Workflow Control Patterns in @enterprise

Authors: Herbert Groiss, Michael Dobrovnik

Date: May 2023

Document version: 1.1

Groiss Informatics GmbH
Strutzmannstr. 10
A-9020 Klagenfurt, AUSTRIA
http://www.groiss.com

Abstract

In this paper we show how the twenty workflow control patterns described in the paper from van der Aalst et.al. [1] can be implemented in **@enterprise**. Additionally, a complete example shows the usage of *all* patterns in one process.
@enterprise is a commercial workflow management system on the market since the year 2000. It is used by many customers for production and administrative workflows, especially where high demands on throughput or process complexity exist.
This paper has at least two goals: to ease the comparison of the modeling capabilities between **@enterprise** and other workflow systems and to help the workflow designer choosing the right control structure for real-world modeling challenges.
The original version of the paper has been written 2005, in 2023 we updated the graphical notation, now using BPMN.

Introduction

@enterprise is a workflow management system developed and distributed by Groiss Informatics GmbH (www.groiss.com). Many customers from different industries have used the system for several years. It is used in mission critical areas and where strict requirements must be met:

- security: applications in banking and military organizations
- performance: several thousands of users, gigabytes of data daily
- process complexity: processes with more than a hundred single steps and integration to more than a dozen other applications

Process definition in **@enterprise** can be done graphically or using a script language. For this textual notation the language WDL (for Workflow Definition Language) has been developed, syntax and semantics inspired by procedural programming languages like Java.
The graphical process definition is using BPMN2.0 (see [3]), however, the allowed structures are restricted, allowing a 1:1 transformation to the textual notation. The process editor, normally used for defining processes, only allows to construct syntactically correct graphs, i.e. when an *if* is inserted by the user, the if node, the end node, and the two edges between them are drawn automatically. Such structured processes are easier to read and understand, especially, if they are complex.
Fig.1 shows the WDL-script and the graphical definition of a simple workflow. In the WDL script you may recognize the syntactical elements of the control structures.
The specification of an activity consists of an agent description followed by an activity id. The agent can either be the id of a user, the id of a role, the reference to the agent of a previous activity, or the name of a form field. In the latter case the form field must contain the id of a user or a role (at run-time). After the activity id we can optionally specify which forms are in the scope of the activity.

```
process ifdemo()
  version 1;
  name "ifdemo";
  forms f Jobform;
  application default;
  begin
    if (f.recipient = null) then
      r0 left(f);
    else
      r1 right(f);
    end;
    while (f.'subject' = 1) do
      r2 while1(f);
      r3 while2(f);
    end;
end
```

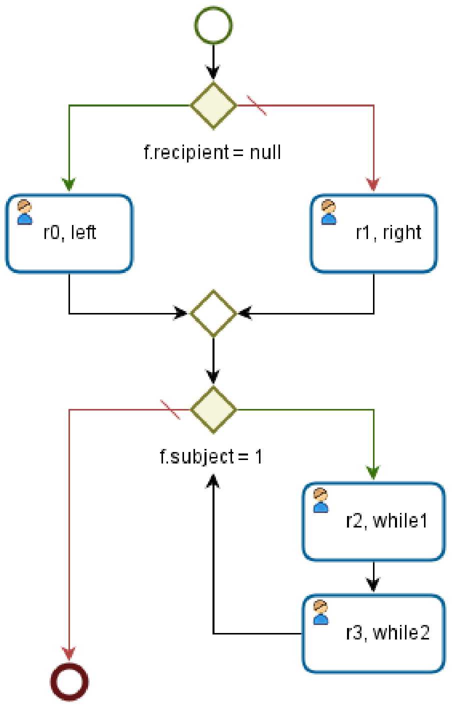


Figure 1: **WDL script and process graph**

In the BPMN graph we use colored edges with the following semantics: The black edges are unconditional ones and are followed when the previous node is finished. The green and red edges originate from binary decision nodes (if, while, ...) and are followed when the condition attached to such a node is true (green) or false (red).
We will not describe the **@enterprise** modeling language here further in detail and refer the reader to the product documentation which can be found on our website [2].
In the next sections we show how the patterns can be realized in **@enterprise** . The names of the patterns, the description and the examples are taken from the paper from van der Aalst el. al. [1].
After this, a comprehensive example shows the usage of all these patterns. The interested reader can download an evaluation version of **@enterprise** from our website:
https://www.groiss.com/

Pattern 1: Sequence

Description

An activity in a workflow process is enabled after the completion of another (preceding) activity in the same process. Synonyms: sequential routing, serial routing.

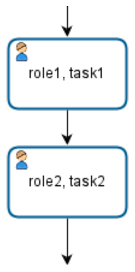
Examples

- Activity *send_bill* is executed after the execution of activity *send_goods*.
- An insurance claim is evaluated after the client's file is retrieved.

Implementation

@enterprise supports the sequence pattern, WDL example and graphical representation:

```
role1 task1();
role2 task2();
```



Pattern 2: Parallel Split

Description

A point in the workflow process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order.
Synonyms: AND-split, parallel routing, fork.

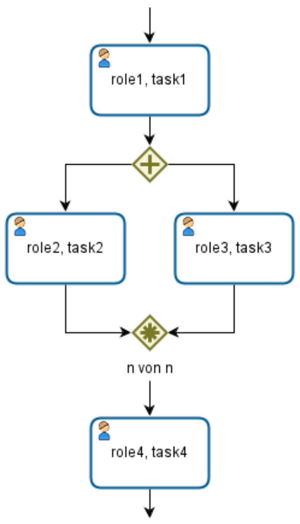
Examples

- The execution of the activity payment enables the execution of the activities ship goods and inform customer.
- After registering an insurance claim two parallel subprocesses are triggered: one for checking the policy of the customer and one for assessing the actual damage.

Implementation

@enterprise supports the parallel split with the `andpar` and `orpar` control structures. WDL example and graphical representation:

```
role1 task1();
andpar
    role2 task2();
 |  role3 task3();
end;
role4 task4();
```



Pattern 3: Synchronization

Description

A point in the workflow process where multiple parallel subprocesses/activities converge into one single thread of control, thus synchronizing multiple threads. It is an assumption of this pattern that each incoming branch of a synchronizer is executed only once (if this is not the case, then see Patterns 13-15 (Multiple Instances Requiring Synchronization)).
Synonyms: AND-join, rendezvous, synchronizer.

Examples

- Activity archive is enabled after the completion of both activity *send_tickets* and activity *receive_payment*.
- Insurance claims are evaluated after the policy has been checked and the actual damage has been assessed.

Implementation

@enterprise supports this pattern through its `andpar` control structure, see task4 in the example illustrating the previous pattern.

Pattern 4: Exclusive Choice

Description

A point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen.
Synonyms: XOR-split, conditional routing, switch, decision.

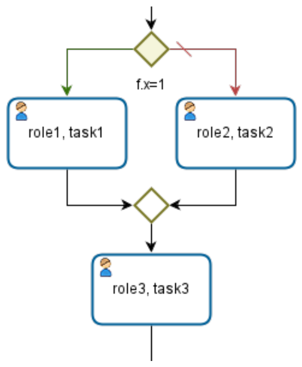
Examples

- Activity *evaluate_claim* is followed by either *pay_damage* or *contact_customer*.
- Based on the workload, a processed tax declaration is either checked using a simple administrative procedure or is thoroughly evaluated by a senior employee.

Implementation

@enterprise supports this pattern through its `if` control structure. WDL example and graphical representation:

```
if (f.x = 1) then
  role1 task1();
else
  role2 task2();
end;
role3 task3();
```



WDL condition have the syntax known from procedural programming languages. They can contain Java method calls and references to form fields (denoted by *formname* ”.” *fieldname*).

Pattern 5: Simple Merge

Description

A point in the workflow process where two or more alternative branches come together without synchronization. It is an assumption of this pattern that none of the alternative branches is ever executed in parallel (if this is not the case, then see Pattern 8 (Multi-merge) or Pattern 9 (Discriminator)). Synonyms: XOR-join, asynchronous join, merge.

Examples

- Activity *archive_claim* is enabled after either *pay_damage* or *contact_customer* is executed.
- After the payment is received or the credit is granted the car is delivered to the customer.

Implementation

@enterprise supports this pattern through its `if` control structure. After execution of one of the if branches the process execution continues with the node after the end node of the if structure (see task3 in the previous pattern).

Pattern 6: Multi-choice

Description

A point in the workflow process where, based on a decision or workflow control data, a number of branches are chosen.
Synonyms: Conditional routing, selection, OR-split.

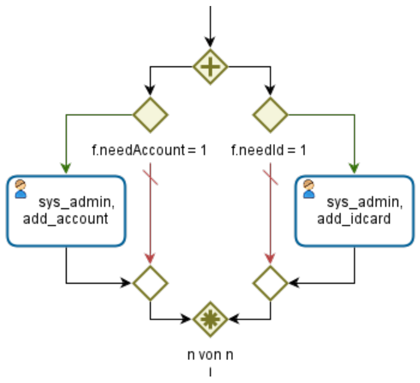
Examples

– After executing the activity *evaluate_damage* the activity *contact_fire_department* or the activity *contact_insurance_ company* is executed. At least one of these activities is executed. However, it is also possible that both need to be executed.

Implementation

The multi-choice is supported in **@enterprise** through a combination of `andpar` and `if`. WDL example and graphical representation:

```
andpar
  if (f.needAccount = 1) then
    sys_admin add_account();
  end
|
  if (f.needId = 1) then
    sys_admin add_idcard();
  end;
end;
```



Pattern 7: Synchronizing Merge

Description

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

Examples

– Extending the example of Pattern 6 (Multi-choice), after either or both of the activities *contact_fire_department* and *contact_insurance_company* have been completed (depending on whether they were executed at all), the activity *submit_report* needs to be performed (exactly once).

Implementation

Implicitly supported in **@enterprise** through use of `andpar` for the multi-choice, see previous pattern.

Pattern 8: Multi-merge

Description

A point in a workflow process where two or more branches reconverge without synchronization. If more than one branch gets activated, possibly concurrently, the activity following the merge is started for every activation of every incoming branch.

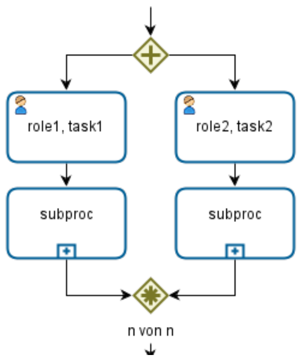
Examples

– Sometimes two or more parallel branches share the same ending. Instead of replicating this (potentially complicated) process for every branch, a multi-merge can be used. A simple example of this would be two activities *audit_application* and *process_application* running in parallel which should both be followed by an activity *close_case*.

Implementation

The multi-choice can be modeled with an `andpar` (or `orpar`) where the activity following the merge is put into each branch. If this activity is complex, a subprocess can be created. WDL example and graphical representation:

```
andpar
  role1 task1();
  call subproc();
|
  role2 task2();
  call subproc();
end;
```



There is an alternative implementation involving the use of `goto`:

```
andpar
  role1 task1();
  goto join_label;
|
  role2 task2();
 <join_label>
 role3 close_case();
end;
```

However, we don't recommend to use gotos, as they circumvent the structuring of processes.

Pattern 9: Discriminator

Description

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

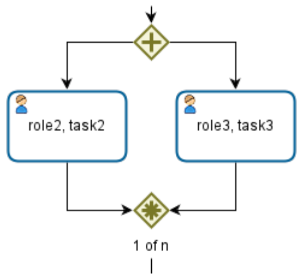
Examples

– To improve query response time, a complex search is sent to two different databases over the Internet. The first one that comes up with the result should proceed the flow. The second result is ignored.

Implementation

@enterprise supports the discriminator pattern with the `orpar` control structure. WDL example and graphical representation:

```
orpar
    role2 task2();
 |  role3 task3();
end;
```



Pattern 10: Arbitrary Cycles

Description

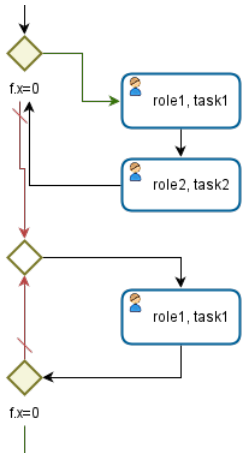
A point in a workflow process where one or more activities can be done repeatedly.
Synonyms: Loop, iteration, cycle.

Implementation

@enterprise supports two types of loops, `while` loops with condition checking at the begin and `repeat` loops with condition checking at the end of the loop. Arbitrary cycles can always be converted to these control structures.
WDL example and graphical representation:

```
while (f.x = 0) do
  role1 task1();
  role2 task2();
end;

repeat
 role1 task1();
until (f.x = 0);
```



Pattern 11: Implicit Termination

Description

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

Implementation

In **@enterprise** every process is implicitly terminated by reaching its unique end node. Since all nodes which allow for alternative execution paths or parallel threads require a corresponding join node there is always a unique end node.

Pattern 12: Multiple Instances Without Synchronization

Description

Within the context of a single case (i.e., workflow instance) multiple instances of an activity can be created, i.e., there is a facility to spawn off new threads of control. Each of these threads of control is independent of other threads. Moreover, there is no need to synchronize these threads.
Synonyms: Multi threading without synchronization, Spawn off facility

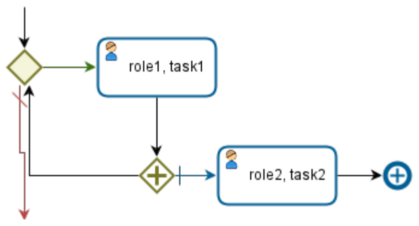
Examples

– A customer ordering a book from an electronic bookstore such as Amazon may order multiple books at the same time. Many of the activities (e.g., billing, updating customer records, etc.) occur at the level of the order. However, within the order multiple instances need to be created to handle the activities related to one individual book (e.g., update stock levels, shipment, etc.). If the activities at the book level do not need to be synchronized, this pattern can be used.

Implementation

The `branch` control structure of **@enterprise** implements this pattern.
WDL example and graphical representation:

```
while (f.x = 1) do
 role1 task1()
 branch
   role2 task2();
 end;
end;
```



Pattern 13: Multiple Instances With a Priori Design Time Knowledge

Description

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

Examples

- The requisition of hazardous material requires three different authorizations.

Implementation

The `andpar` control structure directly implements this pattern, see pattern 2. If the activities in the branches are the same, the use of subprocesses is recommended.

Pattern 14: Multiple Instances With a Priori Runtime Knowledge

Description

For one case an activity is enabled multiple times. The number of instances of a given activity for a given case varies and may depend on characteristics of the case or availability of resources [CCPP98, JB96], but is known at some 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.

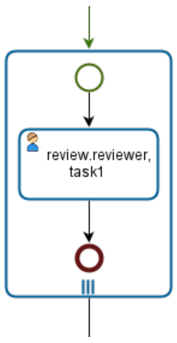
Examples

- In the review process of a scientific paper submitted to a journal, the activity *review_paper* is instantiated several times depending on the content of the paper, the availability of referees, and the credentials of the authors. Only if all reviews have been returned, processing is continued.
- For the processing of an order for multiple books, the activity *check_availability* is executed for each individual book. The shipping process starts if the availability of each book has been checked.
- When booking a trip, the activity *book_flight* is executed multiple times if the trip involves multiple flights. Once all bookings are made, the invoice is to be sent to the client.
- When authorizing a requisition with multiple items, each item has to be authorized individually by different workflow users. Processing continues if all items have been handled.

Implementation

The `parfor` control structure implements this pattern.
WDL example and graphical representation:

```
parallel for review in mainform.1 do
   review.reviewer task1(review);
end;
role2 task2();
```



An execution thread is started for each occurrence of the subform with id ”1” in form mainform. As an alternative to the usage of subforms an iterator can be defined. This is a Java class implementing the interface `com.groiss.wf.ParForIterator`, which allows to define the number of parallel threads generated programmatically based on arbitrarily complex situations. Process execution proceeds after all parallel threads have been finished.

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. The difference with Pattern 14 is that even while some of the instances are being executed or already completed, new ones can be created.

Examples

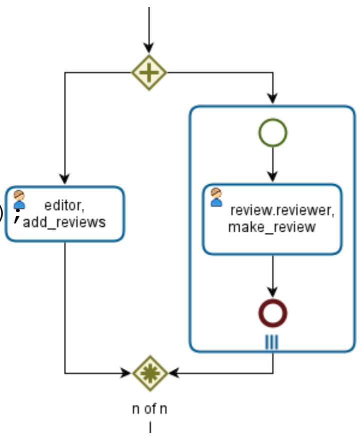
– The requisition of 100 computers involves an unknown number of deliveries. The number of computers per delivery is unknown and therefore the total number of deliveries is not known in advance. After each delivery, it can be determined whether a next delivery is to come by comparing the total number 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.

Implementation

As in the previous pattern we use `parfor` to implement this pattern. It is possible to start additional threads from a task parallel to the `parfor` structure. In this task the agent can add additional subforms. With the function *add parfor steps* available in the @enterprise user interface and API the corresponding *new* tasks can be generated.
WDL example and graphical representation:

```
andpar
  editor add_reviews(f);
|
  parallel for review in mainform.1 do
    review.reviewer make_review(review);
  end;
end;
```



Pattern 16: Deferred Choice

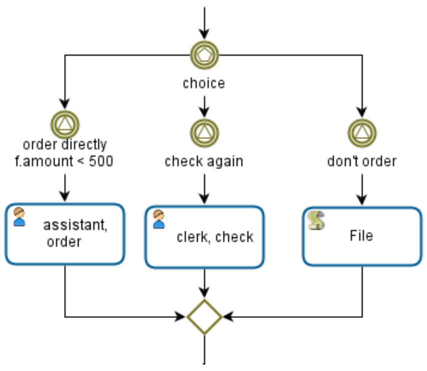
Description

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 AND-split, only one of the alternatives is executed. This means that once the environment activates one of the branches the other alternative branches are withdrawn. It is important to note that the choice is delayed until the processing in one of the alternative branches is actually started, i.e. the moment of choice is as late as possible.
Synonyms: External choice, implicit choice, deferred XOR-split.

Implementation

The `choice` control structure implements this pattern. WDL example and graphical representation:

```
choice
  "order directly", f.amount < 500:
     assistant order(f);
  "check again":
     clerk check(f);
  "don’t order":
     system Archive.file(f) "File";
end;
```



Each path has a name, where an arbitrary string can be given, and an optional condition. The engine first checks the conditions of all branches, only the branches where no condition is specified or where the condition evaluates to true are presented for selection by the user. After the user decides on such a branch the other branches are canceled.

Pattern 17: Interleaved Parallel Routing

Description

A set of activities is executed in an arbitrary order: Each activity in the set is executed, the order is decided at run-time, and no two activities are executed at the same moment (i.e. no two activities are active for the same workflow instance at the same time).
Synonyms: Unordered sequence.

Examples

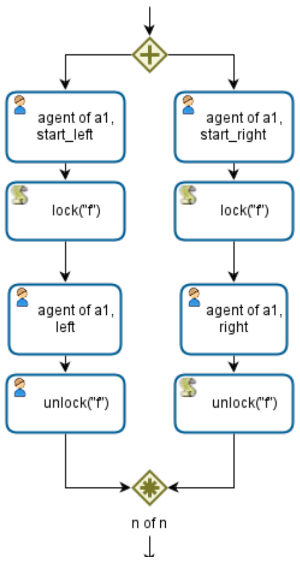
– The Navy requires every job applicant to take two tests: *physical_test* and *mental_test*. These tests can be conducted in any order but not at the same time.

– At the end of each year, a bank executes two activities for each account: *add_interest* and *charge_credit_ card_ costs*. These activities can be executed in any order. However, since they both update the account, they cannot be executed at the same time.

Implementation

In **@enterprise** `andpar` together with a synchronization mechanism can be used for this pattern. For synchronizing we have several options each having its drawbacks. The reason is that in **@enterprise** there is no distinction between the "created" state of an activity and the "processing" state. When the task goes to a role the synchronization can take place when the agent "takes" the activity from the role worklist into his personal worklist. When the task is sent directly to a user, an activity "start_X" must precede the actual activity. The following WDL example and graphical representation shows this possibility:

```
andpar
  a1:user start_left(f);
  system com.groiss.wf.SystemAction.lock("f");
  a1:user left(f);
  system com.groiss.wf.SystemAction.unlock("f");
 |
  a1:user start_right(f);
  system com.groiss.wf.SystemAction.lock("f");
  a1:user right(f);
  system com.groiss.wf.SystemAction.unlock("f");
end;
```



The keyword `system` designates an activity carried out by the workflow engine itself without human intervention. The method `lock` will fail, if there is already a lock on form f. Activities `left` and `right` are never active at the same time.

Pattern 18: Milestone

Description

The enabling of an activity depends on the case being in a specified state, i.e. the activity is only enabled if a certain milestone has been reached which did not expire yet. Consider three activities named A, B, and C. Activity A is only enabled if activity B has been executed and C has not been executed yet, i.e. A is not enabled before the execution of B and A is not enabled after the execution of C. The state in between B and C is modeled by place m. This place is a milestone for A. Note that A does not remove the token from M: It only tests the presence of a token.
Synonyms: Test arc, deadline (cf. [JB96]), state condition, withdraw message.

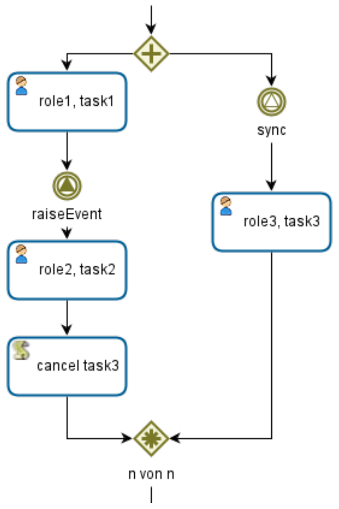
Examples

- In a travel agency, flights, rental cars, and hotels may be booked as long as the invoice is not printed.
- A customer can withdraw purchase orders until two days before the planned delivery.
- A customer can claim air miles until six months after the flight.

Implementation

In **@enterprise** this pattern can be implemented using the event mechanism. A `sync` node waits until a matching event is raised; the event name and an event handler must be specified. The `raiseEvent` node raises a named event; event name and transaction mode (`current_tx` or `new_tx`) are required parameters. WDL example and graphical representation:

```
andpar
  role1 task1();
  raiseEvent(e1, current_tx);
  role2 task2();
  system com.groiss.wf.SystemAction.
    cancelActivity("task3");
|
  sync(e1,com.groiss.event.EventHandler);
  role3 task3();
end;
```



When the task `task1` is completed an event is raised which causes the start of `task2`. When the task `left2` is being finished a ”`cancel`” event is sent to `task3` causing the abortion of `task3` (and the start of the successor node of task3). So, `task3` can be executed after `task1` has been completed but before `task2` has been completed.

Pattern 19: Cancel Activity

Description

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

Examples

– Normally, a design is checked by two groups of engineers. However, to meet deadlines it is possible that one of these checks is withdrawn to be able to meet a deadline.

– If a customer cancels a request for information, the corresponding activity is disabled.

Implementation

@enterprise allows authorized users to cancel activities via the function cancel_task. This allows the agent of the activity to ”cancel” the activity instead of finishing it normally. The function can be attached to specific tasks using the administration interface, it can also be executed via the API.

Pattern 20: Cancel Case

Description

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

Examples

- In the process for hiring new employees, an applicant withdraws his/her application.
- A customer withdraws an insurance claim before the final decision is made.

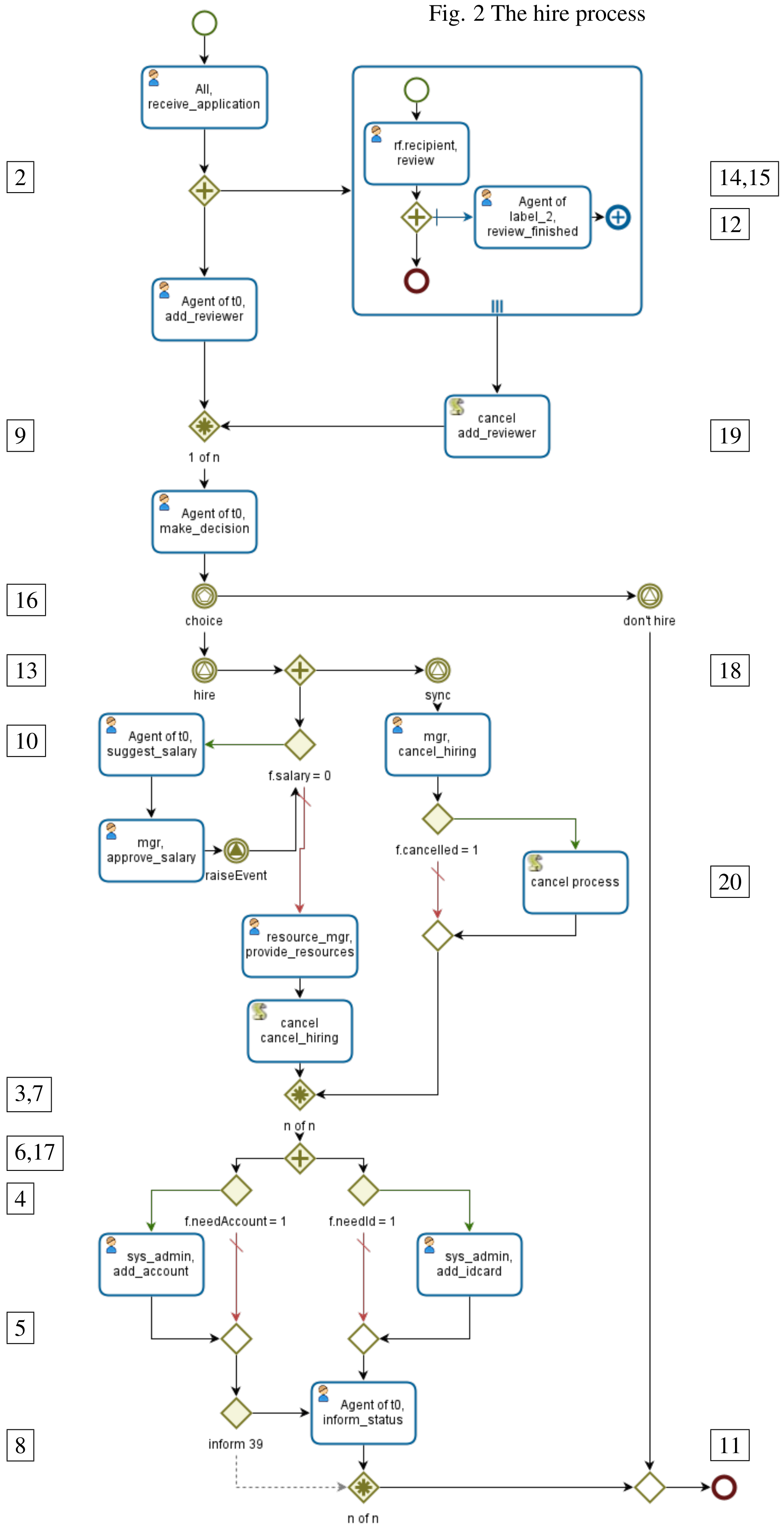
Implementation

@enterprise allows authorized users to cancel a process instance. The function is also available via the API.

A complete Example

In this section we present a complete example containing all workflow control patterns. See Fig. 2 for the process definition.
The example shows the hiring of an employee. The person who receives the application starts the process and chooses a set of reviewers. The process goes to the reviewers where each reviewer has to fill in the review form. Meanwhile the process starter has the possibility to add additional reviewers. When all reviewers are finished a system step cancels the *add_reviewer* task.
In the next step the process owner can view the reviews and makes the decision after finishing the task: the following choice has the two options ”don’t hire” (the process ends) and ”hire”. If the decision is for hiring, the negotiation of the salary follows. In a loop the process owner suggest the salary, the manager either approves it and the process continues or it goes back into the loop. After the first decision of the manager we sent an event to the second parallel path of the process which is waiting for the event to continue. After receiving the event the manager has the opportunity to cancel the process: He must set the corresponding field in the form and finish the task. In this case the process will branch to the *cancel process* system step to abort the process.
After the salary is fixed the resource_mgr will organize the resources needed for the new employee. After this step, the parallel path is canceled (i.e. canceling the process is no longer possible).
In the next parallelism we have the two tasks add_account and add_idcard. They are only executed if some condition is met (depending on checkboxes in the hire form). After each of these tasks the task *inform_status* is executed and the process is finished.

Fig. 2 The hire process



Usage of the patterns

The following table shows where the patterns are used in the hire process. In Fig. 2 the appearances of the patterns are marked with boxes containing the pattern number.

1. Sequence	several...
2. Parallel split	after the first task the process splits: in one branch the reviewers perform the review task, in the other branch the agent of the first task can still add additional reviewers.
3. Synchronization	the andjoin nodes, for example to join the branches where the salary has been fixed (left) and where the manger can cancel the process.
4. Exclusive choice	the ifs: the task *add_account* is performed only when the checkbox *needAccount* is on.
5. Simple merge	end if nodes
6. Multi choice	andpar containing ifs: both of the actions *add_account* and *add_idcard* can be executed, on one of them, or none.
7. Synchronizing merge	several uses of andjoin (end of the andpar)
8. Multi merge	goto to the other parallel branch: the task *inform_status* is executed after *add_account* and after *add_idcard*. However, we don't recommend to use gotos to jump between parallel branches. We only show the possibility here.
9. Discriminator	orjoin (end of orpar)
10. Arbitrary cycles	the while loop
11. Implicit termination	The process terminates when the unique end node is reached.
12. Multiple instance without synchronization	The branch control structure starts the activity *review_finished* whenever a *review* is finished.
13. Multiple instance with a priori design time knowledge	several uses of andpar
14. Multiple instance with a priori runtime knowledge	The *parallel for* control structure starts the *review* task for each review form created for the process in the first task.
15. Multiple instance without a priori runtime knowledge	In the task *add_reviewer* the agent of this task can add additional review forms. Using the task function *add parfor steps* additional *review* tasks for these forms are started.
16. Deferred choice	The *choice* control structure after the task *make_decision*.
17. Interleaved parallel routing	The pattern is not directly visible in the process graph, because the *lock* and *unlock* actions are defined as takeHook and postcondition of the tasks *add_account* and *add_idcard*, respectively. The takeHook is executed when an agent takes the activity from the role-worklist to his personal worklist. the postcondition is executed when the agent finishes the task, the lock will be released.
18. Milestone	The activity *cancel_hiring* is not started before the task *approve_salary* is finished. At this point an event is generated which finishes the sync step in the other par branch, which starts the *cancel_hiring* activity.
19. Cancel activity	The activity *add_reviewer* is canceled when all reviewers have finished their reviews.
20. Cancel case	The process is canceled through a system step if the *canceled* checkbox is on.

WDL script of the hiring process:

```
process hiring()
version 1;
name "hiring";
forms f hireform;
timeoutaction none;
application Hire;
begin
   <t0>
   all receive_application(f);
   orpar
      t0:user add_reviewer(f);
      |
      parallel for rf in f.1 do
         rf.recipient review(rf);
         branch
            t0:user review_finished(rf);
         end;
      end;
      system com.groiss.wf.SystemAction.cancelActivity("add_reviewer");
   end;
   t0:user make_decision(f);
   choice
   "don't hire":
   "hire":
      andpar
         sync(e1, com.groiss.event.EventHandler, f);
         mgr cancel_hiring(f);
         if (f.cancelled = 1) then
            system com.groiss.wf.SystemAction.cancelProcess();
         end;
         |
         while (f.salary = 0) do
            t0:user suggest_salary(f);
            mgr approve_salary(f);
            raiseEvent(e1, current_tx, f);
         end;
         resource_mgr provide_resources(f);
         system com.groiss.wf.SystemAction.cancelActivity("cancel_hiring");
      end;
      andpar
         if (f.needAccount = 1) then
            sys_admin add_account(f);
         end;
         goto inform;
         |
         if (f.needId = 1) then
            sys_admin add_idcard(f);
         end;
         <inform>
         t0:user inform_status(f);
      end;
   end;
end
```

Conclusions

In this paper we have shown how the workflow control patterns described by van der Aalst et.al. can be implemented in **@enterprise**. Although our modeling language differs quite substantially from the one used by van der Aalst, most patterns could be implemented in a very simple, intuitive and straightforward way.
Moreover, almost all of the control structures provided by the WDL of **@enterprise** are used in the control patterns. However, one pattern that we did miss in the treatise of van der Aalst et. al. was recursion, which we came across in several projects in practice, especially in workflows supporting authorization processes in highly hierarchic organizations.
This high coverage gives evidence that these patterns cover most of the control flow requirements that manifest in practical workflows. We make this observation on the strong base of 25 years of practical experience in workflow modeling as well as construction of **@enterprise** and refinement of its capabilities to adequately support real world processes.

References

[1] van der Aalst W.M.P., A.H.M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, Workflow Patterns, Distributed and Parallel Databases, 14(3), pages 5-51, July 2003.

[2] Groiss Informatics: @enterprise documentation, https://www.groiss.com/en/customer-portal/documentation/

[3] Business Process Model and Notation,
https://en.wikipedia.org/wiki/Business_Process_Model_and_Notation