Business Object Process Modeling overview of Workflow patterns

Version of the document: 0.9

Author: Ľudovít Scholtz <ludovit@scholtz.sk>

Date: 17.4.2013-18.4.2013

Table of contents

1 About this paper

This paper links Business Object Process Model with the Control-Flow Workflow Patterns. It is guideline for every pattern on how it can, may and should be implemented.

1.1 Business Object Process Model

Business Object-Process Model (BOPM) is notation, set of rules, set of best practices, and schema for process modeling in object environment. BOPM in difference to UML, BMPN and other notations views the *Process* as set of state of *Process objects*.

BOPM defines standard objects as are defined in the UML, and expects the compatibility with all existing object to database tools. BOPM adds an extension to the existing objects model and defines the *Process object* as the *Object* where it is possible to track changes in the states of the *Object*.

BOPM should be comprehensive by upper management, lower management, all persons who should perform any type of role in the process, programmers and analysts, and should create a "runnable" environment that does not need any other programmer influence. BOPM should be easy to understand and modifiable by the management, programmers and analysts to express and easy implement any new processes and keep them up to date.

BOPM will be managed by nonprofit organization which is currently being created. Until that time, BOPM is managed by Ludovit Scholtz.

1.2 Workflow patterns

The Workflow Patterns initiative is a joint effort of Eindhoven University of Technology (led by Professor Wil van der Aalst) and Queensland University of Technology (led by Professor Arthur ter Hofstede) which started in 1999. The aim of this initiative is to provide a conceptual basis for process technology. In particular, the research provides a thorough examination of the various perspectives (control flow, data, resource, and exception handling) that need to be supported by a workflow language or a business process modelling language. The results can be used for examining the suitability of a particular process language or workflow system for a particular project, assessing relative strengths and weaknesses of various approaches to process specification, implementing certain business requirements in a particular process-aware information system, and as a basis for language and tool development[1].

[1] http://www.workflowpatterns.com/

1.3 RFC-2119

The upper case keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119. [2]

1.4 Definitions

BOPM	Business Object Process Model is notation, set of rules, set of best practices, and schema for process modeling in object environment.
Object	Object as defined in UML
Process object	Object where it is possible to track changes in the states of Object.
Process state	State of the Process object
Process action	Timeout action or Role action
Timeout action	Action executed in time out after no other action was performed
Role action	Action performed by the person, script, or event on behalf of the Process role.
Process role	Role who is allowed to execute the Process action and/or can receive Process messages.
Process message	Message to the user of the process role who should be informed about the development of the Process object
Process	Change of states of the Process object
Process overview	Graphical representation of one Process for selected Process role or Timeout. Graphical representation should be in tables with the process states on one side, and process actions on the other side.

[2] http://www.ietf.org/rfc/rfc2119.txt

2 Control-Flow Patterns

2.1 Sequence

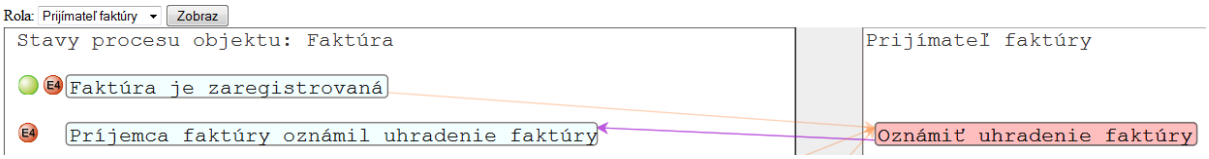
A task in a process in enabled after the completion of a preceding task in the same process.

Sequence is represented by arrows and link between Process state, Process action, and Process state.

Note that *Process action* can be either *Timeout* or *Role action*.

Example:

Invoice has not been paid yet -> Client announced that the invoice has been paid -> Invoice has been paid but payment did not arrived yet

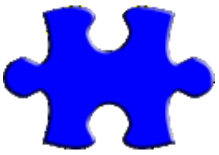


2.2 Parallel Split

The divergence of a branch into two or more parallel branches each of which execute concurrently.

Split is performed in the Process function by lunching new Process. Note that the Process function can be executed before setting the new state, on new state, on active function catch event, before end of the state, and after the state has changed. Process function can be also executed on performing the action.

Every Process state with function defined must have flag Function defined.



Picture 1: Flag Function

Process state MUST have the flag NewProcess defined. In the process overview should be displayed which new process is lunched. In the object overview, should be link between running Process object and new Process object.

Graphical representation of flag NewProcess displayed here, and should be displayed next to the process state in the Process overview:

Picture 2: Flag NewProcess

Example: After registration before the full membership the invoice is issued.

Client has registered > Invoice issued >  Waiting for invoice to be paid by client

2.3 Pattern 3 (Synchronization)

> The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled.

State of the Process object where the Catch event from other Process object is required before the change of the state.

Process state MUST define Synchronisation flag.

Picture 3: Flag Synchronization

Example:

After issuing an invoice, process can continue only when it is paid.

State: Waiting for the invoice to be paid by client -> Action (Manager): Invoice has been paid -> State: User has full membership

2.4 Pattern 4 (Exclusive Choice)

> The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on a mechanism that can select one of the outgoing branches.

Exclusive Choice is natural behavior of BOPM. Every state of Process object can be every time changed only with one Process action, which is exclusive.

Example:

State: Invoice has been issued > Action: Client announced the payment > State: Client has paid, but we have not received money yet

State: Invoice has been issued > Action: We have received payment > State: Client has paid

2.5 Pattern 5 (Simple Merge)

> The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

Simple merge is ensured by the use of one Process action in several Process states or by use of the following from several Process actions to one Process state.

Example:

State: Invoice has been issued > Action: We have received payment > State: Client has paid

State: Client has paid, but we have not received money yet > Action: We have received payment > State: Client has paid

2.6 Pattern 6 (Multi-Choice)

> The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches.

Multi-Choice is performed in the Process function by lunching new Processes.

Process state MUST have the flag NewProcesses defined. In the process overview should be displayed which new processes are lunched. In the object overview, should be link between running Process object and all new Process objects.

Graphical representation of flag NewProcesses displayed here, and should be displayed next to the process state in the Process overview.

Picture 4: Flag NewProcesses

Example: Depending on the nature of the emergency call, one or more of the despatch-police, despatch-fire-engine and despatch-ambulance tasks is immediately initiated.

Process object: Emergency

State: Emergency call received -> Action: Dispatch units -> State:  Required units dispatched

2.7 Pattern 7 (Structured Synchronizing Merge)

> The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. The Structured Synchronizing Merge occurs in a structured context, i.e. there must be a single Multi-Choice construct earlier in the process model with which the Structured Synchronizing Merge is associated and it must merge all of the branches emanating from the Multi-Choice. These branches must either flow from the Structured Synchronizing Merge without any splits or joins or they must be structured in form (i.e. balanced splits and joins).

The Structured Synchronized merge is ensured by the IF conditions of the Process actions, which executes a function that manages all requirements of the synchronized merge.

The Structured Synchronizing Merge action MUST contain flag ConditionalProcessAction. The state of the process where Structured Synchronizing Merge can originate MUST contain the flag Synchronization.

Picture 5: Flag ConditionalProcessAction

Example: Depending on the type of emergency, either or both of the despatch-police and despatch-ambulance tasks are initiated simultaneously. When all emergency vehicles arrive at the accident, the transfer-patient task commences.

Process Object: Rescue

State: All relevant vehicles has been dispatched > Action: Transfer patient -> State: Patient is being transferred

2.8 Pattern 8 (Multi-Merge)

> The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

Multi-merge is ensured by advanced synchronization managed by function.

Synchronization flag must be set for the Process state.

Picture 6: Flag Synchronize

Example: The lay_foundations, order_materials and book_labourer tasks occur in parallel as separate process branches. After each of them completes the quality_review task is run before that branch of the process finishes.

Process object: Building

State: Foundations has been finished -> Action: Perform quality review -> State: Performing quality review

Action: Perform quality review requires the Process object Foundation to be in state Finished, Process object Materials in state ordered or delivered, the Process object Labor in state booked.

2.9 Pattern 9 (Structured Discriminator)

> The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The Structured Discriminator construct resets when all incoming branches have

> been enabled. The Structured Discriminator occurs in a structured context, i.e. there must be a single Parallel Split construct earlier in the process model with which the Structured Discriminator is associated and it must merge all of the branches emanating from the Structured Discriminator. These branches must either flow from the Parallel Split to the Structured Discriminator without any splits or joins or they must be structured in form (i.e. balanced splits and joins).

The function that manages the merge of one or many processes into one SHOULD throw an event for the processes which should no longer be taken into consideration, so that they may be canceled.

The action can be performed either from the function check of the state of the other processes or object variables, or the action can catch the events from previously lunched processes.

Example: When handling a cardiac arrest, the check_breathing and check_pulse tasks run in parallel. Once the first of these has completed, the triage task is commenced. Completion of the other task is ignored and does not result in a second instance of the triage task.

Process object: Cardiac arrest

State:  Checking cardiac arrest > Action:  Breathing is not OK > State: Performing triage task

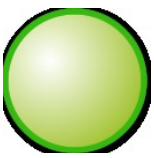
State:  Checking cardiac arrest > Action:  Pulse is not OK > State: Performing triage task

The action Breathing is not OK catches the event of previously lunched process Check breathing or the function checks the state of the previously lunched process. The action Pulse is not OK catches the event of previously lunched process Check pulse or the function checks the state of the previously lunched process. The first of the action that is caught changes the state of the process to new State. The function should throw new event to terminate the other process that is no longer needed.

2.10 Pattern 10 (Arbitrary Cycles)

> The ability to represent cycles in a process model that have more than one entry or exit point. It must be possible for individual entry and exit points to be associated with distinct branches.

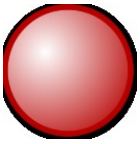
Processes can have more than one entry and exit points.



Picture 7: EntryPoint

Every state of entry point should be market with flag EntryPoint

Every state of end point should be marked with flag EndPoint.



Picture 8:EndPoint

Every process state SHOULD have defined Timeout action. Process states that does

not have Timeout action defined SHOULD be EndPoints.

One Process state can be EntryPoint and also EndPoint.

Example:

State: New invoice with no parameters > Action: Set Invoice number > State: Invoice with invoice number

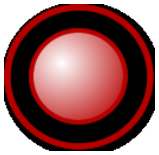
State: State: Invoice with invoice number > Action: Send to client > Invoice has been sent

In this example, system can create blank invoices that requires from accounting department the invoice number, and accounting department can also create invoice with the invoice number already in place. Therefore the Process have 2 distinct entry points.

2.11 Pattern 11 (Implicit Termination)

> A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future and the process instance is not in deadlock. There is an objective means of determining that the process instance has successully completed.

End point that does not have any Process action defined SHOULD be marked as TerminateEndpoint.



Picture 9: Terminate Endpoint

Example:

Process object: Invoice

State: Client announced the payment for the invoice -> Action: We have received the payment -> State: Invoice has been paid

In this example, the invoice is marked as paid, and no action is required.

2.12 Multiple Instances without Synchronization

> Within a given process instance, multiple instances of a task can be created. These instances are independent of each other and run concurrently. There is no requirement to synchronize them upon completion. Each of the instances of the multiple instance task that are created must execute within the context of the process instance from which they were started (i.e. they must share the same case identifier and have access to the same data elements) and each of them must execute independently from and without reference to the task that started them.

There are two possible variants to the way in which this pattern can operate. First creates instance task runs within a loop and the new task instances are created sequentially. Second variant is where task instances are all created simultaneously.

In BOPM both variants are possible. First can be achieved by the loop between process states, where confirmation is done by function that all required instances have been launched. The other variant is achievable by the function that creates multiple instances of the required Process objects in one Process state.

If the process state can create only one new instance of the another Process object, the flag NewProcess MUST be set.

If the process state can create one or more new instances of the process objects MUST be marked with the NewProcesses flag.

Example:

A list of traffic infringements is received by the Transport Department. For each infringement on the list an Issue-Infringment-Notice task is created. These tasks run to completion in parallel and do not trigger any subsequent tasks. They do not need to be synchronized at completion.

Variant 1:

Process object: Verify traffic infringements

State: Need to resolve traffic infringements -> Action: Lunch the process of traffic infringements resolvment -> State: Need to resolve traffic infringements

State: Need to resolve traffic infringements -> Action: All traffic infringements are being resolved -> Wait for next scheduled check

In this example there is a loop on the same process state with the Action able to decide which traffic infringements should be resolved first.

Variant 2:

State: Need to resolve traffic infringements -> Action: All traffic infringements are being resolved -> Wait for next scheduled check

In this example the function lunches new instances of Process object Traffic infringements resolution simultaneously and does not need to synchronize them.

2.13 Pattern 13 (Multiple Instances with a priori Design-Time Knowledge)

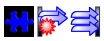
> Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the task instances at completion before any subsequent tasks can be triggered.

Process state of the running process lunches the new processes and synchronizes before proceeding.

Example:

The Annual Report must be signed by all six of the Directors before it can be issued.

Process object: Annual report

State:  Annual report is created and waiting for signatures -> Action:  All directors have signed the Annual report -> State: Annual report is waiting for time to get published

2.14 Pattern 14 (Multiple Instances with a priori Run-Time Knowledge)

> Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but is known before the task instances must be created. Once initiated, these instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered.

Process state of the running process lunches the new processes and synchronizes before proceeding.

Function manages which new processes are being lunched. Data and Resource availability is ensured by the Process Object that is being lunched.

Example: When diagnosing an engine fault, multiple instances of the check-sensor task can run concurrently depending on the number of error messages received. Only when all messages have been processed, can the identify-fault task be initiated;

Process object: Engine diagnostics

State:  Diagnostics is being executed -> Action:  One set of errors is being reported -> State: Fix motor

State: Diagnostics is being executed -> Action: Second set of errors is being reported -> State: Fix electricity

Example: In the review process for a paper submitted to a journal, the review paper task is executed several times depending on the content of the paper, the availability of referees and the credentials of the authors. The review process can only continue when all reviews have been returned;

Process object: Paper

State: Reviewing -> Action: More review is needed -> State: Reviewing

State: Reviewing -> Action: All reviews were passed OK -> State: Passed

State: Reviewing -> Action: Serious problems found in the article -> State: Return for rewriting

2.15 Pattern 15 (Multiple Instances without a priori Run-Time Knowledge)

> Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. Once initiated, these instances are independent of each other and run concurrently. At any time, whilst instances are running, it is possible for additional instances to be initiated. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered.

Process state of the running process lunches the new processes and synchronizes before proceeding.

Function manages which new processes are being lunched. Data and Resource availability is ensured by the Process Object that is being lunched.

Example: The dispatch of an oil rig from factory to site involves numerous transport shipment tasks. These occur concurrently and although sufficient tasks are started to cover initial estimates of the required transport volumes, it is always possible for additional tasks to be initiated if there is a shortfall in transportation requirements. Once the whole oil rig has been transported, and all transport shipment tasks are completed, the next task (assemble rig) can commence.

Process object: Delivery

State: Waiting for the delivery to be finished -> Action: More Deliveries are needed -> State: Waiting for the delivery to be finished

State:  Waiting for the delivery to be finished -> Action: All partial deliveries have finished -> State: Assembling rig

2.16 Pattern 16 (Deferred Choice)

> A point in a process where one of several branches is chosen based on interaction with the operating environment. Prior to the decision, all branches represent possible future courses of execution. The decision is made by initiating the first task in one of the branches i.e. there is no explicit choice but rather a race between different branches. After the decision is made, execution alternatives in branches other than the one selected are withdrawn.

This is natural behavior of BOPM whereas from the running Process object in Process state can only be chosen first Process action depending on the environment.

Example: At the commencement of the Resolve complaint process, there is a choice between the Initial customer contact task and the Escalate to manager task. The Initial customer contact is initiated when it is started by a customer services team member. The Escalate to manager task commences 48 hours after the process instance commences. Once one of these tasks is initiated, the other is withdrawn.

Process object: Complaint

State: Customer has raised the complaint -> Action: Resolve by Service team -> State: Complaint is sent to customer for reviewing

State: Customer has raised the complaint -> Timeout (T+48H) -> State: Complaint is sent to upper manager

State: Complaint is sent to upper manager -> Action: Resolve by Upper manager -> State: Complaint is sent to customer for reviewing

In this example, by simple change of state of the process, the action available for service team has been withdrawn by the timeout and option to resolve the complaint was given to the upper manager.

2.17 Pattern 17 (Interleaved Parallel Routing)

> A set of tasks has a partial ordering defining the requirements with respect to the order in which they must be executed. Each task in the set must be executed once and they can be completed in any order that accords with the partial order. However, as an additional requirement, no two tasks can be executed at the same time (i.e. no two tasks can be active for the same process instance at the same time).

The parallel routing is ensured by lunching several instances of different Process objects. The Process object that supervises all instances of sub processes MUST define the structure of process flow.

Process action that allows use of parameters MUST contain flag UsesParameters.

Picture 10: Flag UsesParameters

Example: When dispatching an order, the pick goods, pack goods and prepare invoice tasks must be completed. The pick goods task must be done before the pack goods task. The prepare invoice task can occur at any time. Only one of these tasks can be done at any time for a given order.[3]

Variant 1:

Process object: Order

State: dispatching an order -> Action: Pick goods -> State: Goods are picked

State: dispatching an order -> Action: prepare invoice -> State: Invoice prepared, ready to pick goods

State: Invoice prepared, ready to pick goods -> Action: Pick goods -> State: Invoice prepared and goods are picked

State: Goods are picked -> Action: pack goods -> State: Goods are packed

State: Goods are picked -> Action: prepare invoice -> State: Invoice prepared and goods are picked

State: Invoice prepared and goods are picked -> Action: pack goods -> State Order has been dispatched

State: Goods are picked -> Action: prepare invoice -> State: Order has been dispatched

By parameter with Prepare invoice action is meant the file of the issued invoice.

Example: When dispatching an order, the pick goods, pack goods and prepare invoice tasks must be completed. The pick goods task must be done before the pack goods task. The prepare invoice task can occur at any time. Only one of these tasks can be done at any time for a given order.

Variant 2:

Process object: Goods

State: Need to pick up goods -> Action: Pick up goods -> State: Goods have been picked up

[3] http://www.workflowpatterns.com/patterns/control/state/wcp17.php

State: Goods have been picked up -> Action: Pack goods -> State: Goods have been packed up

Process object: Invoice

State: New invoice -> Action: Set up invoice -> State: Invoice has been prepared

Process object: Order

State: Dispatch order -> Action: Setup goods -> State: Goods are being prepared, invoice not issued

State: Dispatch order -> Action: Prepare invoice -> State: Invoice is being prepared, goods not yet

State: Goods are being prepared, invoice not issued -> Action: Prepare invoice -> State: Waiting for invoice to be prepared and goods to be prepared

State: Invoice is being prepared, goods not yet -> Action: Setup goods -> State: Waiting for invoice to be prepared and goods to be prepared

State: Waiting for invoice to be prepared and goods to be prepared -> Action: Goods and invoice is prepared -> State: Invoice and goods sent

Please note, that 3 process objects are created. The Order process object manages the process of the process issuance and the process of goods preparation. Process of issuance of invoice and process of good preparation are running in parallel while the process of good pickup and packing is in sequence order.

2.18 Pattern 18 (Milestone)

> A task is only enabled when the process instance (of which it is part) is in a specific state (typically a parallel branch). The state is assumed to be a specific execution point (also known as a milestone) in the process model. When this execution point is reached the nominated task can be enabled. If the process instance has progressed beyond this state, then the task cannot be enabled now or at any future time (i.e. the deadline has expired). Note that the execution does not influence the state itself, i.e. unlike normal control-flow dependencies it is a test rather than a trigger.

This is natural behavior of BOPM. Every Process state is according to definition in Patter 18 Milestone.

The Milestone in BOPM is however defined as Process state with no Timeout defined while it has at least one Process action defined. This state should be marked with EndPoint flag as well as EntryPoint flag. It is used typically for allowing lunching additional processes on one Process Object.

Example: Client can have either FCM or DEM membership. When he has the FCM membership, he must go over compliance procedure to get the DEM membership. Client does not need to change his membership from FCM to DEM, so in the state "Client has FCM membership" is no timeout.

Process object: Client

State: Client has FCM membership -> Action (Client): Request start of becoming DEM member -> State: Client has started the DEM compliance procedure

2.19 Pattern 19 (Cancel Task)

> An enabled task is withdrawn prior to it commencing execution. If the task has started, it is disabled and, where possible, the currently running instance is halted and removed.

The possibility to cancel task MUST be expressed with the Error Process action for every Process state in the cancelable Process where possible.

Process action for which cancels the task must be marked with the flag ErrorAction.

Picture 11: Error flag

Example: The purchaser can cancel their building inspection task at any time before it commences.

State: Building inspection is in phase 3 -> Action (Purchaser): Cancel inspection -> State: Inspection has been canceled

2.20 Pattern 20 (Cancel Case)

> A complete process instance is removed. This includes currently executing tasks, those which may execute at some future time and all sub-processes. The process instance is recorded as having completed unsuccessfully.

The possibility to cancel instance of Process object MUST be expressed with the Error Process action for every Process state of each Process object's process where possible.

The Cancel case Process action must be marked with the Error flag.

The end point of the resolution of the cancel case error action should be termination of the Process object and in the case must be marked with Termination flag.

Example: During an insurance claim process, it is discovered that the policy has expired and, as a consequence, all tasks associated with the particular process instance are cancelled[4].

Process object: Insurance claim

State: Waiting for all documents to be acquired -> Action: Policy has expired -> State: Insurance claim not granted

2.21 Pattern 21 (Structured Loop)

> The ability to execute a task or sub-process repeatedly. The loop has either a pre-test or post-test condition associated with it that is either evaluated at the beginning or end of the loop to determine whether it should continue. The looping structure has a single entry and exit point.

There are three ways how Structured loops can be executed in the BOPM.

First variant relies solely on the function executed before or when the process state is reached. This function takes all required variables and performs any operations with it. Note that this should be used if there is no necessity to have distinct states defined and no special action defined.

Second variant relies on conditional function of the Process action. This variant should be chosen if there is no necessity to have distinct states defined, and no it is preferred to show the conditional Process action that changes the Process object to new state.

Third variant of distinct process states should be chosen if it is required to show distinct process states which are achieved during the structured loop.

Example: Repeat the select player task until the entire team has been selected.

Process object: Team

Variant 1:

State: Selecting team members -> Action (Couch): Team members selected -> Waiting for match to begin

Variant 2:

State: Selecting team members -> Action (Couch): Select team member -> Selecting team members

State: Selecting team members -> Action (Couch): All team members selected -> Waiting for match to begin

[4] http://www.workflowpatterns.com/patterns/control/cancellation/wcp20.php

Variant 3:

State: Selecting team members -> Action (Couch): Select front player -> State: Front player selected

State: Front player selected -> Action (Couch): Select back player -> State: Players selected, waiting for match to begin

2.22 Pattern 22 (Recursion)

> The ability of a task to invoke itself during its execution or an ancestor in terms of the overall decomposition structure with which it is associated.

Recursion is natural behavior of BOPM. Recursion can occur in several cases.

1. When Process state's Process action points to the current Process state.
2. When function lunches new instance of current Process object.
3. When function execute invoke action.

Example: An instance of the resolve-defect task is initiated for each mechanical problem that is identified in the production plant. During the execution of the resolve-defect task, if a mechanical fault is identified during investigations that is not related to the current defect, another instance of the resolve-defect is started. This subprocess can also initiate further resolve-defect tasks should they be necessary. The parent resolve-defect task cannot complete until all child resolve-defect tasks that it initiated have been satisfactorily completed[5].

Process object: Resolve defects

State: Under investigation -> Action: Investigation complete -> State: perform fix

Process object: Car fix

State: Identify issues -> Action: All issues are resolved -> State: Car is prepared for releasing

2.23 Pattern 23 (Transient Trigger)

> The ability for a task instance to be triggered by a signal from another part of the process or from the external environment. These triggers are transient in nature and are lost if not acted on immediately by the receiving task. A trigger can only be utilized if there is a task instance waiting for it at the time it is received.

[5] http://www.workflowpatterns.com/patterns/control/new/wcp22.php

Transient triggers are supported through Events. The external environment is monitored by process which throws the Event. All Process actions which catches the exact event is executed, and states of those Process objects is changed following the Process action.

The events are meant to bring automation for the processes where computing systems can perform tasks faster than humans.

The Process states that throw events must contain flag ThrowEvent. The Process Actions that catches the events MUST contain the flag CatchEvent.

Picture 13: Flag ThrowEvent

Picture 12: Flag CatchEvent

Example: When payment comes physically on the account, responsible person must mark the invoice as paid. The event thrown by the account monitoring system on new incoming money transfer pushes the event over the system, and Process action for the invoice can be executed and invoice can be mark as paid. Afterwards, the function in the state invoice paid throws the event, and the process that issued the invoice can commit to new state after the invoice has been paid.

Process object: Account monitoring

State: Perform check -> Timeout (30s) -> State: Perform check

Process object: Invoice

State: Invoice has been issued -> Action (Manager): Invoice can be marked as paid -> Action: Invoice has been paid

2.24 Pattern 24 (Persistent Trigger)

> The ability for a task to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the process until they can be acted on by the receiving task.

Persistent triggers are supported by instantionalization of the Process object or should be implemented into the Process objects of the existing object. Every meaningful persistent trigger MUST have its own Process object defined.

Example: Start a new instance of the Inspect Vehicle task for each service overdue signal that is received[6].

Process object: Vehicle

State: In good condition -> Action: Service time -> State: Vehicle has to go into service

Process object: Driver

State: Need to perform driving -> Action: Vehicle has to go into service -> State: Plan the service for vehicle

State: Plan the service for vehicle -> Action: Service is planned -> State: Need to perform driving

2.25 Pattern 25 (Cancel Region)

> The ability to disable a set of tasks in a process instance. If any of the tasks are already executing (or are currently enabled), then they are withdrawn. The tasks need not be a connected subset of the overall process model.

Cancel region is supported by the function call which can intelligently cancel set of tasks. The function sends the Cancel ThrowEvent, and all processes which can be canceled must perform the cancel task. Please note that all cancelations of the processes will perform in harmonized way. Also note, that it is not possible to cancel process which is in the state that does not accept the cancellation.

Example: Withdraw all tasks in the Waybill Booking process after the freight-lodgment task[7].

Process object: Freight

State: Freight-lodgment has finished.

Process object: Booking

State: Waiting for booking confirmation -> Action: Cancel booking -> Booking has been canceled

2.26 Pattern 26 (Cancel Multiple Instance Task)

> Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. At any time, the multiple instance task can be cancelled and any instances which have not completed are withdrawn. Task instances that have already completed are unaffected.

[6] http://www.workflowpatterns.com/patterns/control/new/wcp24.php
[7] http://www.workflowpatterns.com/patterns/control/new/wcp25.php

Canceling of instances of running processes can be performed in the function of the controlling process. The function sends the Events, and Process state MUST have flag ThrowEvent .

Example: Run 500 instances of the Protein Test task with distinct samples. If it has not completed one hour after commencement, cancel it[8].

Process object: Protein test

State: Performing tests -> Timeout (1H) -> State: Canceling tests

State: Canceling tests -> Action (medic): Tests have been canceled.

2.27 Pattern 27 (Complete Multiple Instance Task)

> Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered. During the course of execution, it is possible that the task needs to be forcibly completed such that any remaining instances are withdrawn and the thread of control is passed to subsequent tasks.

Multiple instances of the tasks are created as new instances of Process objects. Each created process object reference is stored in the controller's process object. When timeout comes, the controllers process object sends the cancel event to all instances. Finished instances are not canceled because the cancel event does not apply to them. After the call, some instances are in the state finished successfully and some instances are in the state canceled. The controller now can go over all instances and initialize next task.

Example: Run 500 instances of the Protein Test task with distinct samples. One hour after commencement, withdraw all remaining instances and initiate the next task[9].

Process object: Protein test

State: Performing tests -> Timeout (1H) -> State: Canceling unfinished tests

State: Canceling unfinished tests -> Action (medic): Next task.

In this example, the medic can check all tests in next task within the function call.

[8] http://www.workflowpatterns.com/patterns/control/new/wcp26.php
[9] http://www.workflowpatterns.com/patterns/control/new/wcp27.php

2.28 Pattern 28 (Blocking Discriminator)

> The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. The Blocking Discriminator construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the Blocking Discriminator has reset.

Blocking discriminator is supported in BOPM by the execution of Controlling Process Object.

Function can check the state of the initialized process, and can obey executing it again if it is in the process.

Example: The check credentials task can commence once the confirm delegation arrival or the security check task has been completed. Although these two tasks can execute concurrently, in practice, the confirm delegation arrival task always completes before security check task. Another instance of the check credentials task cannot be initiated if a preceding instance of the task has not yet completed. Similarly, subsequent instances of the confirm delegation arrival and the security check tasks cannot be initiated if a preceding instance of the check credentials task has not yet completed.

Process object: Check credentials

State: Check required -> Action: Confirm delegation arrival -> State: Waiting for security check to be finished

State: Waiting for security check to be finished -> Action: Security check finished -> State: Check ok

State: Check required -> Action: The security check is ok -> State: Check ok

Process object: Credentials controller

State: Have person to be checked -> Action: Check ok -> State: Have person to be checked

State: Have person to be checked -> Action: No more person to be checked -> State: Checking has finished

2.29 Pattern 29 (Cancelling Discriminator)

> The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. Triggering the Cancelling Discriminator also cancels the execution of all of the other incoming branches and resets the construct.

The cancelling discriminator is supported by the controller Process object. The controller MUST send the cancel events for the residual processes that does not need any more.

Example: After the extract-sample task has completed, parts of the sample are sent to three distinct laboratories for examination. Once the first of these laboratories completes the sample-analysis, the other two task instances are cancelled and the review-drilling task commences.

Process object: Sample controller

State: Sending extracted samples to the labs -> Action: Receive lab result -> State: Cancel other lab tests

State: Cancel other lab tests -> Action: Canceled -> State: Review drilling task

2.30 Pattern 30 (Structured Partial Join)

> The convergence of two or more branches (say m) into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when n of the incoming branches have been enabled where n is less than m. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The join construct resets when all active incoming branches have been enabled. The join occurs in a structured context, i.e. there must be a single Parallel Split construct earlier in the process model with which the join is associated and it must merge all of the branches emanating from the Parallel Split. These branches must either flow from the Parallel Split to the join without any splits or joins or be structured in form (i.e. balanced splits and joins).

Supported in BOPM by managing the control flow in control process object by lunching several instance types of other process objects, and conditional check in the process action managed by function.

Example: Once two of the preceding three Expenditure Approval tasks have completed, trigger the Issue Cheque task. Wait until the remaining task has completed before allowing the Issue Cheque task to fire again[10].

Process object: Expenditure

State:  Check cheques -> Action:  Two checks have been checked -> State: Settle the checked cheques

State:  Settle the checked cheques -> Action: Cheques has been settled -> State: Check cheques

[10] http://www.workflowpatterns.com/patterns/control/new/wcp30.php

2.31 Pattern 31 (Blocking Partial Join)

> The convergence of two or more branches (say m) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when n of the incoming branches has been enabled (where 2 = n < m). The join construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the join has reset.

Blocking partial joins are supported by synchronization and conditional process action.

Example: When the first member of the visiting delegation arrives, the check credentials task can commence. It concludes when either the ambassador or the president arrived. Owing to staff constraints, only one instance of the check credentials task can be undertaken at any time. Should members of another delegation arrive, the checking of their credentials is delayed until the first check credentials task has completed.

Process object: Verifier

State:  Waiting for visitor -> Action: First visitor arrived -> State:  Checking delegation

State:  Checking delegation -> Action:  Other visitor arrived -> State:  Checking delegation

State:  Checking delegation -> Action: Delegation checked -> State:  Waiting for visitor

The process action adds the other visitor in the row, and waiting for visitor state checks if any visitor is in the row.

2.32 Pattern 32 (Cancelling Partial Join)

> The convergence of two or more branches (say m) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when n of the incoming branches have been enabled where n is less than m. Triggering the join also cancels the execution of all of the other incoming branches and resets the construct.

Cancelling partial join is supported in BOPM similar as Blocking partial joins with the requirement that to the incoming branches are sent the cancel events.

Example: Once the picture is received, it is sent to three art dealers for the examination. Once two of the prepare condition report tasks have been completed, the remaining prepare condition report task is cancelled and the plan restoration task commences[11].

Process object: Review picture

State: Sending to the art dealers for examination -> Action: Two reports have been completed -> State: Planning restoration

2.33 Pattern 33 (Generalized AND-Join)

> The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. Additional triggers received on one or more branches between firings of the join persist and are retained for future firings. Over time, each of the incoming branches should deliver the same number of triggers to the AND-join construct (although obviously, the timing of these triggers may vary).

Supported by either defining distinct states, or by conditional process action function.

Example: When all Get Directors Signature tasks have completed, run the Complete Contract task[12].

Process object: Report

Variant 1:

State: To be signed -> Action (Subdirector): Sign -> State: Signed by subdirector

State: Signed by subdirector -> Action (Director): Sign -> State: Waiting for time to be published

Variant 2:

State: To be signed -> Action: Signed by all parties -> State: Waiting for time to be published

2.34 Pattern 34 (Static Partial Join for Multiple Instances)

> Within a given process instance, multiple concurrent instances of a task (say m) can be created. The required number of instances is known when the first task instance commences. Once n of the task instances have completed (where n is less than m), the next task in the process is triggered. Subsequent completions of the remaining m-n instances are inconsequential, however all instances must have completed in order for the join construct to reset and be subsequently re-enabled.

[11] http://www.workflowpatterns.com/patterns/control/new/wcp32.php
[12] http://www.workflowpatterns.com/patterns/control/new/wcp33.php

Static partial join for multiple instances are supported by conditional process action and catching the events of the subtasks when they commences.

Example: Examine 10 samples from the production line for defects. Continue with the next task when 7 of these examinations have been completed.

Process object: Production line

State: Performing check, samples are being examined -> Action: 7 samples are ok -> State: Shipping items

State: Performing check, samples are being examined -> Action: 4 samples are not ok -> State: Initiating process of fault detection

2.35 Pattern 35 (Cancelling Partial Join for Multiple Instances)

> Within a given process instance, multiple concurrent instances of a task (say m) can be created. The required number of instances is known when the first task instance commences. Once n of the task instances have completed (where n is less than m), the next task in the process is triggered and the remaining m-n instances are cancelled.

Cancelling Partial Join for Multiple Instances is supported in BOPM. Multiple concurrent instances are achieved through instancioning of the process objects, and number of new instances is defined by the variable of the object and managed by function. After synchronization and conditional process action managed by function and achieving new state, the cancel event MUST be sent to instances not yet finished.

Example: Run 500 instances of the Protein Test task with distinct samples. Once 400 of these have completed, cancel the remaining instances and initiate the next task[13].

Process object: Protein test

State: Initializing tests -> Action: Initialized -> State: Waiting for results of tests

State: Waiting for results of tests -> Action: Required amount of results finished -> State: Cancelling remaining tasks

State: Cancelling remaining tasks -> Action: Finished cancelling -> State: Initializing next task

2.36 Pattern 36 (Dynamic Partial Join for Multiple Instances)

> Within a given process instance, multiple concurrent instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state

[13] http://www.workflowpatterns.com/patterns/control/new/wcp35.php

> data, resource availability and inter-process communications and is not known until the final instance has completed. At any time, whilst instances are running, it is possible for additional instances to be initiated providing the ability to do so had not been disabled. A completion condition is specified which is evaluated each time an instance of the task completes. Once the completion condition evaluates to true, the next task in the process is triggered. Subsequent completions of the remaining task instances are inconsequential and no new instances can be created.

Dynamic Partial Join for Multiple Instances is supported in BOPM. It can be achieved initializing new process objects, synchronizing, and controlling them with the conditional process action managed by function.

Example: The dispatch of an oil rig from factory to site involves numerous transport shipment tasks. These occur concurrently and although sufficient tasks are started to cover initial estimates of the required transport volumes, it is always possible for additional tasks to be initiated if there is a shortfall in transportation requirements. Once 90% of the transport shipment tasks are complete, the next task (invoice transport costs) can commence. The remaining transport shipment tasks continue until the whole rig has been transported[14].

Process object: Transportation

State:  Lunching transports -> Action: Transports lunched -> State:  Waiting for transports to get there

State:  Waiting for transports to get there -> Action: Other transports needed ->  Lunching transports

State:  Waiting for transports to get there ->  Action: 90% of shipment tasks are complete ->  Waiting for transports to get there with issued invoice

State:  Waiting for transports to get there with issued invoice ->  Action: All transports finished -> State: Done transportation

2.37 Pattern 37 (Local Synchronizing Merge)

> The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. Determination of how many branches require synchronization is made on the basis on information locally available to the merge construct. This may be communicated directly to the merge by the preceding diverging construct or

[14] http://www.workflowpatterns.com/patterns/control/new/wcp36.php

> alternatively it can be determined on the basis of local data such as the threads of control arriving at the merge.

The convergence of two or more branches which diverged earlier in the process into a single subsequent branch is done by synchronization state. Determination of how many branches require synchronization is made on the basis on information locally available to the merge construct and can be achieved by using the conditional process action and implemented by function.

2.38 Pattern 38 (General Synchronizing Merge)

> The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when either (1) each active incoming branch has been enabled or (2) it is not possible that any branch that has not yet been enabled will be enabled at any future time.

General Synchronizing Merge can be achieved in BOPM by conditional process action implemented by function.

2.39 Pattern 39 (Critical Section)

> Two or more connected subgraphs of a process model are identified as "critical sections". At runtime for a given process instance, only tasks in one of these "critical sections" can be active at any given time. Once execution of the tasks in one "critical section" commences, it must complete before another "critical section" can commence.

Limiting concurrent execution of a set of activities can be achieved by custom Process object that manages the concurrent executions.

Example: Both the take-deposit and insurance-payment tasks in the holiday booking process require the exclusive use of the credit-card-processing machine. Consequently only one of them can execute at any given time.

Process object: Credit-card-processing machine

State:  Waiting for task to execute -> Action:  Get task from stack -> State: Executing action

State: Executing action -> Action:  Add task to stack -> State: Executing action

State: Executing action -> Action: Task finished -> State: Waiting for task to execute

2.40 Pattern 40 (Interleaved Routing)

> Each member of a set of tasks must be executed once. They can be executed in any order but no two tasks can be executed at the same time (i.e. no two tasks can be active for the same

> process instance at the same time). Once all of the tasks have completed, the next task in the process can be initiated.

Limiting concurrent execution of a set of activities can be achieved by custom Process object that manages the concurrent executions.

Example: The check-oil, test-feeder, examine-main-unit and review-warranty tasks all need to be undertaken as part of the machine-service process. Only one of them can be undertaken at a time, however they can be executed in any order.

Process object: Service machine

State: Available for task -> Action: Perform check oil task -> State: Checking oil

State: Available for task -> Action: Perform test feeder task -> State: Testing feeder

State: Available for task -> Action: Perform examination of main unit -> State: Examining main unit

State: Available for task -> Action: Perform reviewing warranty -> State: Reviewing warranty

State: Available for task -> Action: All checks has finished -> State: Doing shipping

State: Checking oil -> Action: Checking has finished -> State: Available for task

…

State: Reviewing warranty -> Action: Reviewing has finished -> State: Available for task

2.41 Pattern 41 (Thread Merge)

> At a given point in a process, a nominated number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution.

Supported in BOPM by management of sub processes by one main Process object with conditional process action. After the execution of the conditional process action, new state should send cancel event for unfinished sub processes.

Example: Instances of the register-vehicle task run independently of each other and of other tasks in the Process Enquiry process. They are created as needed. When ten of them have completed, the process-registration-batch task should execute once to finalise the vehicle registration system records update.

Process object: Registration

State: Waiting for new vehicle to register -> Action: New vehicle registered -> State: Waiting for new vehicle to register

State: Waiting for new vehicle to register -> Action: 10 vehicles registered -> State: finalise records update

2.42 Pattern 42 (Thread Split)

> At a given point in a process, a nominated number of execution threads can be initiated in a single branch of the same process instance.

Thread split is done with the lunching of new process objects.

Example: At the completion of the confirm paper receival task, initiate three instances of the subsequent independent peer review task[15].

Process object: Paper

State: Paper has been received -> Action: All reviews have been received -> State: Deciding if the paper should be published

2.43 Pattern 43 (Explicit Termination)

> A given process (or sub-process) instance should terminate when it reaches a nominated state. Typically this is denoted by a specific end node. When this end node is reached, any remaining work in the process instance is cancelled and the overall process instance is recorded as having completed successfully, regardless of whether there are any tasks in progress or remaining to be executed.

When process object gets to the Termination state, it SHOULD cancel all instances of other process objects it has created.

Example: When payment has been done, invoice has been paid.

Process object: Invoice

State: To be paid -> Action (Manager): Payment for invoice received -> State: Invoice has been paid

[15] http://www.workflowpatterns.com/patterns/control/new/wcp42.php

3 Conclusion

This paper has shown, that the BOPM is the notation in which it is able to implement all work flow patterns. Until now I am not aware of any other notation that would implement all workflow patterns, therefore I believe that the overview on the processes as the set of states of the objects is the breakthrough in the process modeling. To support my point, please note that what is described in this document is already mostly implemented and works at CEB[16].

[16] http://www.sk.test.mon.kbb.sk/process?export=578e05c5edaeebbb30762c870cd8a345&type=svg

4 List of pictures