gold Coast, Australia, December 1998. Springer.

7. J. Dehnert and P. Rittgen. Relaxed Soundness of Business Processes. In K.R. Dittrich, A. Geppert, and M.C. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *Lecture Notes in Computer Science*, pages 157–170, Interlaken, Switzerland, June 2001. Springer.

8. M. Dumas and A.H.M. ter Hofstede. UML activity diagrams as a workflow specification language. In M. Gogolla and C. Kobryn, editors, *Proceedings of the 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML)*, volume 2185 of *Lecture Notes in Computer Science*, pages 76–90, Toronto, Canada, October 2001. Springer.

9. L. Fischer, editor. *Workflow Handbook 2003, Workflow Management Coalition*. Future Strategies, Lighthouse Point, FL, USA, 2003.

10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison Wesley, Reading, MA, USA, 1995.

11. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.

12. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2003. Available via http://www.workflowpatterns.com.

13. B. Kiepuszewski, A.H.M. ter Hofstede, and W.M.P. van der Aalst. Fundamentals of Control Flow in Workflows. *Acta Informatica*, 39(3):143–209, 2003.

14. B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On structured workflow modelling. In B. Wangler and L. Bergman, editors, *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 1789 of *Lecture Notes in Computer Science*, pages 431–445, Stockholm, Sweden, June 2000. Springer.

15. P. Langner, C. Schneider, and J. Wehler. Petri Net Based Certification of Event driven Process Chains. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets 1998*, volume 1420 of *Lecture Notes in Computer Science*, pages 286–305, Lisbon, Portugal, June 1998. Springer.

16. R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, UK, 1999.

17. E.A.H. Platier. *A logistical view on business processes: BPR and WFM concepts (in Dutch)*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1996.

18. D. Riehle and H. Züllighoven. Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.

19. P. Rittgen. Modified EPCs and their Formal Semantics. Technical report 99/19, University of Koblenz-Landau, Koblenz, Germany, 1999.

20. N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow Data Patterns. QUT Technical report FIT-TR-2004-01, Queensland University of Technology, Brisbane, Australia, 2004.

21. Eastman Software. *RouteBuilder Tool User's Guide*. Eastman Software, Inc, Billerica, MA, USA, 1998.

22. WfMC. Workflow Management Coalition Terminology & Glossary, Document Number WFMC-TC-1011, Document Status - Issue 3.0. Technical report, Workflow Management Coalition, Brussels, Belgium, February 1999.

23. P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In I.Y. Song, S.W. Liddle, T.W. Ling, and P. Scheuermann, editors, *Proceedings of the 22nd International Conference on Conceptual Modeling (ER)*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215, Chicago, IL, USA, October 2003. Springer.

24. P. Wohed, E. Perjons, M. Dumas, and A. ter Hofstede. Pattern-Based Analysis of EAI Languages - The Case of the Business Modeling Language. In O. Camp, J. Filipe, S. Hammoudi, and M. Piatinni, editors, *Proceedings of the 5th International Conference on Enterprise Information Systems (ICEIS)*, pages 174–182, Angers, France, April 2003. Escola Superior de Tecnologia do Instituto Politécnico de Setúbal, Setúbal, Portugal.