OPAALS PROJECT

Contract n° IST-034824

WP2: Automatic Code Structure and Workflow Generation from Models

Del2.4 - Prototypical implementation of dynamic workflow reconfiguration

Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: IST-034824

Project Acronym: OPAALS

Deliverable N°: D2.4

Due date: November, 2008

Delivery Date: 6 March 2009

Short Description: This report provides a theoretic study about dynamic workflow reconfiguration and a description of a prototypical implementation for changing a workflow model at run-time while its activities are actually executing.
A theoretical basis and a graphical notation have been introduced, allowing to specify a practical solution to those simple cases of dynamic reconfiguration. These cases were classified by defining patterns of reconfiguration. For each pattern have been identified the conditions that must be fulfilled in order to carry out the updating of a workflow model through the application of the same patterns, without any data loss or inconsistencies. The study has produced, in addition, a software component that verifies the admissibility of the reconfiguration of the modifications made to the model, identifying the applied reconfiguration patterns and evaluating the conditions of applicability. If the update is enabled, the component allows the migration of running instance from the old to the new model without loss of data.

Author: T6 ECO (Antonella Filieri, Antonio Margarito)

Partners contributed: T6 ECO

Made available to: OPAALS Consortium and European Commission

Versioning		
Version	**Date**	**Name, organization**
0.1	17/12/2008	T6 ECO
0.2	09/02/2009	T6 ECO
0.3	20/02/2009	T6 ECO
0.4	27/02/2009	T6 ECO

Quality check

Internal Reviewers: Paul Krause (UNIS), TV Prabhakar (IITK)

Dependences:

Achievements*	In the reported period T6 ECO has be developed: - a theoretic study about workflow reconfiguration issues, identifying the most important reconfiguration patterns and providing a graphical notation to visualize and manage them. - a deep analysis of Bonita workflow Engine features/functionalities in order to evaluate the possibility to enhance and modify it basing on the workflow reconfiguration requirements - a prototypical implementation of a module that is able to perform a comparison between two versions of process definition model in order to recognize reconfiguration patterns. The output is used to distinguish between admissible or not-admissible instance modifications. In the reported period T6 ECO did not develop: - a complete solution for the implementation of a dynamic workflow reconfiguration due to some critical aspects related to the lack of customization features of the modern open source workflow engines.
Work Packages	This deliverable is connected to activities of WP3 that deals with evolutionary networks of SMEs. Such scenario is characterized by the necessity of handling inter-organizational re-configuration of workflows in order to react to environmental variations.
Partners	University of Surrey (UniS), Indian Institute of Technology Kanpur (IITK), University of Kassel (UniKassel), University of Central England Birmingham (UCE).
Domains	Computer science domain.
Targets	Computer Science researchers, SMEs and public administrations.
Publications*	The reported work has still not been published.
PhD Students*	

Outstanding features*	With a precise analysis of workflow reconfiguration requirements and with an attempt to implement them, we are able to make a positive contribution to the state of art. Moreover the work done could have a significant impact to how effectively and practically support the changes of a business process model with positive effects for SMEs.
Disciplinary domains of authors*	Antonella Filieri: computer science and business modelling domain Antonio Margarito: computer science and business modelling domain

The information marked with an asterisk (*) is provided in order to address Recommendation n. 4 from the Year 2 review report

Table of contents

Figures and Tables

1 Introduction

This work is part of WP2 and contributes to its second-phase activities. The objective of WP2 is to provide a set of guidelines and conceptual/technological means to allow automatic and semi-automatic generation of software systems from natural language based specifications [1] [2].

This work starts from the results of the "Extended vocabulary and rule set for an existing scenario" documented in D2.3 [3], that showed how to obtain a process model, in XPDL format, starting from its description in SBVR SE.
As described in the previously delivered documents related to this WorkPackage [see OPAALS D2.2 and D2.3], enacting a workflow model that is defined in a business-oriented language (for us SBVR SE), requires the transformation of such model to an equivalent one expressed in a language interpretable by a workflow execution engine (e.g. XPDL, BPEL, jBPL and others).

We have chosen the Bonita Project as workflow engine because it is the most promising active project of Open Source Java-based Workflow Engine and also because this has been recommended[1] by the EU Open Source Observatory in the ambit of the EU's programme called IDABC[2], which stands for Interoperable Delivery of European eGovernment Services to public Administrations, Business and Citizens and contributes to the "i2010 initiative" of modernising the European public sector. IDABC is a Community programme managed by the European Commission's Directorate-General for Informatics (DIGIT)[3]. The aim of the project is to improve efficiency and collaboration between European public administrations by using state-of-the-art information and communication technologies, by developing common solutions and services and by providing a platform for the exchange of good practice

[1] http://ec.europa.eu/idabc/en/document/5113/5813

[2] http://ec.europa.eu/idabc/en/chapter/3

[3] http://ec.europa.eu/dgs/informatics/contact/index_en.htm

between public administrations and to encourage and support the delivery of cross-border public sector services to citizens and enterprises in Europe.
For completeness an overview of the architecture of the Nova Bonita workflow engine is provided in Appendix 1.
In this deliverable, instead, the starting point is the obtained process model and its enactment and management within a workflow engine. In particular, we propose a discussion about the possibility to implement a dynamic workflow reconfiguration, because we believe that it is a key issue for an enterprise that wants to maintain adequate levels of competitiveness.
The importance to be able to change the characteristics of a workflow during its actual execution (i.e. at run-time) derives essentially from two factors:

- It is hard to create a workflow model that perfectly reflects reality, thus requiring adjustments even at an early stage of its execution, when imperfections of the model become evident;
- Conditions that are external or internal with respect to the business (business goals, business policies, rules) change during the execution of the workflow, requiring some kind of reaction and adaptation. Note that the probability to incur in a change of conditions increases proportionally with execution-time, number of activities, number of participants and, in general, with all those elements that contribute to the overall complexity of the workflow model.

In real business scenarios, workflows are often characterised by high complexity, long execution-time, but rigid mechanisms of workflow enactment that inhibit any modification of running workflow instances. The ability to quickly and adequately react to business process changes is a key feature to guarantee/preserve a high level of competitiveness for any enterprise. We refer to the adaptation necessity as *dynamic reconfiguration requirements*.

Before starting with the description a prototypical implementation, it is important to provide a formal foundation and a theoretical analysis of the principles that underpin dynamic workflow reconfiguration.

Section 2 deals with the dynamic workflow reconfiguration issue and shows some patterns that allow modeling some possible change scenarios.
Section 3 describes the behaviour and the architecture of the software component that performs the verification of the reconfiguration admissibility.
Appendix A provides an overview of Nova Bonita workflow engine focusing on the PVM (Process Virtual Machine) tier and on the workflow instances management.

2 Dynamic workflow reconfiguration

In this section we firstly expose the dynamic workflow reconfiguration issues and secondly show a set of patterns that allows modeling some possible change scenarios.
Before starting with the description, it is necessary to provide an overview of the terminology and notation that are used to better expose our work.
In the following paragraph, we will briefly introduce those concepts and explain in details the way in which they are used in our work.

2.1 Adopted terminology and notation

In line with the previous choices [2] [3], we decided to formulate/define most of the concepts presented in this section basing mainly on the XPDL specification [4], but also on the Bonita Reference.
Process Model (or **Process Definition** according to the XPDL specification terminology) is one of the most important concepts because it contains the workflow definition logic (i.e. the elements that makes up a workflow).
Note that it is essential to distinguish the concept of Process Definition from that of **Process Instance** that represents, instead, a specific execution of a workflow process.
Both process definition and process instance are characterised by a specific life cycle. In particular, a process can have a life cycle made up of two possible states: **Deployed** or **Undeployed**. The former refers to the case is which the process is successfully deployed, that is the process is created into the engine. The latter refers to the case in which the process is successfully removed from the list of executable or executed processes.
A process instance, instead, has a life cycle that consists of three states: **Initial, Started** and **Finished**. When the process instance has been created, its state is set to Initial. If the execution automatically begins then the state becomes started. The Finished state is reached when the last activity of a workflow process is performed.

In order to completely describe the process instance state, it is also necessary to consider the state of each activity that composes such process instance.
Moreover, the concept of state, at activity level, depends on the activity types: **Manual, Automatic** and **Route**.
Manual activities require explicit user interaction in order to start or finish the activity execution. According to the Bonita reference terminology, we refer to manual activities also as **Task**.
A route activity is a sub element of Activity and is used to implement routing logic.
Automatic activity, instead, is fully controlled by the workflow engine that automatically performs it as soon as any condition of the incoming transitions is satisfied.
Between the above listed activity types, only manual activities have an own life cycle, made up of six states: Ready, Initial, Executing, Suspended, Resume, Finished.
Initial is the default state and means that a task is not yet ready to be executed. To indicate this state we use the visual notation showed in Figure 1.

Figure 1 – Graphical notation of the Initial state

The state can shift to Ready when a task can be executed, i.e. one of the following situations occurs:

- there is no ingoing transitions;
- ingoing connected activities are successfully finished and transition conditions were evaluated to true.

We use the graphical notation in Figure 2 to indicate a task that in Ready state.

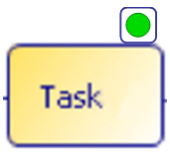

Figure 2 – Graphical notation of the Ready state

When a task is under execution its state is **Executing.** The visual notation of Executing state is showed in Figure 3.

Figure 3 – Graphical notation of the Executing state

It is possible to temporarily stop a task when it is in Ready state or while it is Running, in both cases the task state becomes Suspended. The graphical notation of Suspended state is showed in Figure 4.

Figure 4 – Graphical notation of the Suspended state

The Finished state, instead, is reached when the task has been successfully concluded. Figure 5 shows the used graphical notation for Finished state.

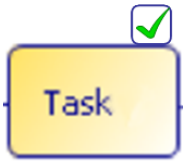

Figure 5 – Graphical notation of the Finished state

Figure 6 shows an UML state diagram in which all possible task state transitions are represented. As we can see, bidirectional transitions are only possible between Ready and Suspended state and between Executing and Suspended state.
The transitions from Initial to Ready and from Executing to Finished are not reversible whereas the transition from Ready to Executing is not directly reversible.

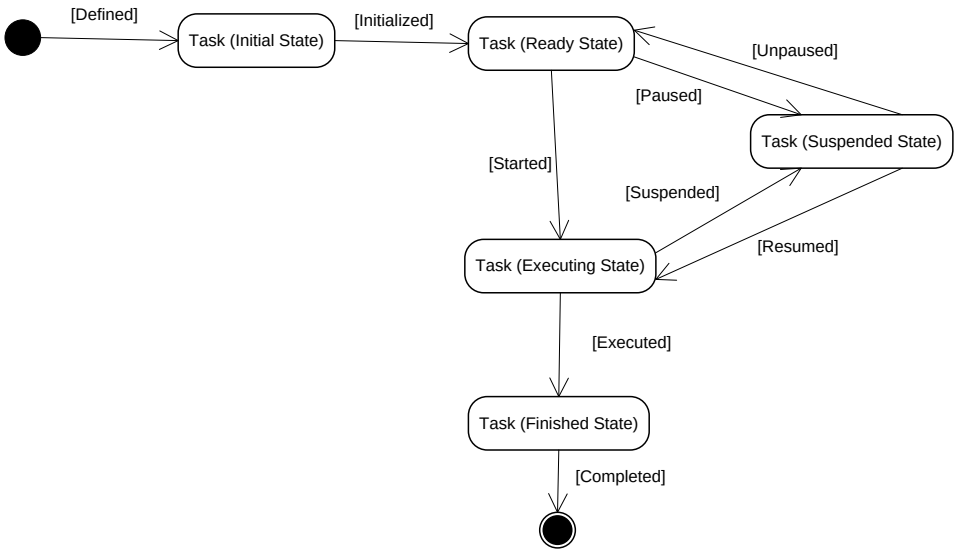

Figure 6 – Task Lifecycle State Diagram

Using the above introduced notation, Task lifecycle can be represented as showed in Figure 7:

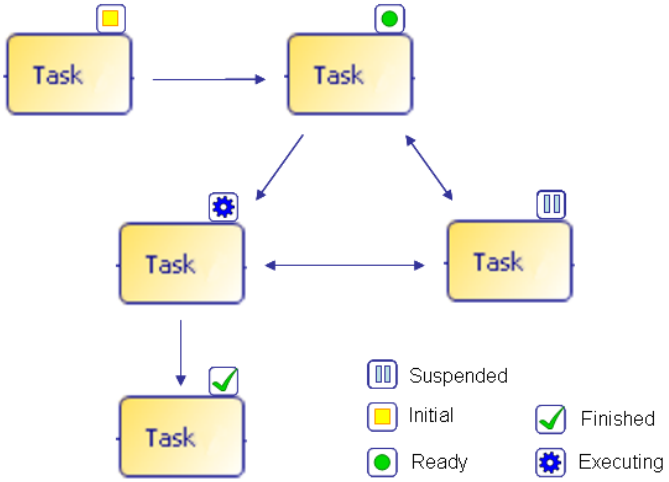

Figure 7 – Task Lifecycle in new notation

Basing on the definitions provided above, it is possible to graphically represent the running instances of a process, in any time instant, enriching the process model, from which the instances has been created, with explicit information about the execution status of each task, as showed in Figure 8.

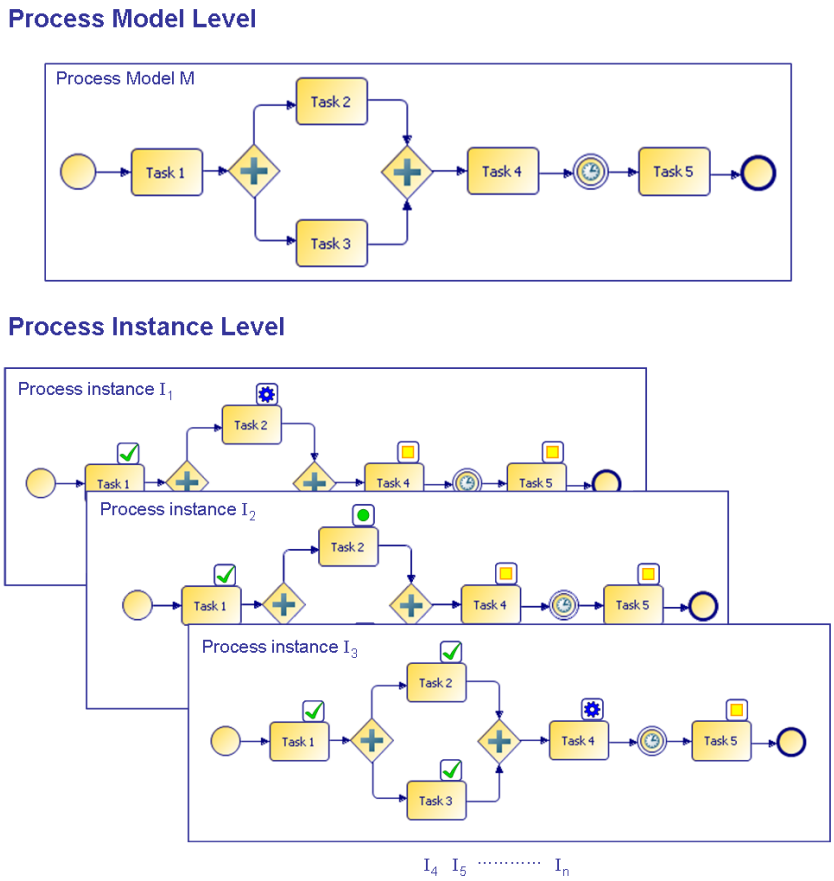

Figure 8 – Process Model and Process Instances

2.2 Dynamic workflow

Dynamic workflow reconfiguration is about the possibility to change the flow of a process instance (at runtime) and to evaluate the effects that such alterations could generate. Changes may happen/occur on two different levels: process model level and process instance level [9].

If the change influences just one instance of a workflow, then it is sufficient to insert an additional logic to manage the modifications in an ad-hoc manner. But this scenario is not very interesting because in most cases changes have to be applied to all workflow instances in execution or to a subset of instances based on specific filtering criteria. There are many approaches supporting such adaptive workflow requirements [5, 6, 8].

Therefore, in the following we use the term “dynamic workflow reconfiguration” to indicate the problem of handling the running instances of a workflow when a new, i.e. improved, version of workflow process definition is introduced.
In other word, it is important to understand if and how it is possible to transfer the old instances to a new improved version of the process definition.
To better explain this problem, we provide an example using a part of the ATM withdrawal workflow process [3], as showed in Figure 9. The objective is to evaluate the possibility to change the parallel workflow process (top section of Figure 9) into a workflow process where tasks *Deliver money* and *Update account* are executed sequentially (down section of Figure 9) or vice versa. We will demonstrate that, in this scenario, it is always possible to shift from the sequential workflow process to the parallel one, whereas the contrary is not always true. In the following we use the terminology, described in paragraph 2.1, to indicate the state in which an instance or a task may be.

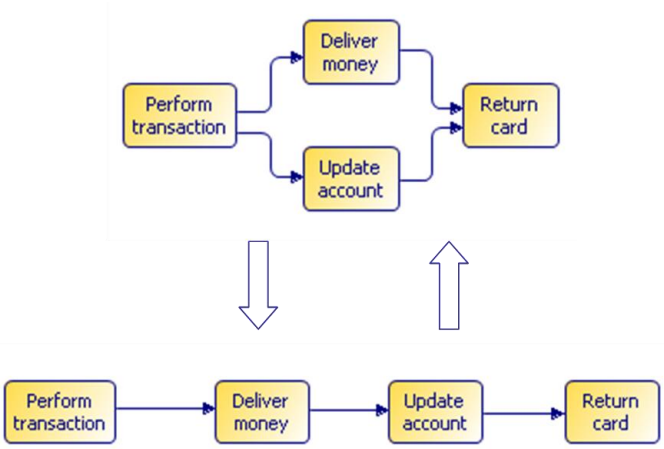

Figure 9 – The dynamic workflow reconfiguration problem

From sequential to parallel

The sequential process may be in several execution points when a modification in the process definition occurs. Each state of the sequential process can be mapped on a state in a parallel process. We suppose, for example, that the state of *Deliver money* task is Running while the state of *Update account* task is Initial, as showed in the down-section of Figure 10. This scenario can be mapped into the parallel instance state in which *Perform transaction* task has been performed, *Deliver money* is running and both tasks *Update account* and *Return card* still need to be executed,

as showed in the top section of Figure 10. It is easy to verify that any possible combination of the task states, which identifies the instance state, in the sequential process can be mapped into a parallel instance state.

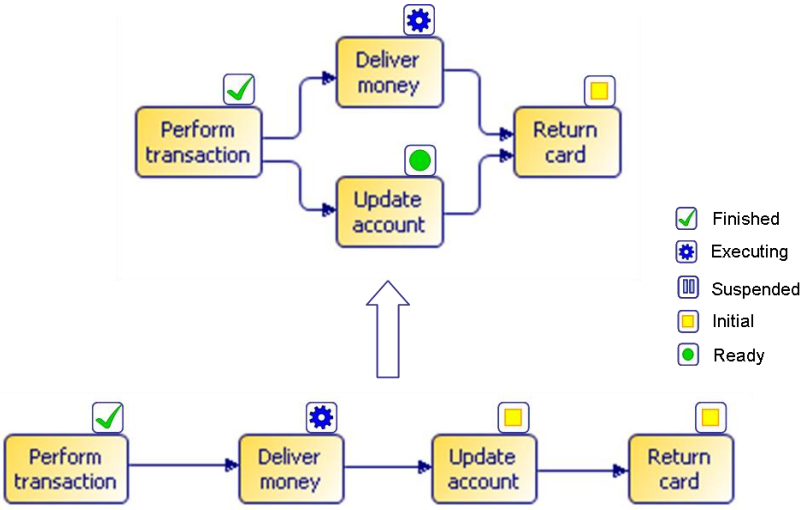

Figure 10 – From sequential to parallel process

From parallel to sequential

Now consider the inverse scenario, where the parallel process is changed into the sequential one, i.e. from top to down section of Figure 9. For most of the states of the parallel process, the transition to the sequential version does not cause any problems.

However, we consider the parallel instance state in which *Deliver money* task state is Ready and *Update account* task state is Running, as showed in the top-section of Figure 11. As we can simply verify there is no corresponding state in the sequential process because it is not possible to execute *Update account* task before *Deliver money* task in the sequential process.

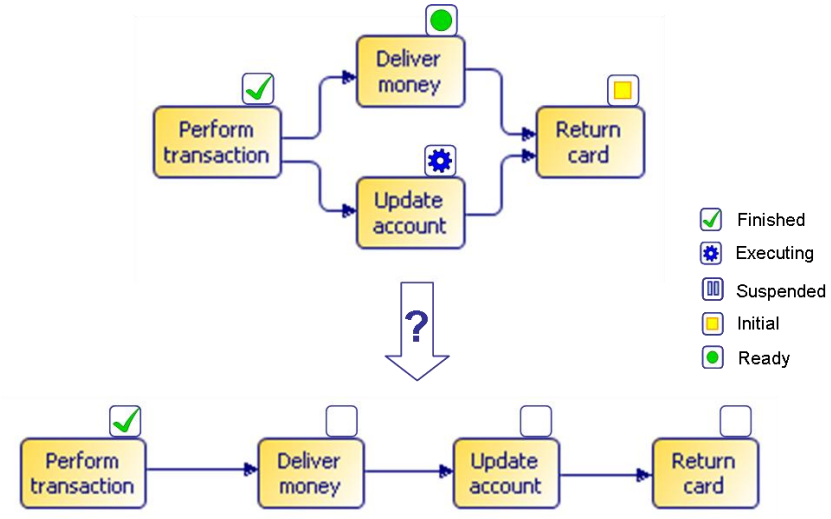

Figure 11 – From parallel to sequential process

Similar problems may occur when the order of tasks is changed, for example when two sequential tasks are swapped.

Finally, with the above example we demonstrate how the migration of a running instance, in order to follow a changed version of process definition, is not a straightforward issue.

2.3 Reconfiguration issue

The key issues regarding workflow reconfiguration [5] are related to the preservation of the soundness of Control-flow between the activities and of the consistency and correctness of Data-flow, during a running instance modification.

A process model, in fact, is not just about coordinating activities but also about managing the data that are used by these activities.

In particular two kinds of data are directly managed inside a process model. Process data and activity data. Process data are properties/variables that can be accessed and changed by all the activities of a process. Activity data, instead, act as input and output parameters. Each activity has a list of input and output data. These data are passed from an activity to another during process execution.

Consistency and correctness, both from control-flow and data-flow perspective, may be altered by the addition, deletion and movement of tasks and in general by the modification of the process model.

As explained in the previous paragraph, it is important to carefully consider the effect of run-time alteration of the running process instances in order to preserve their correctness, both from control-flow and data-flow perspective.

In order to be correct, an instance not only has to execute all the activities according to the corresponding workflow model but also has to satisfy all the constraints imposed by the workflow model.

From a control-flow perspective, it is important to assure that any modification to the model is reflected to actual running instances, allowing such modifications to be applied only to those workflow sections that must still be executed. This requires a precise specification of a workflow execution progress in terms of tasks state. Moreover it is important not to affect executing tasks in order to prevent any sort of issue in the PVM environment.

From a data-flow perspective, it is important to allow only those modifications that do not introduce data-inconsistency during run-time. Note indeed that there is data dependency among tasks since variables and constants can be defined, initialized, modified and deleted in any point of the executing workflow. A simple example of data-dependency is the case in which a variable is initialised/defined in a task and modified/deleted in a succeeding one. The second task depends on the first one, thus it would be impossible to delete the first without causing inconsistency and run-time errors in the workflow.

In line with this, it is important to analyse all the functions that may determine problems to a process instance. For example, the opportunity to add a task, to alter the predefined task sequences, to skip tasks with or without waiting their end, to transform two sequential tasks to parallel ones and vice versa, or even to delete a task or to modify its attributes that could imply significant inconsistencies specially if the task is already ended.

In this work we mainly focus on control-flow perspective and we refer to [5] [7] for further details about data-flow consistency issues.

Finally, the problem of workflow reconfiguration can be handled considering two aspects. The first one is to entirely detect and take into consideration all the structural dependencies between tasks, related either to data or temporal dependences, whose modifications can impact on running instances. The second one is to consider the

execution state of an instance at the time of the modification of its model. For example if the new process definition requires that some additional steps have been executed before an instance can legitimately enter a specific execution state *S*, then all instances being in state *S* (or in a subsequent state) cannot follow the new model.
Therefore, a primary distinction can be made between admissible and not-admissible running process instances. The former case occurs when an existing process instance can be transferred, i.e. the running instance state can be mapped to a corresponding state in the new process model. The latter case occurs if it is not possible to shift the running instance without causing errors or inconsistencies, i.e. the running instance state cannot be mapped in any states of the new process model.
In the following section we provide a set of patterns modeling some modification scenarios and a set of corresponding rules that allow evaluating the transferring suitability of a running instance. These rules, based on the preservation of the correctness and consistency of a process instance, can be used as a parameter to decide if a specific change can be applied or not, i.e. if a running instance can be considered admissible or not-admissible.
If a change cannot be applied on a running instance, i.e. it is not-admissible, it is necessary to decide how this situation has to be managed. In other words it is indispensable to define a strategy that specifies how to handle the running process instances that could not migrate to the new process model.
The solution in such case could be to abort running instances that cannot be migrated. This approach is often inconvenient or impossible to be applied, above all if the instances have been running for a long time thus implying the loss of their related information. Furthermore, in certain processes, it is impossible to stop the activities in order to apply the change.
Another solution could be an ad-hoc adjustment for each instance, but also such solution is not practicable above all if the number of instances to be handled is high.
In our work we decided to provide different versions of process models such that those instances that cannot be migrated to the new version can remain with the old version.

2.4 Reconfiguration patterns

In this section we provide a set of reconfiguration patterns [10] that are able to model some structural change in the process definition. Moreover, for each pattern we define some rules that allow distinguishing if updating an instance can be an admissible or not-admissible operation.

For each pattern we provide a name, a brief description of the problem it addresses, and a graphical example. Moreover, for each pattern we identify a set of admissibility conditions, i.e. the conditions that each running instance must satisfy in order to be migrated to a new version of process model.

Reconfiguration patterns *Insert Process Activity* and *Delete Process Activity* allow respectively the insertion and deletion of process activity in a given process model. Moving and replacing activity is supported by reconfiguration patterns *Shift Process Activity* and *Swap Process Activity*.

2.4.1 Insert Process Activity Pattern

Description The *Insert Process Activity* pattern can be used to add an activity to a process model. One of the most important features regards the position at which the new process activity has to be embedded. In the following we refer to it as insertion point. A task can be added between two directly succeeding activities as showed in Figure 12. We refer to this scenario as *Sequential Insertion*.

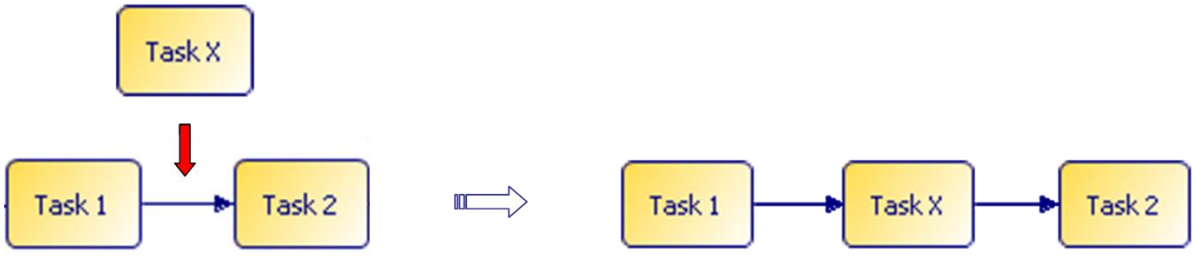

Figure 12 – *Sequential Insertion* pattern

Moreover the insertion can be also parallel, i.e. the new process activity has to be executed in parallel to another one, as showed in Figure 13. We indicate the first

insertion point as split insertion point and the second as merge insertion point. We refer to this scenario as *Parallel Insertion* pattern.

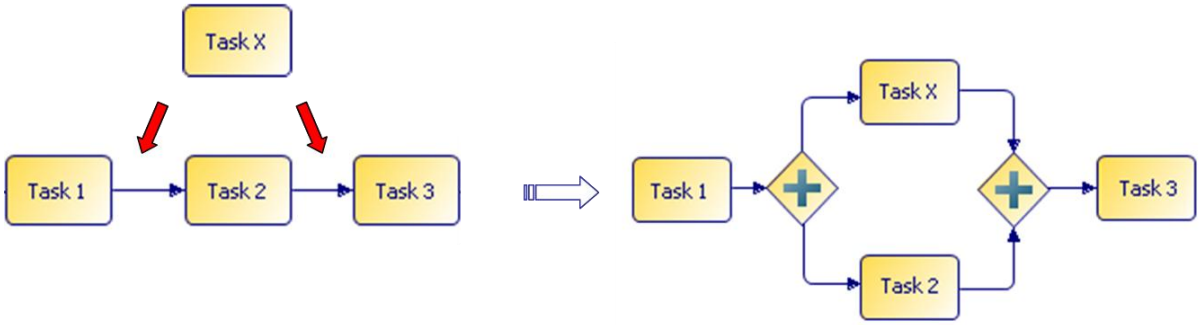

Figure 13 – *Parallel Insertion* pattern

Finally an insertion can be conditional, i.e. the new process activity has to be performed as an alternative to another one basing on an additional condition, as showed in Figure 14. We name it *Conditional Insertion* pattern.

Figure 14 – *Conditional insertion* pattern

Admissibility conditions

Modify a running instance applying a *Sequential Insertion* or *Conditional Insertion* pattern is an admissible operation only if all tasks that follow the insertion point are in the “initial” state. In Figure 15 a possible admissible scenario is showed.

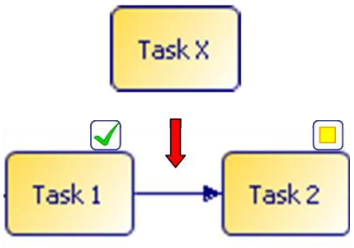

Figure 15 – Admissible scenario for *Sequential Insertion* pattern

The insertion (sequential or conditional) is, instead, considered not-admissible if after the insertion point there is even only one task whose state is “ready”, ”finished” or “executing”. Figure 16 shows some of all possible not-admissible scenarios.

Figure 16 – Not-admissible scenarios for *Sequential Insertion* pattern

A *Parallel Insertion* pattern is considered admissible if all tasks belonging to the running instance which stands after the merge insertion point are in the “initial” state. All tasks between split and merge insertion points can be in any state, i.e. “initial”, “ready” or “executing”. Some admissible scenarios are illustrated in Figure 17.

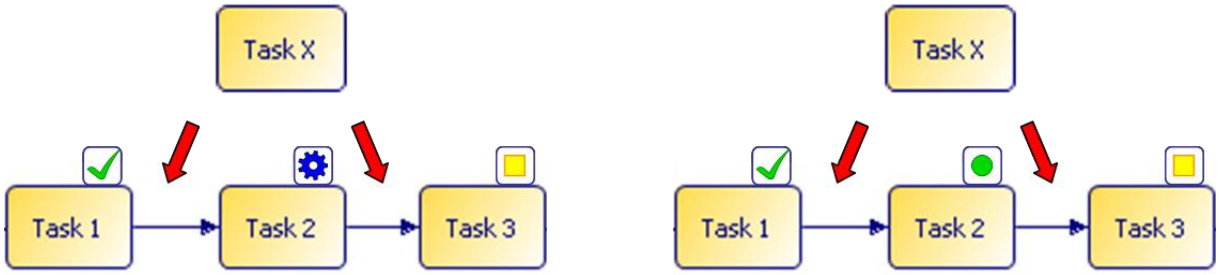

Figure 17 – Admissible scenarios for *Parallel Insertion* pattern

The *Parallel Insertion* pattern is not applicable to a running instance, if tasks subsequent to the merge insertion point are in the “ready”, “finished” or “executing” state. A possible not admissible scenario is depicted in Figure 18.

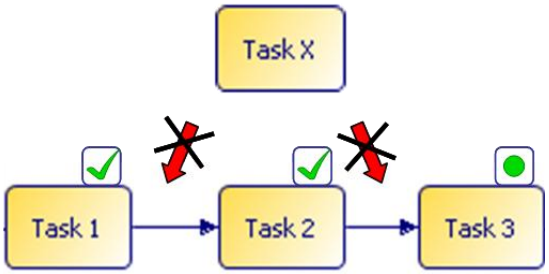

Figure 18 – Not-admissible scenario for *Parallel Insertion* pattern

2.4.2 Delete Process Activity Pattern

Description The *Delete Process Activity* pattern can be used to remove a process activity from a process model. There are several possible scenarios of activity deletion: activities can be deleted if these are in a sequential, parallel or conditional scenario. We can remove a task when it is included between two succeeding activities as showed in Figure 19. We refer to this scenario as *Sequential Activity Deletion.*

Figure 19 – *Sequential Activity Deletion* pattern

If the task we want to remove is executed in a parallel branch, as showed in Figure 20, we talk about *Parallel Activity Deletion* pattern.

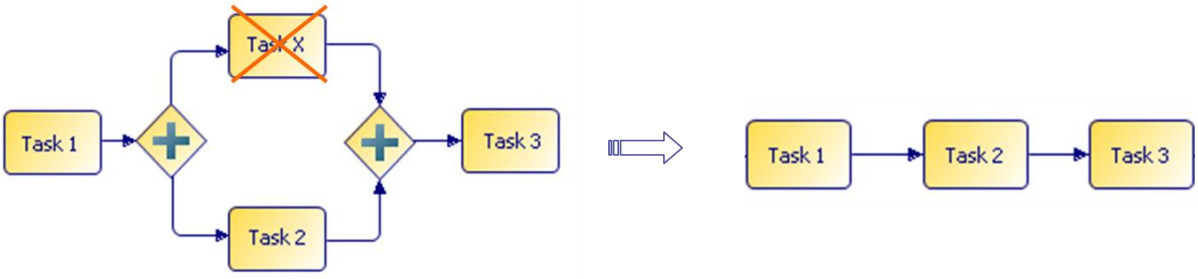

Figure 20 – *Parallel Activity Deletion* pattern

We refer to *Conditional Activity Deletion* pattern when the task to remove comes after a conditional decision point, as showed in Figure 21.

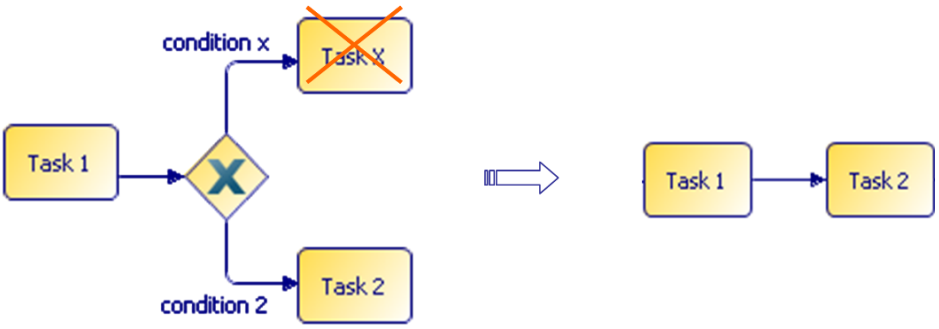

Figure 21 – *Conditional Activity Deletion* pattern

Admissibility conditions

Depending on the scenario, we can have different admissibility conditions for activity deletion.

As regard the deletion in a sequential scenario, modification on a running instance can be admissible if the state of task that must be removed is “initial” or “ready”, as showed in Figure 22.

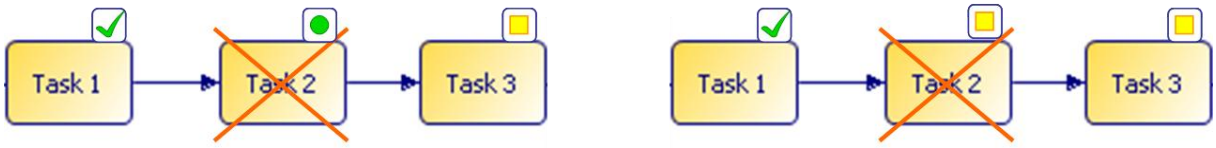

Figure 22 – Admissible scenarios for *Sequential Activity Deletion* pattern

In all other cases (if the state of task that must be removed is “executing” or “finished”), it is not possible to update the running instances. Some possible not-admissible scenarios are depicted in Figure 23.

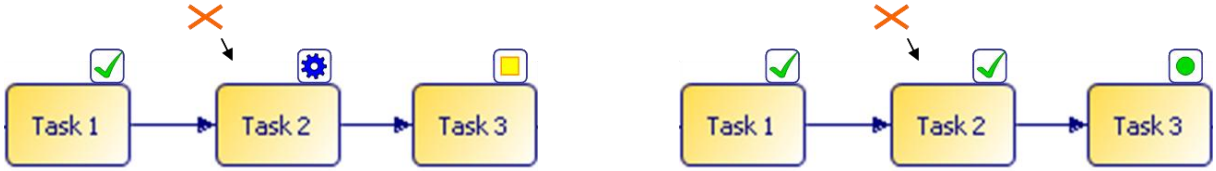

Figure 23 – Not-admissible scenarios for *Sequential Activity Deletion* pattern

As regards the *Parallel Activity Deletion* pattern, if the state of the task we want to remove is “initial” or “ready”, then the modification is admissible and the process instances can be updated. This scenario is showed in Figure 24.

Figure 24 - Admissible scenarios for *Parallel Activity Deletion* pattern

The not-admissible scenarios for *Parallel Activity Deletion* pattern are represented in Figure 25. These scenarios occur when the task we want to remove is in “executing” or “finished” state.

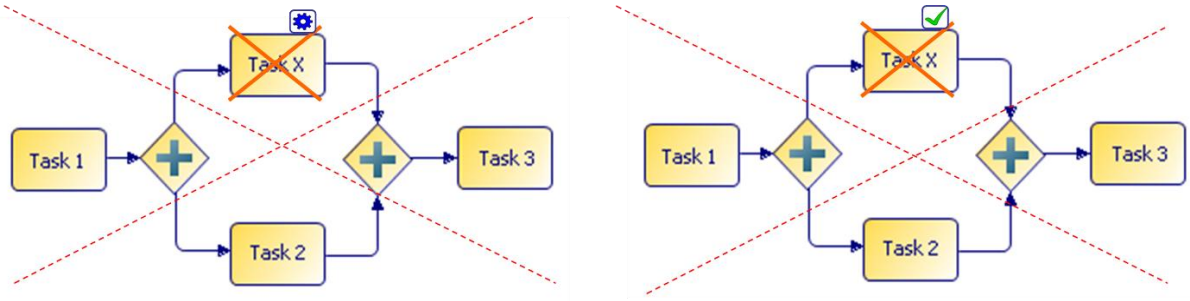

Figure 25 - Not-admissible scenarios for *Parallel Activity Deletion* pattern

The admissibility conditions for the *Conditional Activity Deletion* patterns are more complicated than those related to other scenarios. These conditions strongly depend by the status of the activity that precedes the decision point, namely Task1 if we refer to Figure 21, as well as the state of the task we want to remove. Referring to Figure 26, if the Task 1 state is set to “initial”, “ready” or “executing”, we can in any case remove Task X.

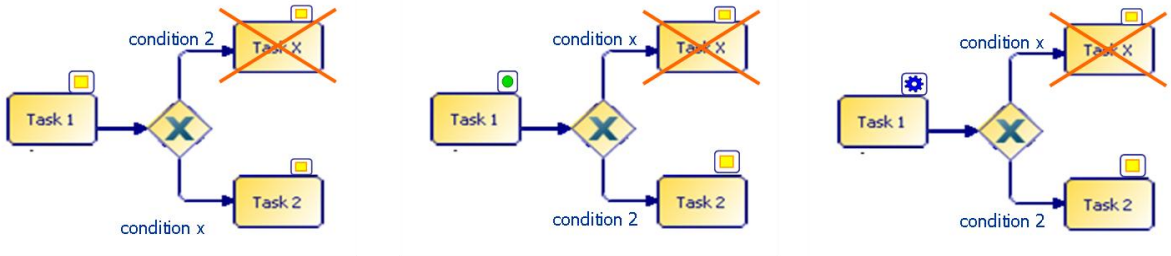

Figure 26 – Admissible scenarios for *Conditional Activity Deletion* pattern

Referring to Figure 27, if the Task 1 state is set to “finished”, the removal of the Task X from the process model is also influenced by the true condition evaluation at the decision point: if condition x is true, then the next task to be executed is Task X. For this reason we can remove it only if this is in “ready” state. Otherwise, if the condition x is false, next task to be executed will be Task 2, and so we can remove Task X without consider its state (it will never be executed). These situations are represented in Figure 28.

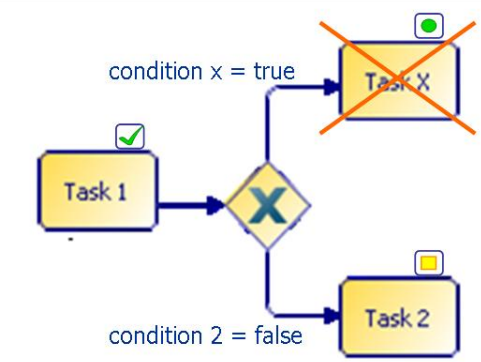

Figure 27 – Admissible scenarios for *Conditional Activity Deletion* pattern (if condition is true)

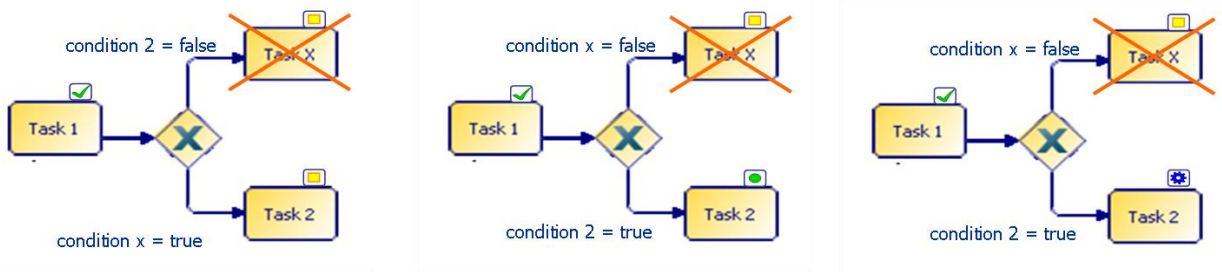

Figure 28 – Some admissible scenarios for *Conditional Activity Deletion* pattern (if condition is false)

In all other cases it is not possible to update the running instances. Some possible not-admissible scenarios are depicted in Figure 29.

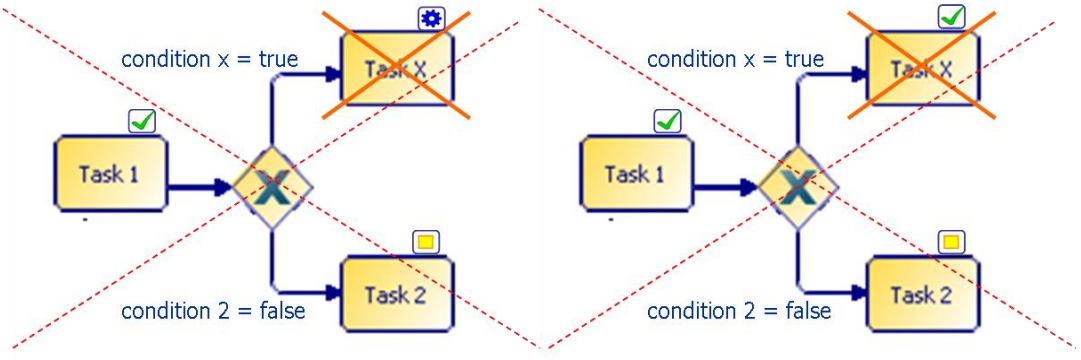

Figure 29 – Some not-admissible scenarios for *Conditional Activity Deletion* pattern

2.4.3 Shift Process Activity Pattern

Description The *Shift Process Activity* pattern allows moving a process activity from its current position to a new one within the same process model. Such pattern could

be obtained by the combined use of Insert/Delete Process Activity patterns but we introduce it as a separate pattern in order to make easy the explanation of the problems related to the shifting operation.

Some considerations are needed about the new position to which the process activity is moved, in the follow we refer to it as target position. If the activity has to be moved between two successive activities with a forwarding movement (see Figure 30) then the movement is called *Sequential Forward Shift*.

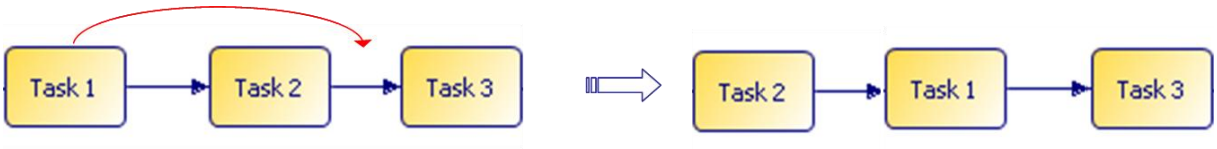

Figure 30 – *Sequential Forward Shift* pattern

If the target position is between two directly succeeding activities and the movement is backward then the pattern is called *Sequential Backward Shift*. This scenario is visualized in Figure 31.

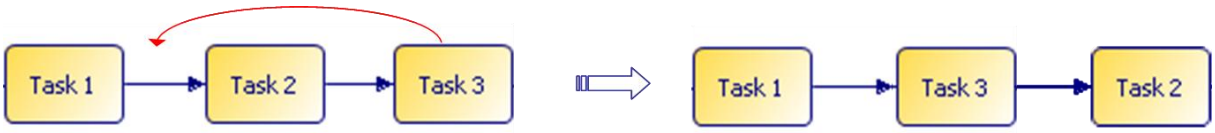

Figure 31 – *Sequential Backward Shift* pattern

The activity can also be moved to a forward target position at which it will be executed as alternative to or in parallel with another activity. We respectively refer to such scenarios as *Conditional Forward Shift* (Figure 32-a) and *Parallel Forward Shift* (Figure 32-b).

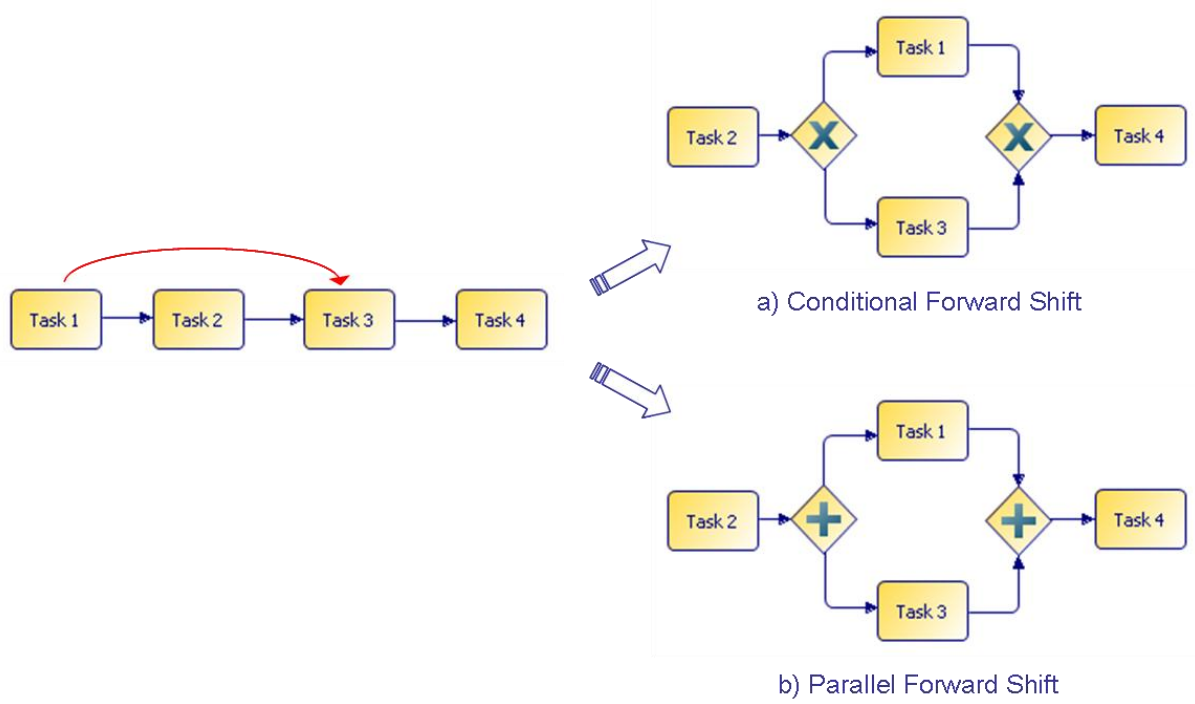

Figure 32 – *Conditional* and *Parallel Forward Shift* patterns

The activity can also be moved to a backward target position at which it will be executed as alternative to or in parallel with another activity. We respectively refer to such scenarios as *Conditional Backward Shift* (Figure 33-a) and *Parallel Backward Shift* (Figure 33-b).

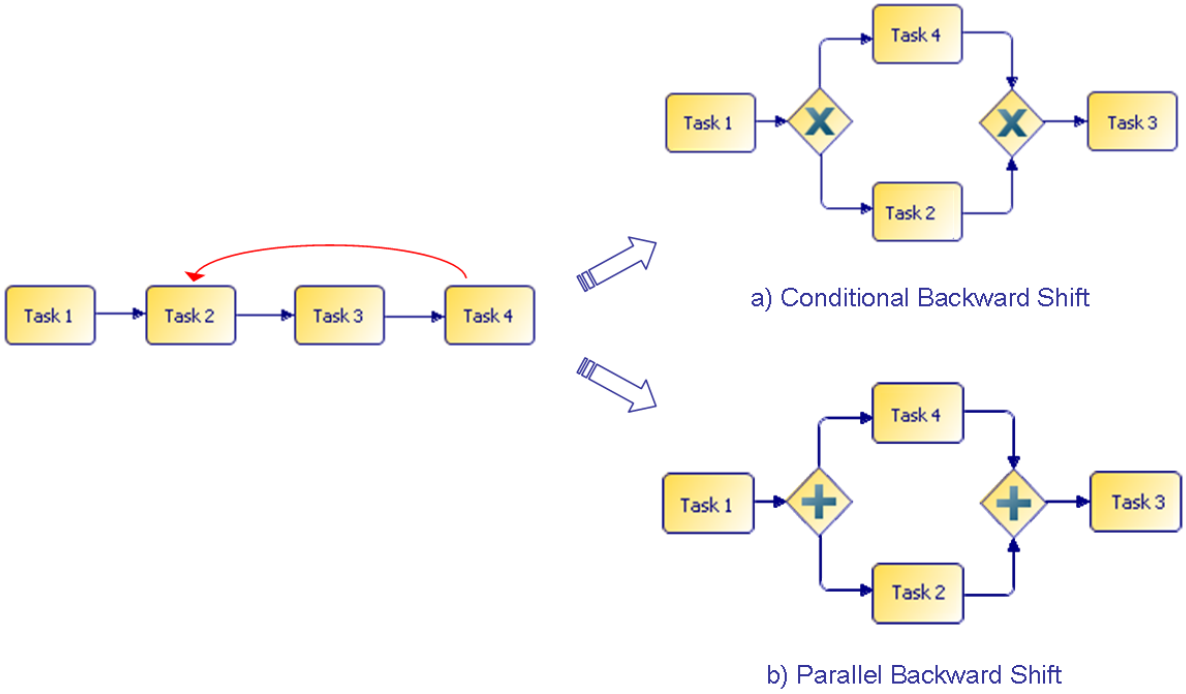

Figure 33 – *Conditional* and *Parallel Backward Shift* patterns

Moreover it is possible to move activities so that parallel or conditional flows can become sequential. We refer to such scenarios as *Conditional-to-Sequential Shift* (Figure 34-a) and *Parallel Backward Shift* (Figure 34-b).

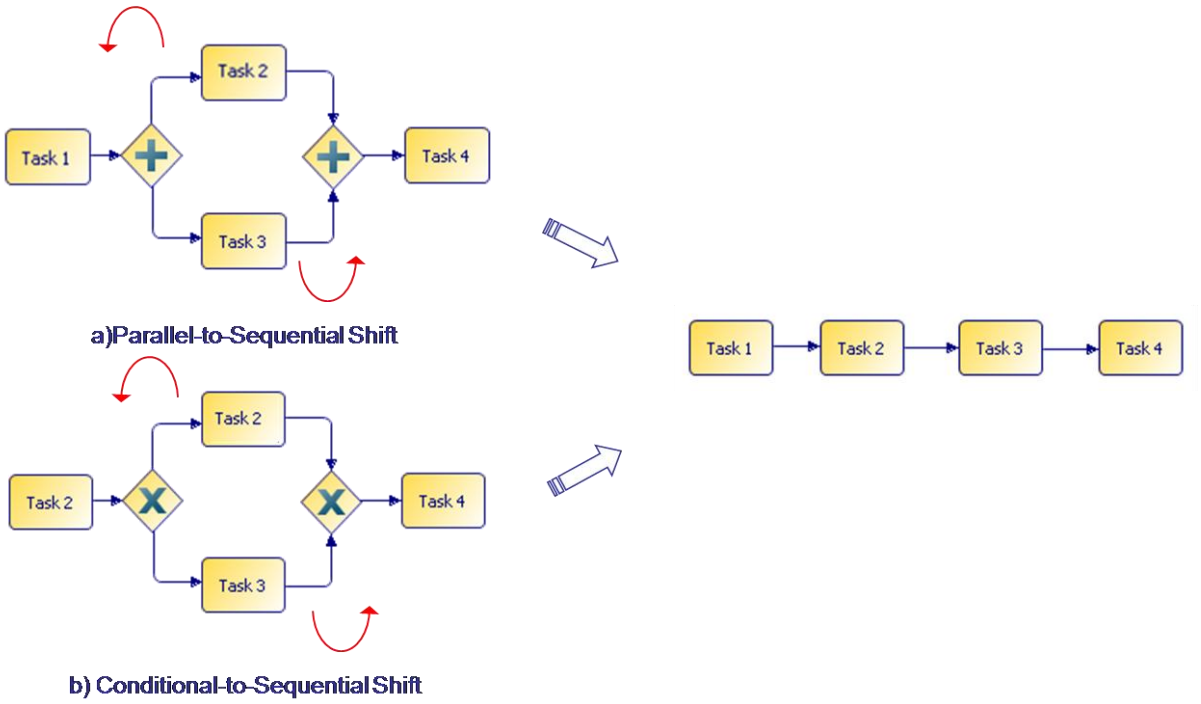

Figure 34 – *Parallel-to-Sequential* and *Conditional-to-Sequential* patterns

Admissibility conditions

As regard *Sequential Forward Shift* pattern, if the state of the task we want to move is “initial” or “ready”, then the modification is admissible and the process instances can be updated. This scenario is showed in Figure 35.

Figure 35 – Admissible scenarios for *Sequential Forward Shift* pattern

If the state of the task to be moved is “executing” or “finished”, then the scenario is not-admissible as showed in Figure 36.

Figure 36 – Not-admissible scenarios for *Sequential Forward Shift* pattern

The admissibility conditions for *Sequential Backward Shift* pattern depend on the state of the activity that strictly precedes the target position. Thus, referring to Figure 37, if the state Task1 is set as “initial”, “ready” or “executing” then the modification is permitted.

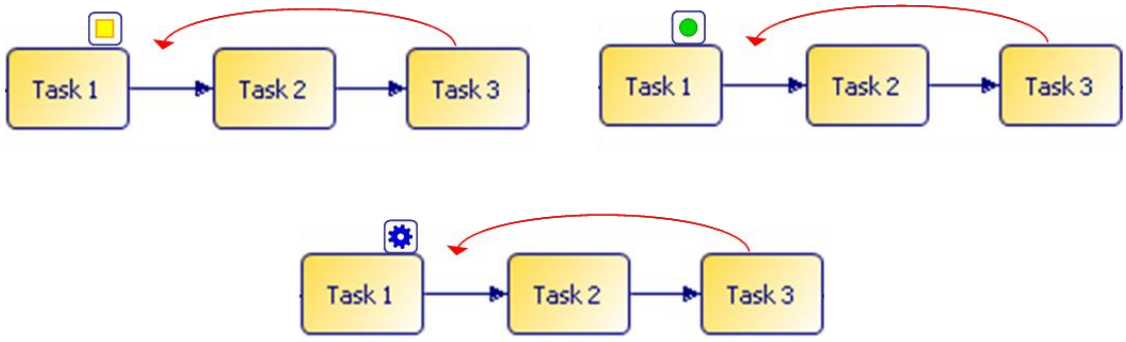

Figure 37 – Admissible scenarios for *Sequential Backward Shift* pattern

As showed in Figure 41, if Task1 state is set to “finished” and Task2 is in the “executing” or “finished” state then the modification is not-admissible because Task2 is already processing or has already processed data, so some problems of data inconsistency can arise.

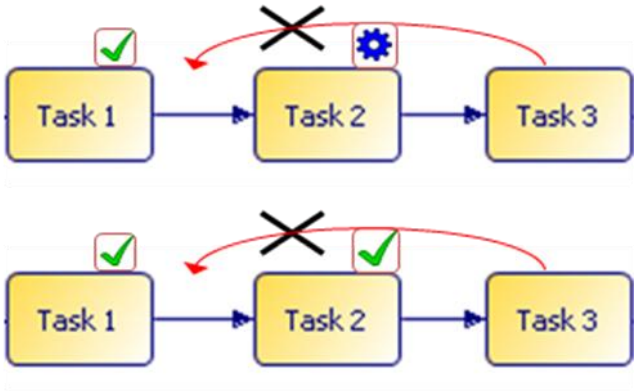

Figure 38 – Not-admissible scenarios for *Sequential Backward Shift* pattern

As regard *Conditional Forward Shift* and *Parallel Forward Shift* patterns admissibility and not-admissibility conditions are the same as those of *Sequential Forward Shift*, as showed in Figure 39 and Figure 40.

Figure 39 – Admissible scenarios for *Parallel* or *Conditional Forward Shift* patterns

Figure 40 – Not-Admissible scenarios for *Parallel* or *Conditional Forward Shift* patterns

The admissibility conditions for *Conditional Backward Shift* pattern are more restrictive with respect to *Sequential Backward Shift*: Such pattern is admissible only if the target task is in “initial” state. This scenario is showed in Figure 41.

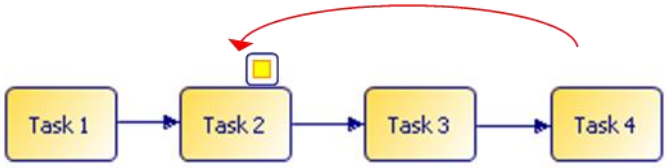

Figure 41 – Admissible scenarios for *Conditional Backward Shift* pattern

This modification is considered not-admissible if the target task is in states different from “initial”. This is showed in Figure 42.

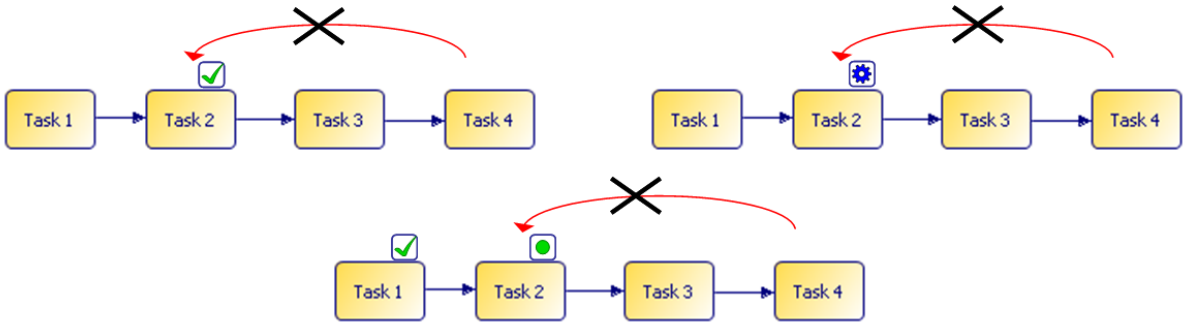

Figure 42 – Not-Admissible scenarios for *Conditional Backward Shift* pattern

As showed by Figure 43, admissibility conditions for *Parallel Backward Shift* pattern require that the target task we want to parallelize is in “initial”, “ready” o “running” state.

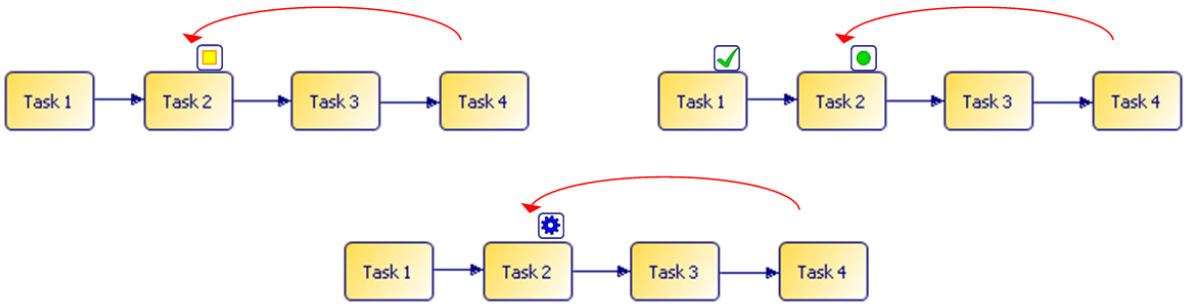

Figure 43 – Admissible scenarios for *Parallel Backward Shift* pattern

The modification is not-admissible only if the target task state is set to “finished”, as showed in Figure 44.

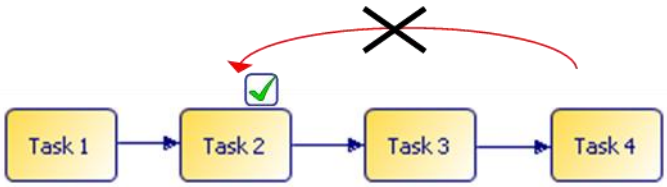

Figure 44 – Not-Admissible scenarios for *Parallel Backward Shift* pattern

In Figure 45 are shown the admissibility conditions valid both for the Conditional-to-Sequential Shift and *Parallel-to-Sequential Shift* pattern. If Task1 is in “initial”, “executing” or “ready” state it is possible to apply the modifications, because Task2 and Task3 are still in “initial” state.

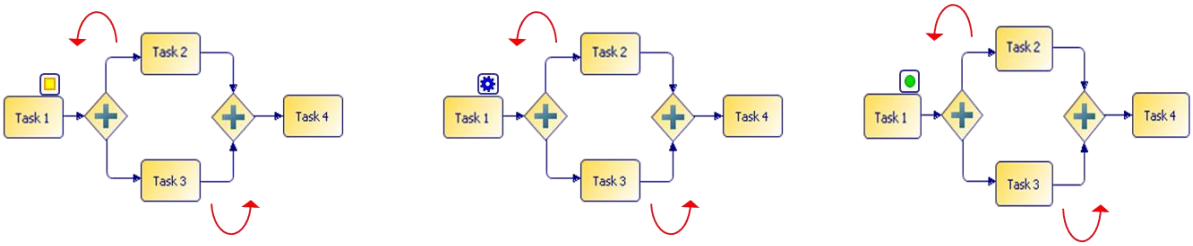

Figure 45 –Admissible scenarios for *Parallel-To-Sequential Shift* and *Conditional-To-Sequential Shift* patterns

Only for *Parallel-To-Sequential Shift* pattern, it is possible to apply the modification also if Task1 is set to “finished” and both Task2 and Task3 are in “ready” state. This scenario is shown in Figure 46.

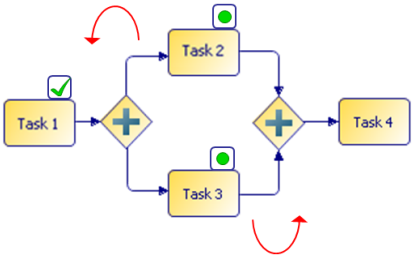

Figure 46 – Admissible scenario for *Parallel-To-Sequential Shift* pattern

As regards *Conditional-To-Sequential Shift* pattern, it is possible to apply the modifications also if Task1 is set to “finished” and one between Task2 and Task3 is in “ready” state (it is impossible that both are in “ready” state because is a conditional flow, only one branch is follow according to a condition evaluation).as we can see in Figure 47

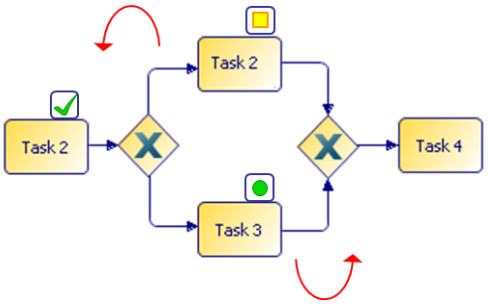

Figure 47 – Admissible scenario for *Conditional-To-Sequential Shift* pattern

This modification is considered not-admissible for *Parallel-To-Sequential Shift* pattern even if only one task to be executed in parallel (in the example Task2 or Task3) is in "executing" or "finished" state. This is shown in Figure 48.

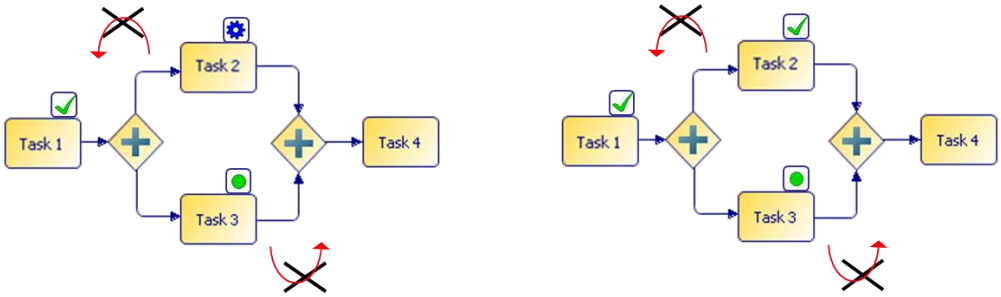

Figure 48 – Not-admissible scenarios for *Parallel-To-Sequential Shift* pattern

For the *Conditional-To-Sequential Shift* pattern the modification is considered not-admissible if the task to be executed in the chose branch (in the example Task2 or Task3) is in "executing" or "finished" state. This situation is shown in Figure 49.

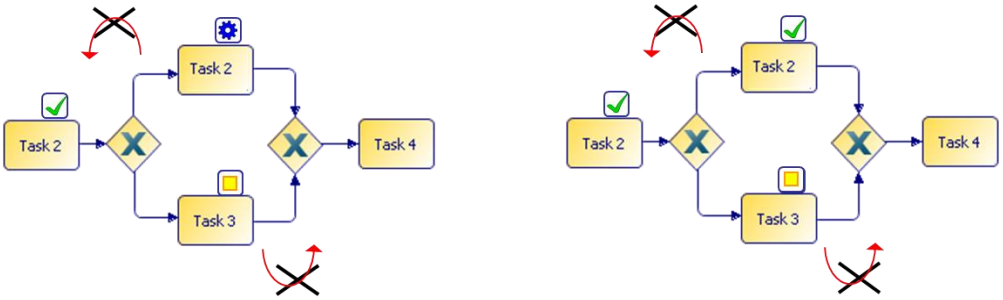

Figure 49 Not-admissible scenarios for *Conditional-To-Sequential Shift* pattern

2.4.4 Swap Process Activities Pattern

Description The *Swap Process Activity* permits to change the predefined ordering of two activities by swapping their position in the same process schema.

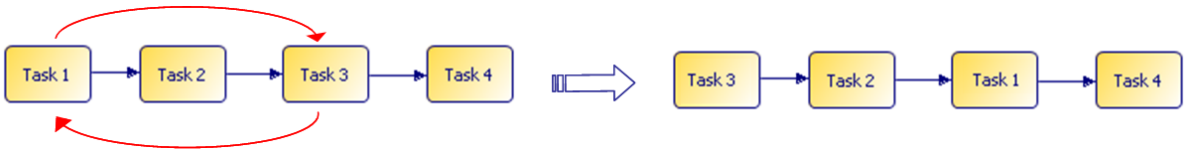

Figure 50 – *Swap Process Activity* pattern

Admissibility conditions

The *Swap Process Activity* is an admissible pattern if, referring to Figure 51, the state of Task1 is “initial” or “ready”.

Figure 51 – Admissible scenarios for *Swap Activities* pattern

In the remaining cases the modification is considered not-admissible. These scenarios are shown in Figure 52.

Figure 52 – Not-admissible scenarios for *Swap Activities* pattern

3 Workflow Instance migration with Nova Bonita

In order to allow a change in the process model that can also be applied to an instance of the process currently running, it is necessary to make some changes to system architecture and to implement some additional features in the workflow engine, because they are not currently available (Nova Bonita currently does not support dynamic reconfiguration, even if this feature is expected shortly [18]).

3.1 Architectural Modifications

At this moment the system architecture is as depicted in Figure 53: the user defines the process model in SBVR Structured English in an editor. A java library (called SBVR4WF that we have previously developed for OPAALS deliverables D2.2 [19] and D2.3 [3]) converts the model from SBVR SE representation to XPDL one. In figure it is called Transformer. Finally, the process model described in XPDL will be passed to the workflow engine (which is accessed through its API) so that it can run the process.

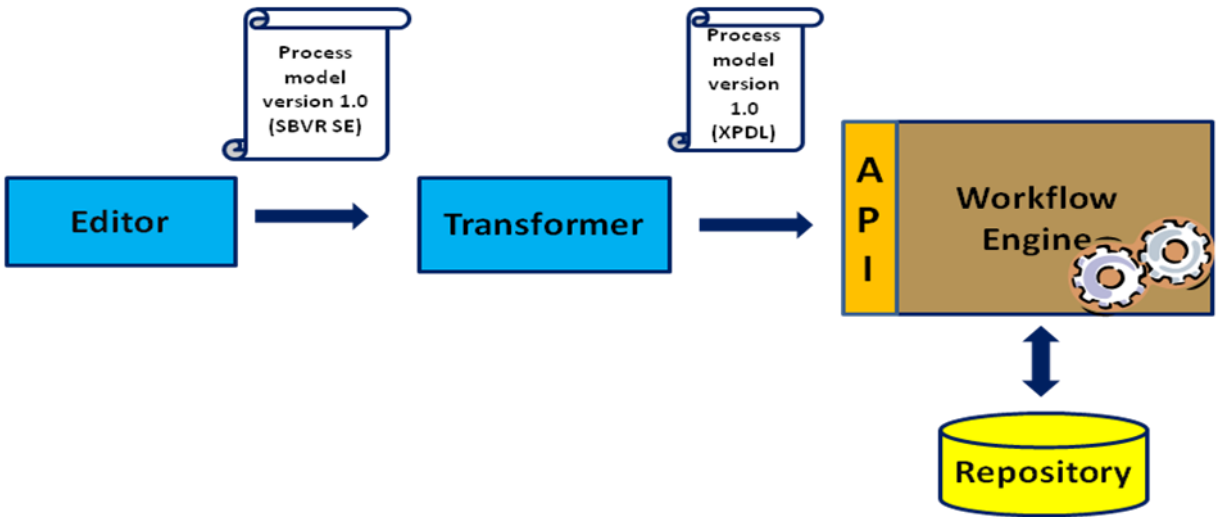

Figure 53 – Current System Architecture

The Figure 54 shows the addition of the component analyzer, which deals with:

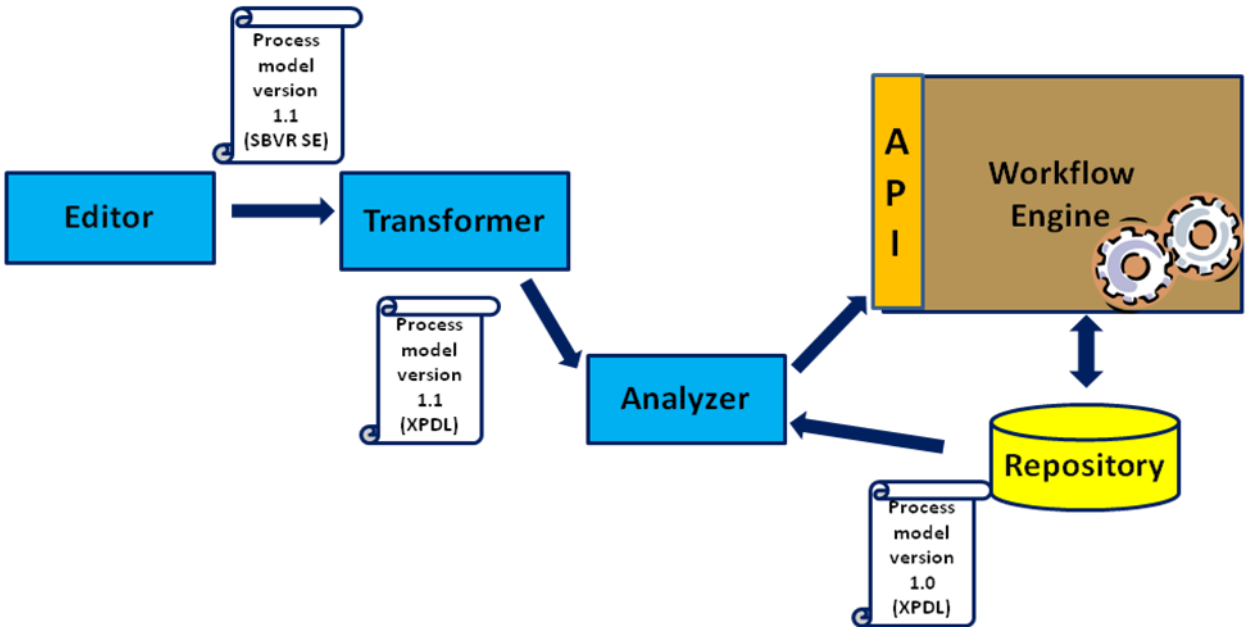

Figure 54 – Architectural modifications

a) checking if the process model executed by the workflow engine is a new version of a model already running on the same engine;

b) checking whether the new model is a new version of an already instantiated workflow model and whose instance is already in progress. If this is the case, it suspends processing of instances for the process currently running on the previous version of the model by calling the API of the workflow engine; otherwise it instantiates a new process on the basis of the new model, as normally expected;

c) identification of the changes between the old model and the new one (to identify the patterns defined in the preceding paragraph);

d) checking if the process model for a certain running instance can be updated on the basis of the pattern established in the preceding paragraph by comparing the runtime data (i.e. see, for all instances, if all the changes are admissible).

Therefore:

- if it is possible to update the model (i.e. all the patterns are admissible), the "new model" will be sent to the workflow engine and the possibility to choose how to handle the instances of the running process will be given to the user;
- if it is not possible to update the model (i.e. even a pattern is not admissible), the "new model" will be sent to the engine and the ability to choose how to handle the instances of the running process will be given to the user.

The management of the instances is done by calling the appropriate function of the workflow engine through its API. (Note: currently there are no features).

3.2 Structure and behaviour of the Workflow Reconfiguration Analyzer

The “Workflow Reconfiguration Analyzer” is the component of the system that controls the admissibility of the reconfiguration.
The library performs many functions.
First of all, the component checks if the process model that it receives as input from the editor is an updated version of a process model already running in the engine or if the model is a different process.
To perform this control the component compares the process name and its version in the XPDL file of the new launched process with the respective values in the XPDL files of the running processes.
The actions to be taken depend on the outcome of this control.

If the analyzed model describes a new process never initialized in the engine (i.e. if the value of the Name of <WorkflowProcess> in the new XPDL file does not match the same value of any process running in engine) then the analyzer ends its task and it passes the XPDL input file to the workflow engine. The engine can upload the new model into its repository. A process based on this new loaded model can be later instantiated and executed.

More complex is the operation if, instead, the new model is "only" a new version of an old model with already one or more running instances. This occurs if the value of the Name of <WorkflowProcess> in the new XPDL file corresponds to the same value of some process running on the engine, but with value <version> (child of <WorkflowProcess> and <RedefinableHeader>) different. In this case the component (by calling the appropriate methods of the Nova Bonita API called RuntimeAPI on the instances of the process involved) suspends all instances of the process running on the old version of process model.
It is possible to identify the cited items into the following two pieces of XPDL code (written according to the XPDL specifications 1.0 [17]).

Listing 1

```
..
<WorkflowProcesses>
   <WorkflowProcess Name="ProcessABC" Id="ProcessID">
     <ProcessHeader />
     <RedefinableHeader>
       <Version>1.1</Version>
     </RedefinableHeader>
</WorkflowProcesses>
…
```

Listing 2

```
..
<WorkflowProcesses>
   <WorkflowProcess Name="ProcessABC" Id="ProcessID">
     <ProcessHeader />
     <RedefinableHeader>
       <Version>1.0</Version>
     </RedefinableHeader>
</WorkflowProcesses>
…
```

When all instances of the process are suspended, the analyzer starts to perform the controls to decide on the admissibility of the reconfiguration.
The new process model is compared with the old model that is now executed and reconfiguration patterns, introduced in section 3.4, are identified.

The information related to the new model are recovered directly from XPDL file, while information on the structure of the running process are called via the methods made

available by the API of the NovaBonita Workflow Engine (in particular the QueryDefinitionAPI) directly from the items allocated to the examined process. Then the runtime information are retrieved from the repository of NovaBonita for each instance of process and finally a check is performed to see, for each instance, if all the pattern of reconfiguration can be considered admissible, as stated in section 3.4.

If, for every instance, all reconfiguration patterns identified on the new process model are considered admissible, then the current running instance on the basis of the new model will continue.
The user can choose how to manage the current instance.

The user can choose one of the following options:

a. to continue the execution on the basis of the old model;
b. to stop running and delete the suspended instance and to restart running the instance on the basis of the new model;
c. to continue the elaboration of the suspended instances under the new model.

Point a) is easy to obtain: it is enough to "restart" the process previously suspended and all its tasks (by calling the methods of the RuntimeAPI). The new process instances will follow, however, the new model.
Point b) is obtained by deleting the entire instance process that is running (via the appropriate methods of the RuntimeAPI of NovaBonita called for that instance, like *deleteProcessInstance (ProcessInstanceUUID)*). In this way all data already processed will be lost and a new instance that is related to the new version of the process model will start, according to the normal operation (via the appropriate methods of the RuntimeAPI, like *instantiateProcess (ProcessDefinitionUUID)*).
Regarding the point c), the work has focused on certain classes of Nova Bonita and the normal behaviour of the engine was modified to find an adhoc solution.
Currently, the workflow engine does not provide functionality to perform this change on the fly preserving the data already compiled at runtime.
In NovaBonita, the engine identifies uniquely (according to the UUID standard and implemented by the java.util.uuid class) any definition of the process model through the assignment of the uuid (by assigning a unique value to ProcessDefinitionUUID).

Moreover, it is assigned a unique identifier to each instance of the process that is running that particular model (ProcessInstanceUUID).In order to continue the execution of a process (instance) on a new model of the process it is necessary to "migrate" the runtime data (previously processed) from an instance to another.
After the admissibility verification, a new process is instantiated on the basis of the revised model and all the objects necessary for the execution and data storage of the process are created, allocated and initialized.
Migration is achieved "pouring" all the runtime information from objects related to the former execution on the new objects. This data flow includes both persistent information stored in the repository by the workflow engine during process execution, and non-persistent information, represented by the values of the volatile variables used during various elaborations.
Finally, old objects and old data will be erased (saving logs of performed operations) and the execution of the new process will be resumed from where it left off the old (new feature added to RuntimeAPI).

Even if a single pattern is not acceptable, it is not possible to update the process model for the current instance.
In this case the user may choose:

a. to continue the execution of the suspended instances on the basis of the old model;
b. to stop running and delete the suspended instance and to restart running the instance on the basis of the new model;

as described before.

4 Conclusions

This document deals with the intricate issue of automatic reconfiguration of a workflow for a long running process.

Some of the main problems encountered during the modification of workflow model that underlies a certain process instance while it is running have been identified.

A theoretical basis and a graphical notation have been introduced, allowing to specify a practical solution to those simple cases of dynamic reconfiguration. These cases were classified by defining patterns of reconfiguration. However, more complex situations could be in future handled leveraging on work done in WP3 of OPAALS, above all in the Deliverable D3.2 [20], about concurrency control in long running transactions. In that document is provided a deeper theoretical analysis, which is based on the definition of transaction vectors, that can help understanding and managing more complicate cases.

For each pattern have been identified the conditions that must be fulfilled in order to carry out the updating of a workflow model through the application of the same patterns, without any data loss or inconsistencies.

The study has produced, in addition, a software component, which is based on the workflow engine (NovaBonita) we used for the previous deliverables of this WP.

This component verifies the admissibility of the reconfiguration of the modifications made to the model, identifying the applied reconfiguration patterns and evaluating the conditions of applicability.

If the update is enabled, the component calls the appropriate methods of workflow engine, which allow the migration of running instance from the old to the new model without loss of data.

The component Reconfiguration Workflow Analyzer can be downloaded from: https://sourceforge.net/projects/wfrecfganalyzer/

A. Appendix - Bonita workflow engine

In the following we provide an overview of the architecture of the Nova Bonita workflow engine solution, the most promising active project of Open Source Java-based Workflow Engine, in particular focusing on to the Process Virtual Machine component and on to the tools to manipulate running instances of workflow models.

A.1. Overview

Bonita[4], which since its 4th version is known with the name of Nova Bonita, is a complete workflow open source solution for handing long-running, user-oriented processes providing out of the box workflow and BPM functionalities to handle business processes. It is a middleware component that provides the features for handling process definition, execution and control. Nova Bonita provides, in fact, a generic process engine, which is called Process Virtual Machine (PVM) and which will be discussed in paragraph A.2, enabling support for multiple process languages (such BPEL, XPDL…), a wide collection of APIs, several tools for defining and controlling processes, a support to XPDL 1.0 standard (activities, Join, Split, Activity Route etc.) and various workflow iterations, and many out of the box integrations in portal and Enterprise Content Management (ECM) solutions. Bonita is downloadable under the LGPL License[5].

The maturity of the project is proved by the numerous success stories around the world.

Figure 55 represents the new architecture of the Nova Bonita system, a complete framework for defining, running and managing workflow models. The entire system is built on top of the component called Process Virtual Machine, which is a generic and extensible engine for running workflow models.

This component defines a common model shared between a wide set of workflow orchestration languages (e.g. BPML, BPEL /Business Process Execution Language/,

[4] http://bonita.objectweb.org

[5] http://forge.objectweb.org/projects/bonita

XPDL, BPELJ, jPDL, etc) and defines a runtime environment for their execution. The execution language adopted by Bonita is XPDL [4]. A support to XPDL 1.0 specification is provided by an ad-hoc module, by several libraries and APIs.

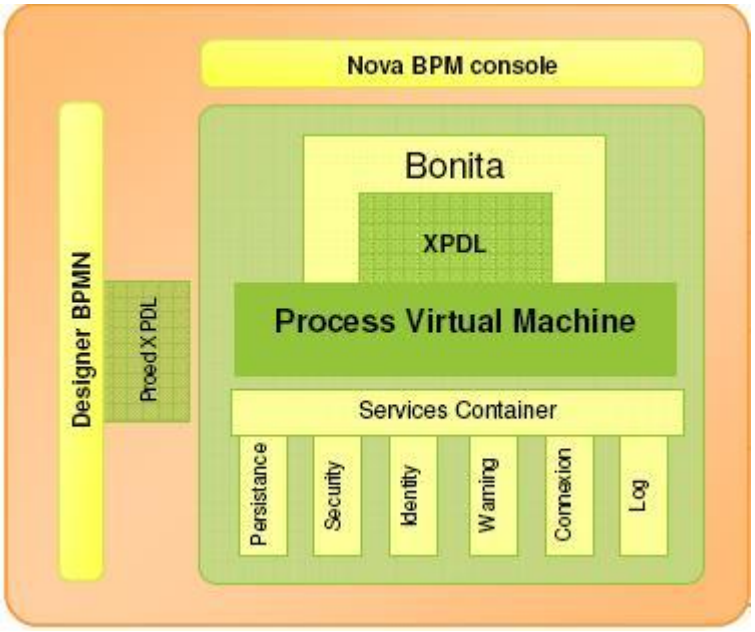

Figure 55 – Nova Bonita Architecture Overview

Nova Bonita provides a graphical editor (ProEd) that allows to design workflow models in BPMN and to obtain automatically an equivalent XPDL representation. It is also possible to import new models directly in XPDL format thanks to specific APIs. This fully integrated BPM Designer improves collaboration between analysts and developers in a graphical and unique development environment. The Nova Bonita Designer can be both integrated in the Eclipse environment and used as a desktop application.

The entire lifecycle of the workflow model, from definition to execution, can be managed and monitored by means of a Web 2.0 BPM Console that is available from the latest release of the workflow engine and covers all BPM deployment, execution and monitoring steps. All these BPM features are released as portlet in an eXo Platform environment[6].

Additional advanced features can be built on top of the PVM by third parties (e.g., BPM vendors, integrators) to allow customization of the solution thanks to Inversion of Control (or IoC) software architecture design. A service container built in the PVM

[6] http://wiki.exoplatform.org/xwiki/bin/view/Portal/

provides the services pluggability that can be injected into the PVM from the outside (in this case from Nova Bonita's APIs). The pluggable technical and business services developed in Nova Bonita are:

- persistence,
- transaction,
- security,
- identity,
- asynchronous messages or timers notifications
- task management,
- Business Intelligence (BI)
- Business Activity Monitoring (BAM).

Thanks to such extensibility, both Standard (Standalone Java based) and Enterprise (JEE Server based) versions of Nova Bonita can be easily configured through the services container.

At the end of this overview, a short introduction to the Nova Bonita APIs is given in order to better explain the capabilities of this software.

The Nova Bonita APIs are divided into 5 different areas[7]:

- DefinitionAPI: to create/modify major process elements into the engine (packages, processes, activities, role mappers, variables by calling java methods instead of importing xpdl files. It will allow also to modify the execution of runtime elements such as tasks and instances.
- QueryDefinitionAPI: to get workflow definition data for packages, processes, activities, role, mappers.
- RuntimeAPI: to manage process, instance and task life cycle operations as well as to set/updates variables.
- QueryRuntimeAPI: to get recorded/runtime information for packages, processes, instances, activities, tasks (support for dynamic queries will be added in the future).
- ManagementAPI: to deploy workflow processes into the engine. XPDL files and advanced entities such hooks, mappers and performer assignments can be deployed individually or in one shot.

[7] http://wiki.bonita.objectweb.org/xwiki/bin/view/Main/ArchitecturalDocs

There is also a generic API that allows executing specific commands needed by the workflow based application:

- CommandAPI: to allow developers to write and execute their own commands packaged within the application.

A.2. Process Virtual Machine

As we said in section 2.1, the Nova Bonita solution is build on top of the Process Virtual Machine[8] (from now on PVM).

The PVM[9] is a revolutionary technology for process based applications that two of the most important open source communities, Red Hat (with Jboss[10]) and OW2 Consortium (with Bull[11]), have delivered together. This is a generic process engine which allows building on top of it embeddable, pluggable and extensible workflow and BPM solutions and that enable the support for multiple process languages (such BPEL, XPDL…). In this way business analyst are free to chose their modeling tools and developer can leverage process technologies embedded in a Java application.

The PVM is a simple Java library for building and executing process graphs that serves as a basis for all kinds of workflow, Business Process Management (BPM) and orchestration process languages. Support for any process language can be build on top of it as plugins.

As Java library, PVM is embeddable: the BPM engine runs as part of an application in any architecture. In fact, the PVM runs in all Java environments, be it:

- a standalone swing application,
- a web application on a servlet container,
- a spring environment,
- an enterprise application on an application server.

For all of these environments, it is possible to set a relational database persistence.

Figure 56 provides an overview of the PVM architecture as used in the Nova Bonita implementation.

[8] http://www.jboss.org/jbossjbpm/pvm/

[9] http://docs.jboss.com/jbpm/pvm/article; http://www.jboss.org/jbossjbpm/pvm_documentation/

[10] http://www.jboss.org

[11] http://www.bull.com

The core element provides the features:

- to define the workflow core elements (like Process, Node, Transition, Event, etc..);
- to execute the process model graph defined by some process language (by using Execution, Variable, Action elements);
- to extend the core infrastructure via external services.

PVM provides a full support to many process languages. Nova Bonita leverages such flexibility focusing on defining workflow models in XPDL, as shown in the top of Figure 56.

Finally a services container is integrated in the PVM. It is used to link external services to the machine, allowing extension of its features.

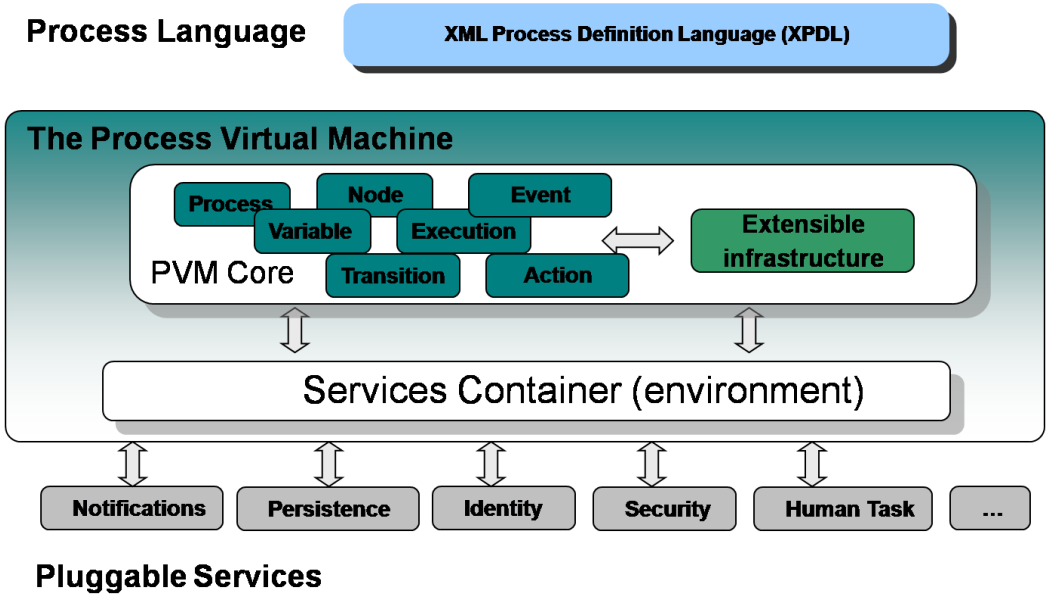

Figure 56 – PVM Architecture

A.3. Instances management

In this section we will see how PVM manages the instances of a given process. Every BPM solution built on top the PVM, thus also Nova Bonita, must take into consideration how the machine works.

Basic Features of PVM

A process model is a graphical description of an execution flow. Each process model can have many execution instances.

The basic structure of a process is made up of nodes and transitions. Transitions have a direction and hence a process forms a directed graph, as represented in Figure 57. Nodes can also have a set of nested nodes.

Figure 57 – Example of process

Each node in the process has a piece of Java code associated, implementing its actual behaviour. The interface to associate Java code with a node is shown in the following piece of code:

```
public interface Executable {
void execute(Execution execution) throws Exception;
}
```

That is the Executable interface used for specifying the behaviour of nodes.

Now, let's look at the runtime data structure. During execution, a pointer, named execution, keeps track of the current position in the process graph. When a new execution of a process is started, the pointer is set on the initial node of the process. Then the execution waits for an external trigger, which is given with the method named *proceed(String transitionName)* of the Execution Interface. In Figure 58 is represented the UML class diagram for the Execution Interface. Thanks to the *proceed* method it is possible to define how to interpret the process graph: in this method, in fact, will be specified the next transition that execution will take and which node will be the next to be executed. Then, the execution will update the node pointer and invoke the node's behaviour.

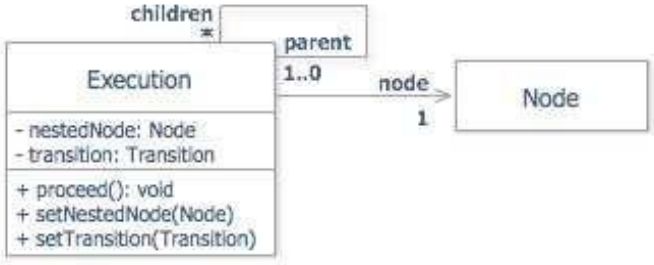

Figure 58 – UML class diagram of the Execution Class

The execution is passed as parameter to the method which define a node's behaviour, named *execute*(). In the *execute*() method, which belongs to the Node Class that implements the Executable Interface, it is defined the programming logic that is executed at runtime. In this way the node's behaviour has access to the current state of the process and the propagation of the execution is subjected to the behaviour.

Working on the *execute()* method, it is possible to specify all possible behaviours, so we can obtain the desired workflow. The execution has only to update the execution node's pointer from a node to the following, so that this pointer can always indicate the current state. In this way, leaving to the interpretation of the process graph only to the behaviour implementation, the behaviour of the engine is pluggable.

There are many examples of **extensions** that can be plugged in to the PVM, like:

- Variables, which contains contextual information related to a single execution.
- Actions, which permits to define action to do in a particular moment during the process execution.
- Concurrent path of execution, which permits to define the behaviour of the engine for the fork, split and join operations.
- Process composition, which permits to manage the situation of a node that refers to another process and must wait the end of the execution of the referenced sub-process before it can proceed with the normal execution.
- Asynchronous continuations, which is the extension of the PVM with an asynchronous message queue. In this way we can manage situations in which synchronous propagation of the execution isn't possible or must be avoided.
- Process updates, which permits to manage the modifications to a particular process like adding or removing nodes and transitions. It is possible to extend the

behaviour of the PVM for two different uses cases: a process inheritance and an instance process customization. The former is the case in which a process differs from another one only for some minor details, so it is possible to specify a sort of process inheritance in which the specialized process inherits all the characteristics from the original process and extends it with a set of updates. The latter case is the modification of a certain execution for a given process to modify the behaviour of a single process instance.

- History, which is the mechanism that provides property to keep track of all the updates done during a process execution. This is obtained generating history logs while the process is being executed. The generated log objects are Java objects subsequently passed to a logging service. It is possible to use the log information to generate, for example, compensation transactions.
- Persistence, which provides the mechanism of storage information. Thanks to the adopted ORM (Object Relational Mapping) technology, such as Hibernate, it is possible in the PVM to refer to Java persistence objects, without any reference to database records. The object relational-mapper in a subsequent moment will associate the java objects with the correspondent database records. So persistence of the process and its execution can be provided by separate services. For this reason, process languages that don't need persistence, like e.g. pageflow, simply don't use a database to store data.

  For the PVM, if a database is required, the data in a workflow database are split in three areas, as represented in Figure 59:

  1. Static Process Definition Information, which contains the complete process structure;
  2. Runtime Execution Information, which are typically updated in every transaction that includes workflow operation so that the execution's node pointer can be moved to the subsequent node;
  3. History Logs.

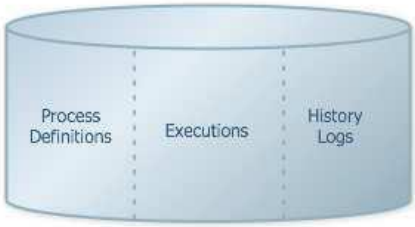

Figure 59 – Subdivision of a workflow database

- Timers, indispensable for operations that are scheduled to be executed in the future.
- Service, which is an essential aspect as regards the embeddabity of the engine inside of the client application. The availability of external services is provided to execution and node behaviour implementations by means of interfaces. The simpler way is to add a context property to the execution. Such context can manage a set of named services and objects. The client knows the environment it is running in, so it constructs the context object and injects it in the execution before a method is invoked on it. If an Inversion of Control (IoC) container is used for the context, then the services can be created on demand during execution of a process. In this case, the particular implementation for the services that we have to use is specified in a XML configuration file.

Bibliography

[1] Raimund Eder, Thomas Kurz, Thomas J. Heistracher, Mario Russo, Antonella Filieri, Prabhakar TV, Hagen Peukert, Alexandros Marinos, Stan Hendryx. **D2.2 - *Automatic code structure and workflow generation from natural language models***. OPAALS Project, March 2008.

[2] Raimund Eder, Thomas Kurz, Thomas J. Heistracher, Victor Bayon, Mario Russo, and Antonella Filieri. **D2.1 - *Design of Software Generation Prototype***. OPAALS Project, October 2007.

[3] Antonella Filieri, Antonio Margarito, Raimund Eder, Thomas Kurz. **D2.3 – *Extended vocabulary and rule set for an existing scenario*.** OPAALS Project, November 2008.

[4] WfMC. **XML Process Definition Language (XPDL 2.1)**, March 2008. http://www.wfmc.org/xpdl.html.

[5] Stefanie Rinderle, Manfred Reichert, Peter Dadam. ***Correctness criteria for dynamic changes in workflow systems - a survey***. Data and Knowledge Engineering 50, pages. 9-34, 2004.

[6] H.Schonenberg, R.Mans, N.Russell, N.Mulyar, and W.M.P. van der Aalst. ***Process Flexibility: A Survey of Contemporary Approaches***. In J.Dietz, A.Albani, and J.Barjis, editors, *Advances in Enterprise Engineering I*, volume 10 of *Lecture Notes in Business Information Processing*, pages 16-30. Springer-Verlag, Berlin, 2008.

[7] Manfred Reichert, Peter Dadam. ***ADEPT$_{flex}$ – Supporting Dynamic Changes of Workflows Without Loosing Control***. JIIS 10 (2),pages 93-129. 1998.

[8] M.Pesic, M.H.Schonenberg, N.Sidorova, and W.M.P. van der Aalst. ***Constraint-Based Workflow Models: Change Made Easy***. In F.Curbera, F.Leymann, and M.Weske, editors, *Proceedings of the OTM Conference on Cooperative information Systems (CoopIS 2007)*, volume 4803 of *Lecture Notes in Computer Science*, pages 77-94. Springer-Verlag, Berlin, 2007.

[9] W.M.P van der Aalst and S.Jablonski. ***Dealing with workflow change: identification of issues and solutions***. *International Journal of Computer Systems, Science, and Engineering*, 15(5),pages 267-276. 2000.

[10] B.Weber, S.B.Rinderle, and M.U.Reichert. ***Change support in process-aware information systems - a pattern-based analysis***. Technical Report Technical Report TR-CTIT-07-76, ISSN 1381-3625, Centre for Telematics and Information Technology, University of Twente, Enschede, 2007.

[11] W.M.P. van der Aalst and T.Basten. ***Inheritance of Workflows: An approach to tackling problems related to change.*** Computing Science Reports 99/06, Eindhoven University of Technology, Eindhoven, 1999.

[12] Stefanie Rinderle, Ulrich Kreher, Markus Lauer, Peter Dadam, ***On Representing Instance Changes in Adaptive Process Management Systems***. Enabling Technologies: Infrastructure for Collaborative Enterprises, 2006. WETICE '06. 15th IEEE International Workshops on.

[13] F.Casati, S.Ceri, B.Pernici, G.Pozzi. ***Workflow evolution***. Data and Knowledge Engineering 24 (3) pages 211-238. 1998.

[14] W.M.P. van der Aalst. ***Exterminating the Dynamic Change Bug: A Concrete Approach to Support Workflow Change***. BETA Working Paper Series, WP 51, Eindhoven University of Technology, Eindhoven, 2000.

[15] Mathias Weske. ***Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System***. Proceedings of the 34th Hawaii International Conference on System Sciences. 2001.

[16] J.Cardoso, Amit Sheth, ***Adaptation And Workflow Management Systems***. International Conference WWW/Internet 2005, Lisbon, Portugal, 19-22 October 2005. pages 356-364. 2005

[17] WfMC. XML Process Definition Language (XPDL 2.1), March 2008. http://www.wfmc.org/xpdl.html.

[18] Nova Bonita Workflow Engine Roadmap. http://wiki.bonita.objectweb.org/xwiki/bin/view/Main/Roadmap

[19] Raimund Eder, Thomas Kurz, Thomas J. Heistracher, Mario Russo, Antonella Filieri, Prabhakar TV, Hagen Peukert, Alexandros Marinos, Stan Hendryx - ***D2.2 - Automatic code structure and workflow generation from natural language models.*** OPAALS Project, March 2008.

[20] Amir Razavi, Sotiris Moschoyiannis, Paul Krause- ***Deliverable D3.2: Report on formal analysis of autopoietic P2P network, together with predictions of performance.*** OPAALS Project, June 2007.