Recovering the Rationale of Change from Process Model Revision Histories: a taxonomy of process model change patterns, detection algorithms and their implementation

Research Report

Selim Erol

Created: Feb 14, 2015, Last update: July 3, 2015

1 Introduction

Large to medium sized organizations document their business process architecture in some way. On the one hand informal process descriptions are used for the goal of documentation, analysis, knowledge transfer and governance. On the other hand business processes are described through (semi-)formal languages (e.g. EPC [Scheer, 1998], BPMN[OMG, 2011]) that potentially allow for their automation and for the orchestration of related software services and applications [Giaglis, 2001; Davies et al., 2006]. The latter kind of artifacts, commonly referred to as process models, are created, stored and maintained by means of software-based modeling environments (e.g. ARIS Business Architect[1]). These modeling environments typically consist at least of a model editor and a model repository. The repository is used to store larger collections of process models, enable version management and efficient model retrieval. Such process model repositories often contain hundreds to thousands of process model artifacts and – in case some kind of versioning exists – also a multiple of model versions.

In practice process models are developed in an iterative manner. From vague ideas of the process in focus to informal process descriptions and finally deployable workflow specifications and so forth. Process models are furthermore continuously adapted to changing business requirements. Keeping track of model changes is typically accomplished by an change log or a revision history. In the first case change operations are logged by storing at least the order of an operation, the type and the target object. Thus, a sequence of change operations is obtained that exactly shows the actions a modeler has undertaken to change a model. In the case of the revision history approach snapshots at discrete points in time are taken and stored along with a time-stamp. Model changes are then reproducible by comparing two subsequent revisions revealing only the substantial changes but hiding all intermediate changes that led to the final change. Both approaches have their advantages and disadvantages which have been broadly discussed in literature, e.g. in [Mens, 2002]. In practice, the revision history approach has gained ground in both model and software development tools.

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

Versioning systems like SVN[2] are exemplary implementations of the revision history approach. One of the drawbacks of the revision history approach is that the rationale of changes to a process model is not easily recoverable if not documented explicitly or implicitly through a change log. By rationale the reason for a change is meant which subsequently is transformed into an operational change of a process model artifact. The above mentioned problem is due to the fact that compound changes reflecting a change rationale need to be "guessed" a posteriori from a set of unrelated atomic change operations which in turn have been derived from a simple revision comparison.

To tackle the above mentioned challenges an approach has been chosen that builds upon the concept of change patterns formulated in Weber et al. [2007] and extends this concept in various ways. First, by conducting an explorative study of changes in a highly dynamic process modeling environment – a cloud-based modeling environment – and subsequent classification of compound changes through change patterns. Second, by introducing a pattern language allowing the description of typical compound changes (change patterns) in an easy, practicable but still comprehensible way. Second, by formulating algorithms for recovering compound changes from revision comparisons. Finally, through an prototypical implementation in a respective modeling environment and it's subsequent evaluation.

In the subsequent sections of this paper the approach for describing change patterns through a pattern language and related detection algorithms are described in detail. In section 2 basic concepts of revision history, revision comparison and compound changes are discussed. In section 3 the pattern based approach to specifying compound changes is presented. Section 4 is dedicated to the formulation of respective detection algorithms. In the last section a prototypical implementation is presented along with it's evaluation. Results are summarized and discussed in the last section.

2 Basic concepts

2.1 Process model

For the following considerations a simple meta-model for process models has been used. Accordingly a process model M is a finite set of model elements $\dot{m} \in M$ where each model element is classified into three distinct types of model elements a..activity, g..gateway, e..sequence flow edge. Instances of such classes form subsets of M: the set of all activities $A = \{a_0..a_n\}$, the set of all gateways $G = \{g_0..a_m\}$, the set of all edges $E = \{e_0..e_k\}$ where $n, m, k \in \mathbb{N}$. Thus the set of model elements M can be described as a three tuple (A, G, E). Where A is a finite set of activities, G is a finite set of logical gateways and E is a finite set of directed sequence flow edges. E represents different sequences of A and G. To be precise it can specified as a subset $E \subseteq (A \times G) \cup (G \times A) \cup (A \times A) \cup (G \times G)$. Different types of gateways $\{\times, +, \circ\}$ exist where type $T(G) = \times$ refers to an exclusive gateway, type $T(G) = +$ refers to a parallel gateway and type $T(G) = \circ$ refers to an inclusive gateway. Gateways are also classified according to their role in the process model. We distinguish the set of split gateways G_s which includes all gateways g that have at least two outgoing sequence flow edges and have exactly one ingoing sequence flow edge, formally $G_s = \{g \in G \mid OUT(g) \geq 2 \wedge IN(g) = 1\}$ where OUT is a function that counts all outgoing and IN is a function that counts all ingoing edges for a given gateway node and a set of join gateways G_j that is defined as $G_j = \{g \in G \mid IN(g) \geq 2 \wedge OUT(g) = 1\}$. Figure 1 shows a UML[3] diagram of the types of model elements incorporated for subsequent considerations. Note that we use a rather general and

[2]https://subversion.apache.org
[3]http://www.uml.org

relaxed meta-model and formalization. This is due to the requirement that we want to cover as well process models in progress that are not complete in the sense of executability. We also did not include events in the meta-model as we wanted to provide a general meta-model that abstracts from concrete modeling languages and keeps the semantics simple to have more explanatory power. The formal description of the meta-model provided above builds upon the work of van der Aalst [1999].

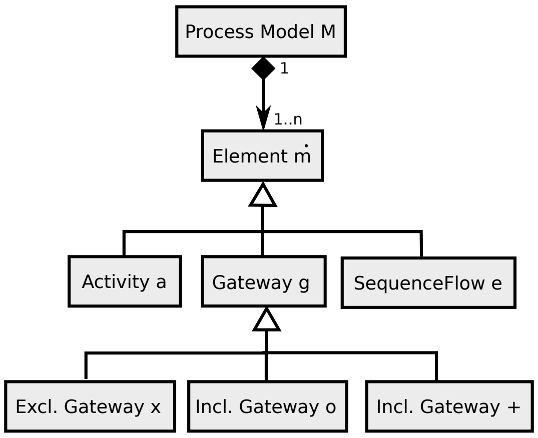


Figure 1: Meta-model of process models as used in subsequent considerations

2.2 Process model change

Process models are created through a model editing software and are maintained in process model repositories [Weber et al., 2011; La Rosa et al., 2011; Dijkman et al., 2012]. Changes to process models are performed through their graphical representations. Such graphical representations usually consist of shapes and edges of different types to represent the semantics of process model elements. E.g a rectangular shape with rounded corners to represent a task. When an agent changes a process model then it performs a set of subsequent atomic change operations until a desired new state of the model is reached. These atomic change operations are the result of some change rationale – an agents intent to change a process for some reason. The rationale of change therefore can be related to a set of compound change operations that has been applied subsequently and follows some order. This set of change operations can be defined as as a tuple $S := (O, A)$ consisting of an unordered set of atomic change operations $O := \{o_0, o_1, ..., o_n\}$ and a set of relations $A := \{a_0, a_1, ..., a_m\}$ that determine the order change operations have been executed. Each atomic change operation $o := o_i(\dot{m}, type, args)$ is defined through it's target model element $\dot{m}$, the $type$ of operation ("add", "delete", "modify") and optional arguments $args$.

2.3 Process model revisions and revision comparison

A revision history of a process model represents different states of a process model over time. These different states of a process model result from interactions of some kind of agent with the process model artifact leading to respective changes. The scenario depicted in figure 2 shows two model interactions that take place in sequence. The first modeler u_A opens (checks out) a model revision r_i from a repository at point in time $t_{A,s}$, changes the model, and submits (checks in) the changes to the repository at point in time $t_{A,e}$ which leads to model revision r_{i+1}. Subsequently, a second modeler u_B accesses the model revision r_{i+1}, opens it at a point in time $t_{B,s}$, changes it and submits as well his new model revision at point in time $t_{B,e}$ which leads to a revision r_{i+2}. This scenario actually reflects a special case of users interacting with a process model. In practice much more complicated scenarios with multiple agents and interfering changes are possible (see for example in Erol [2012]; Erol and Neumann [2013]).

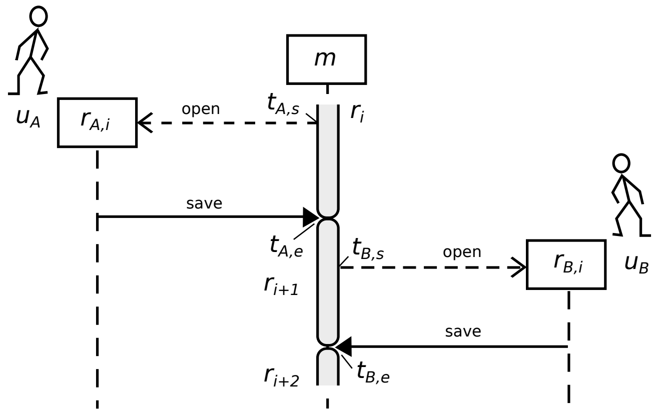


Figure 2: Multiple agents subsequently changing a process model which leads to revisions r_i, r_{i+1}, r_{i+2}

However, any change of a process model and it's subsequent submit to a model repository leads to a revision history $R = \{r_{t_0}, r_{t_1}, r_{t_2}, .., r_{t_n}\}$ where the number of revisions is $n+1$, the earliest revision is r_{t_0} and the latest revision is r_{t_n}. r_{t_n} reflects the current state of the process model. Each revision has an index t_i which reflects the time-stamp of the revision. For each subsequent pair of timestamps a temporal order $t_i > t_{i+1}$ is defined. This leads to the conclusion that revision $r_{t_{i+1}}$ is the direct successor of r_{t_i}. For the purpose of simplicity time-stamp notation is replaced by indices only such that $R = \{r_0, r_1, r_2, .., r_n\}$ without losing any relevant information for subsequent considerations. Each revision r_i of a process model consists of a set of model elements $r_i = \{m_0, m_1, .., m_k\}$ where each model element m_j is assumed to have a unique identifier j.

Comparing two subsequent revisions r_i, r_{i+1} requires that unchanged parts of the original revision are distinguished from those parts that have been changed. Thus a diff is obtained that can be used to derive change operations that have been applied. Distinction of changed from unchanged parts of a model can be accomplished by a pairwise comparison of model elements, their unique identifiers, their properties and references to other model elements. The concept of unique identifiers allows for a convenient distinction between identical model elements that have remained unchanged, identical model elements that have been changed in the newer revision, deleted model elements and added elements. As a result a revision r_i can be represented as $r_i := r_{i,identical \wedge equal} \cup r_{i,added} \cup r_{i,deleted} \cap r_{i,identical \wedge modified}$ where $r_{i,added}$, $r_{i,deleted}$, $r_{i,identical \wedge modified}$, $r_{i,identical \wedge equal}$ represent subsets of model elements that are classified according to the above distinction. A revision difference (short diff) d then is represented through $d := r_{i,added} \cup r_{i,deleted} \cup r_{i,modified}$.

Having compared and classified changed model parts, it is possible to infer a – yet unordered – set of atomic change operations that has been applied to an antecedent revision r_i. This inference is accomplished by relating each elementary difference to an atomic change operation, e.g. a newly added model element in r_{i+1} to a change operation of type "add", non-existent element in r_{i+1} to a change operation of type "delete" and so on. The set of change operations applied to r_i can be defined as as a tuple $S_i := (O, A)$ consisting of an unordered set of atomic change operations $O_i := \{o_{i,0}, o_{i,1}, ..., o_{i,n}\}$ and a set of relations A_i that is empty as we can not determine the order relations from a revision comparison. A revision may comprise subsets of compound changes $sc_j \subset O_i$ that refer to some rationale or may refer as a whole to some rationale $sc_j = O_i$.

3 Discovering change patterns from revision histories

The term "change pattern" in the context of process model change was probably first introduced by Weber et al. [2007]. The authors suggest a taxonomy of change patterns but do not base their findings on a detailed and systematic empirical study of process model changes. Therefore we extend their notion of change patterns with regard to the work of Alexander [1964]; Gamma et al. [1995] were patterns are understood as frequently recurring types of design choices or best-practices in design practice. Accordingly change patterns are descriptions of changes that abstract from actual applications. They reflect a principal structure of a change rather than its precise implementation.

For the purpose of identifying and classifying change patterns and their respective semantics an explorative study of revision histories of several process models was conducted. The revision histories investigated are the result of a large case-study of collaborative process modeling performed between years 2011 and 2012 Erol [2012]. The case-study resulted in 415 process models and 3859 process model revisions. For exploring change patterns we selected a subset of process models from the case-study. Namely we excluded process models with at most one model element[4] and less than two revisions which led to a final set of 204 process models[5].

The explorative study of process model revisions was performed in a systematic way. By systematic we mean that for each process model we compared succeeding model revisions, identified, named and described model changes and at the same time classified them. Each new model change identified was checked against already identified change patterns and in case it could not be associated to existing patterns it was used to build a new class or sub-class. Thus we arrived at a taxonomy of change patterns.

The identification of model changes from succeeding model revisions was performed in a semi-automated way. Although the semantics of a model change and it's classification was performed visually by a human agent it was strongly supported by computational means. To be concrete we computed differences between succeeding revisions through detecting newly added and deleted model elements as well as highlighting them in the respective process visualization (a process diagram). We used three colors: green for newly added model elements, red for deleted model elements (model elements that did not exist anymore in the succeeding revision) and yellow for model elements that have been added and deleted within the same model revision. To facilitate visual comparison and detection of model changes revisions were scaled to a size that allowed to capture the whole process model at one sight but detailed enough to recognize small scale changes as well, e.g. change of a single element. All revisions for a model were listed as metaphoric tiles placed nearby in an open ended and scrollable list that allowed for back and forth comparison (see figure 3).

Through pairwise comparison of model revisions we finally arrived at a taxonomy that includes several high-level classes of changes (extension, reduction, parallelization, etc.) and several layers of subclasses (see figure 4) which represent specializations of their respective super class. Dark highligted classes are reverse patterns that transform a given state of a process model into its original state before the application of the change. We symbolized this relation with dashed edges in figure 4. For the procedure of pattern identification and classification we used a simple paper and pencil method. Each presumed candidate for a pattern was sketched graphically on a dedicated paper card

[4]these models are the result of a peculiarity of the software environment
[5]process model data is publicly available on http://www.erol.at/pwiki/xowiki_list_pages.php

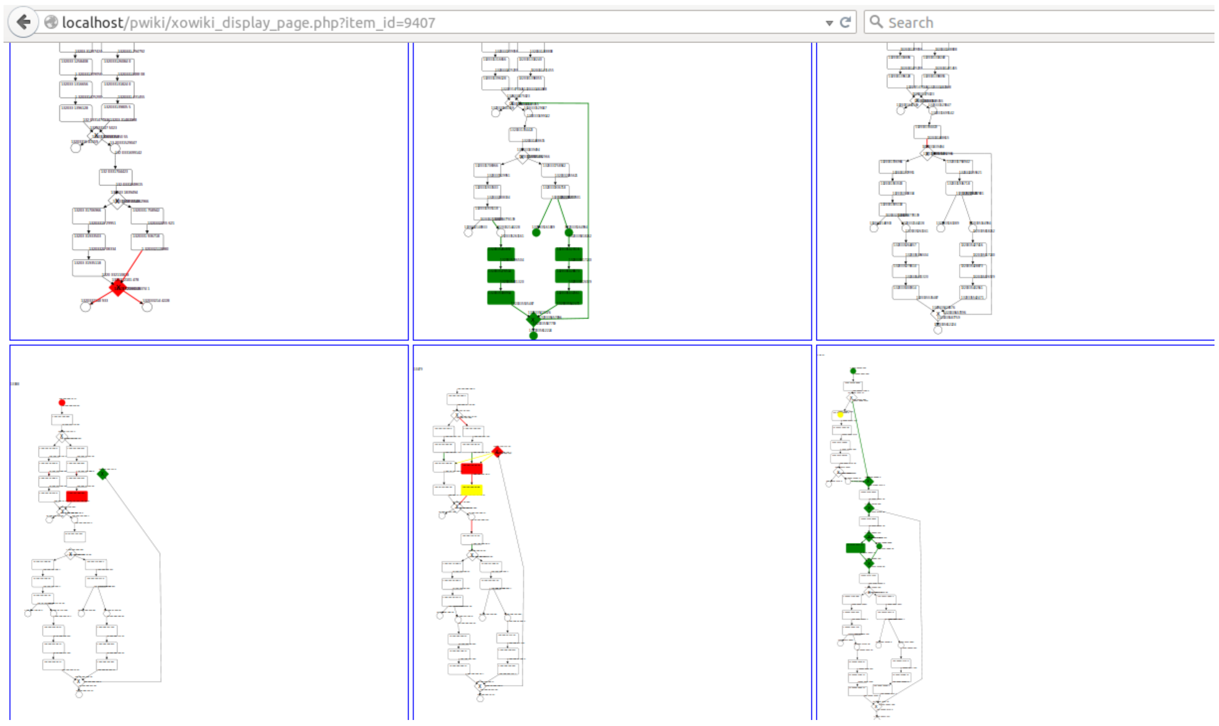


Figure 3: Screenshot of a process model revision history and its visualization as a list of tiles to facilitate identification and classification of changes. The process models represent a book borrowing process at the university library.

(see figure 24) and given a preliminary pattern name and if feasible assigned to an existing super class. Problems during classification occurred mainly due to three reasons:

- Model revisions are only timely snapshots of process models. They are the result of a set of subsequent changes rather than reflecting an isolated semantically well delineated change. e.g. we experienced that the first revision of a process model already covers the largest part of the final process model. All extensions and modifications that led to this model state are not recoverable (see figure 5).

- Model revisions are not necessarily created after a change has been completed from a rationale point of view. E.g. a modeler chooses to interrupt his change activity and save an intermediate state of the model as he decided to go for a coffee break. Afterwards he continues and completes the intended change – the change rationale is scattered over two or more revisions (see figure 6 (a)).

- Model revisions not necessarily include only one particular change pattern but multiple intertwined patterns that makes them difficult to distinguish, e.g. a modeler extends the flow of activities by inserting several activities and at the same time branched the same sequence of activities by inserting a gateway (see figure 6 (b)).

Nevertheless, the large number of process models and attached revisions made it possible to extract a large number of unambiguous patterns that formed the basis of our pattern taxonomy. The possibility of tracking changes over more than one revision helped to reconstruct distributed change

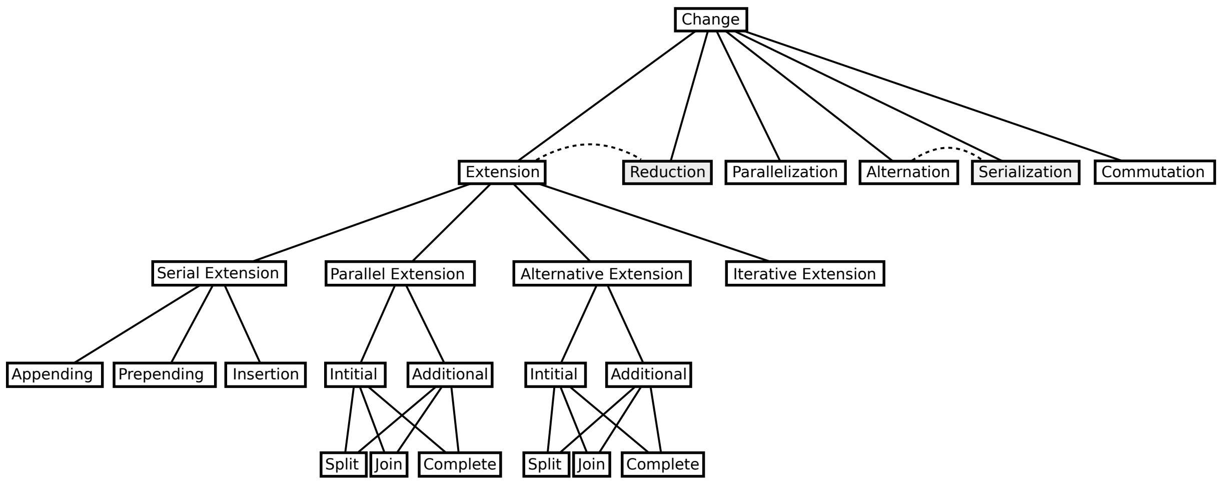


Figure 4: Change pattern taxonomy. Nodes represent classes of changes (change patterns) and edges represent "is-a" relations ships read from bottom to top.

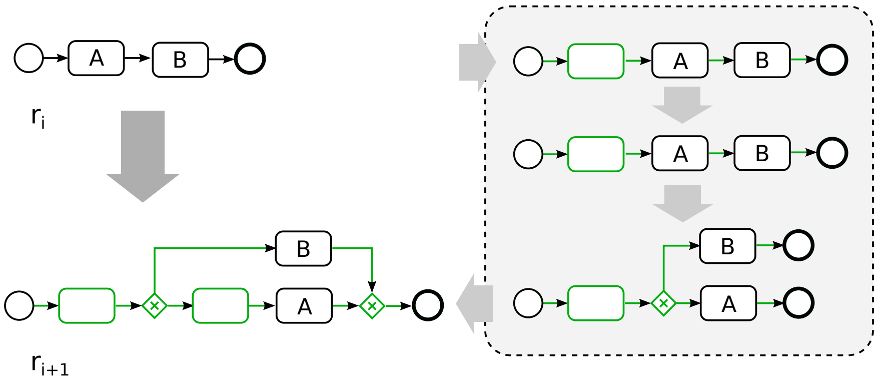


Figure 5: Visible and hidden revisions.

patterns. The problem of intertwined change patterns proved to be not easily solvable. However, for classification purposes such change patterns were treated as separate patterns if recognizable and else disregarded.

4 Describing change patterns through a pattern language

Classifying change patterns required a dedicated language that abstracts from individual pattern instances. For this purpose we developed a pattern language in the sense of the well-known design pattern concept [Gamma et al., 1995] together with a set of graphical symbols to illustrate change patterns. The set of graphical symbols along with their semantics is depicted in figure 7. The graphical symbols where designed with the need to describe change operations regardless their concrete nature. This means that we aimed at providing symbols to describe model parts such as sequences of activities, parallel or alternative branches of activity sequences without having to specify the concrete number of activities or branches included. We also added gateway symbols to describe the logical flow of activities according to the meta-model mentioned above. The little $\oplus$ and $\ominus$ symbols are used to denote change operations of type "add" or "delete". The large arrow in gray is used to

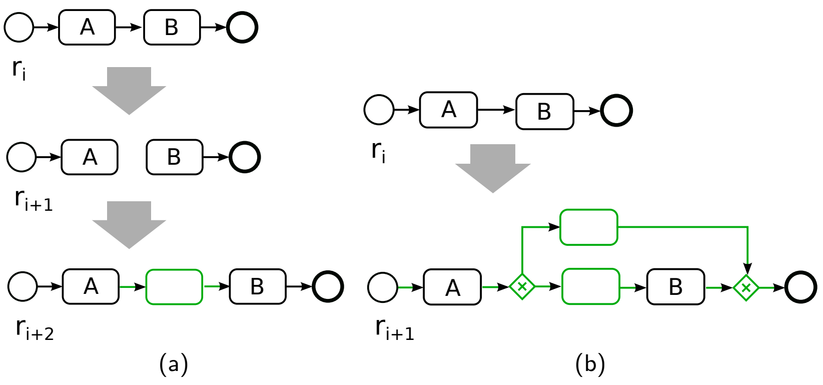


Figure 6: (a) Changes scattered across multiple revisions. (b) Intertwined changes.

denote the direction of a change – the order of revisions $r_i \rightarrow r_{i+1}$.

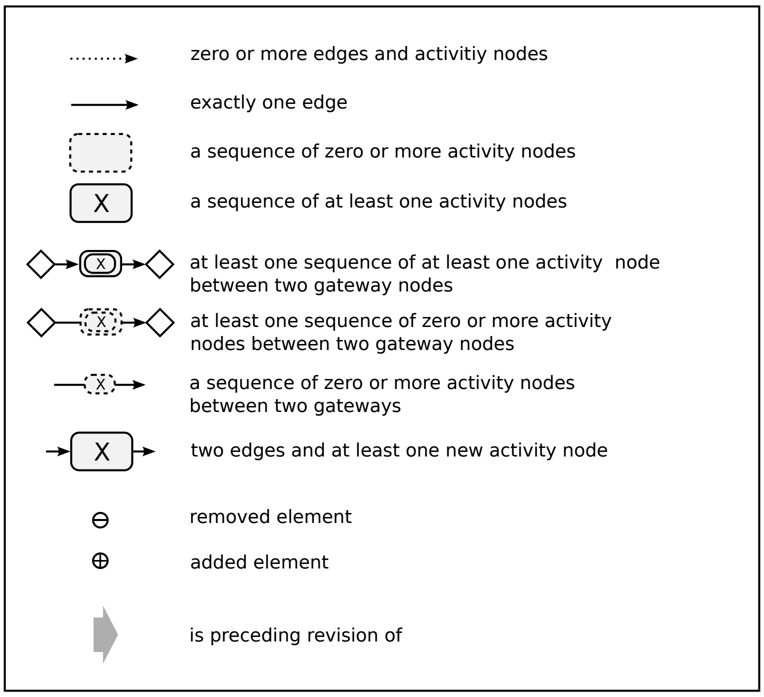

zero or more edges and activitiy nodes
exactly one edge
a sequence of zero or more activity nodes
X
a sequence of at least one activity nodes
X
at least one sequence of at least one activity node between two gateway nodes
X
at least one sequence of zero or more activity nodes between two gateway nodes
X
a sequence of zero or more activity nodes between two gateways
X
two edges and at least one new activity node
⊖
removed element
⊕
added element
is preceding revision of

Figure 7: Set of graphical symbols to illustrate change patterns

In the following subsections each change pattern is described in detail. For this purpose we use a template that contains a fixed set of properties, thus providing a practical way to consistently describe and collect such patterns. We use a pattern `id` to identify a pattern (e.g. `Ext.Ser.Ins`), a descriptive `name` and a more explanatory `description`. We also provide an `illustration` of the change pattern where we use the graphical language presented above. The `signature` property is used to provide a precise search criterion for detecting a respective change pattern given that we have the subset of change operations or affected model elements at hand. The algorithmic implementation is discussed in a later chapter. Finally, we provide a property to capture the `rationale` of the change pattern. The latter is important as we aim to support a modeler in understanding and interpreting model revisions. As an `example` we use a standard online shopping process.

4.1 Pattern `Ext.Ser.R`: Serial extension to right, appending

Description Additional activity elements are appended to an existing sequence of activities.

Illustration see figure 8.

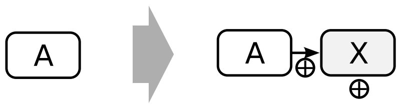


Figure 8: Change pattern: Serial extension to right

Signature At least the following atomic change operations are applied to a terminal activity: `add_sequenceflow(A,X)`, `add_activity('X')`.

Rationale This pattern is usually applied when a process turns out to be not complete in the sense that additional activities have to be performed or additional events are expected to occur at the end of the process to achieve a predefined goal, result or output.

Example For the payment process at Amazon one needs to additionally enter the three digit security code to complete the order.

Category Extension patterns

Related change patterns

`Red.Ser.R` is reverse of `Ext.Ser.R`.

4.2 Pattern `Ext.Ser.L`: Serial extension to left, prepending

Description Additional activity elements are prepended to an existing sequence of activities.

Illustration see figure 9.

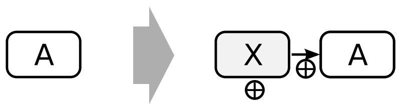


Figure 9: Change pattern: Serial extension to left

Signature At least the following atomic change operations are applied to an initial activity: `add_activity(X)`, `add_sequenceflow(X,A)`.

Rationale This pattern is usually applied when a process turns to be not complete in the sense that additional activities have to be performed or additional events are expected to occur at the beginning of the process to achieve a predefined goal, result or output.

Example To read the content of a web page one needs to select the language or location before.

Category Extension patterns

Related change patterns `Red.Ser.L` is reverse.

4.3 Pattern Ext.Ser.Ins: Serial insertion

Description Additional activity elements are placed between two succeeding activity elements.

Illustration see figure 10.

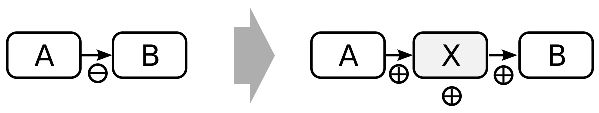


Figure 10: Change pattern: Serial insertion

Signature At least the following atomic change operations are applied: `delete_sequenceflow(A,B)`, `add_activity(X)`, `add_sequenceflow(A,X)`, `add_sequenceflow(X,B)`.

Rationale This pattern is usually applied when a process needs to be refined to achieve a predefined goal, result or output. This means that additional activities need to be performed or additional events are expected to occur in the course of a process.

Example A sign in activity is needed in the course of an online ordering process. After browsing the catalog and adding products to a cart a sign-in activity is required to proceed with check-out activity.

Category Extension patterns

Related change patterns `Red.Ser.Ins` is reverse.

4.4 Pattern `Ext.Par.Ini.Comp`: Complete Initial Parallel Extension

Description An additional activity sequence is added in parallel to an other activity element as part of a sequence of activities. Therefore two sequence flow edges are deleted, two parallel gateways are inserted, an additional activity element is inserted.

Illustration see figure 11.

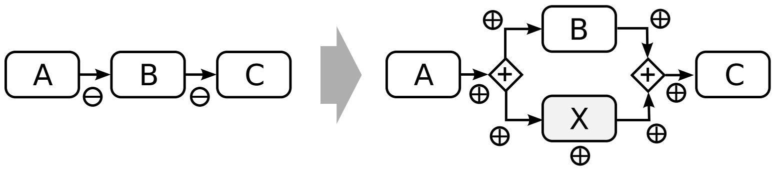


Figure 11: Change pattern: Complete Initial Parallel Extension

Signature ..
`delete_sequenceflow(A,B)`, `delete_sequenceflow(B,C)`, `add_activity(X)`, `add_gateway(G1)`, `add_gateway(G2)`, `add_sequenceflow(A,G1)`, `add_sequenceflow(G1,B)`, `add_sequenceflow(B,G2)`, `add_sequenceflow(G2,C)`, `add_sequenceflow(G1,X)`, `add_sequenceflow(X,G2)`.

Rationale This pattern is usually applied when a process needs additional activities to achieve a predefined goal, result or output. Additional activities are added in parallel to other activities which means that they do not require each other as a prerequisite. The process converges after parallel activity sequences are completed.

Example For an order fulfillment process at Amazon the activity of picking and packing is extended with a parallel activity of planning for an optimal route for shipping.

Category Extension patterns

Related change patterns `Red.Par.Fin.Comp` is reverse.

4.5 Pattern `Ext.Par.Add.Comp`: Complete Additional Parallel Extension

Description An additional activity element is placed in parallel to an already existing set of parallel activity branches. Additional sequence flows and an additional activity element are inserted.

Illustration see figure 12.

Signature ..
`add_activity(X)`, `add_sequenceflow(G1,X)`, `add_sequenceflow(X,G2)`.

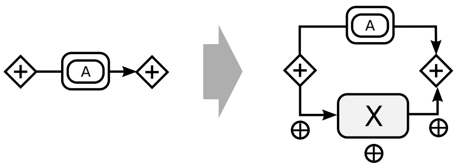


Figure 12: Change pattern: Complete Additional Parallel Extension

Rationale This pattern is usually applied when a process needs additional activities to achieve a predefined goal, result or output. Additional activities are needed to be performed in parallel to other activities. The process converges after all parallel activity sequences are completed.

Example In addition to picking, packing and finding an optimal route for shipping as well the invoice creation and printing activity is added to be performed in parallel.

Category Extension patterns

Related change patterns `Red.Par.Comp` is reverse.

4.6 Pattern `Ext.Alt.Ini.Comp`: Complete Initial Alternative Extension

Description A sequence of activities (or a single sequence flow) is extended with alternative sequences of activities (or a single sequence flow). Therefore alternative gateways are inserted along with additional activities.

Illustration see figure 13.

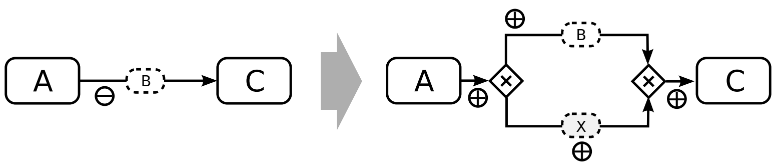


Figure 13: Change pattern: Complete initial alternative extension

Signature ..
`delete_sequenceflow(A,C)`, `add_activity(X)`, `add_gateway(G1)`, `add_gateway(G2)`, `add_sequenceflow(A,G`
`add_sequenceflow(G1,B)`, `add_sequenceflow(B,G2)`, `add_sequenceflow(G2,C)`, `add_sequenceflow(G1,X)`,
`add_sequenceflow(X,G2)`.

Rationale This pattern is usually applied when a process needs to be refined in order to cover variations in the flow of activities.

Example The invoice creation process at Amazon is extended for EU countries as different tax and customs regulations apply. Different informations need to be printed on the invoice and declarations have to be made for customs compliance. The process therefore needs to be split, an alternative branch has to added for orders that originate in EU countries.

Category Extension patterns

Related change patterns `Red.Alt.Ini.Comp` is reverse.

4.7 Pattern `Ext.Alt.Add.Comp`: Complete Additional Alternative Extension

Description An additional activity element is placed as an alternative branch to an already existing set of alternative activity branches. Additional sequence flows and an additional activity element are inserted.

Illustration see figure 14.

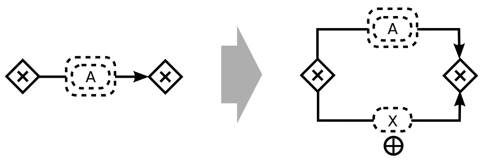


Figure 14: Change pattern: Complete additional alternative extension

Signature ..
`add_activity(X)`, `add_sequenceflow(G1,X)`, `add_sequenceflow(X,G2)`.

Rationale This pattern is usually applied when variations of a process need to be extended with an addtional variation (alternative flow of activities).

Example The invoice creation process at Amazon which covers both US market and EU is extended for China. Different informations need to be printed on the invoice and declarations have to be made for customs compliance. The process therefore needs to be extended, an alternative branch has to added for orders that originate in China.

Category Extension patterns

Related change patterns `Red.Alt.Add.Comp` is reverse.

4.8 Pattern `Ext.Iter`: Iterative Extension

Description A sequence of activities is extended with a loop back branch. Therefore alternative gateways are inserted and a sequence flow is inserted that leads back to the first activity in the sequence of activities.

Illustration see figure 15.

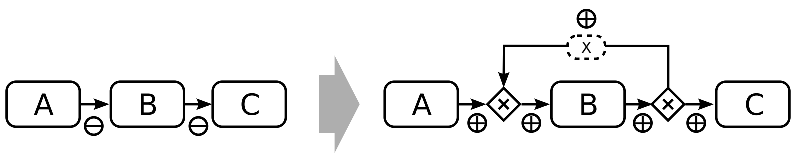


Figure 15: Change pattern: Iterative extension

Signature ..
`delete_sequenceflow(A,B), delete_sequenceflow(B,C), add_activity(X), add_sequenceflow(A,G1), add_sequenceflow(G1,B), add_sequenceflow(B,G2), add_sequenceflow(G2,C), add_sequenceflow(G2,G`

Rationale This pattern is applied when a sequence of activities may be repeatedly performed until a predefined goal, result or output is achieved or condition is fulfilled.

Example At Amazon a customer may repeatedly browse the catalog, select products and add them to the shopping cart until she decides to check-out.

Category Extension patterns

Related change patterns `Red.Iter` is reverse.

4.9 Pattern `Parallel`: Parallelization

Description Activity elements that are arranged as a sequence are now rearranged in parallel. For this purpose parallel gateways are inserted and associated sequence flow edges must be deleted and added.

Illustration see figure 16.

Signature `add_gateway(G1)`, `add_gateway(G2)`, `delete_sequenceflow(A,B)`, `add_sequenceflow(G1,A)`, `add_sequenceflow(G1,B)`, `add_sequenceflow(A,G2)`, `add_sequenceflow(B,G2)`.

Rationale This pattern is usually applied when a process needs to rearranged to reduce lead time. Although the same set of activities is performed as before the lead time from process start event to process end event is reduced.

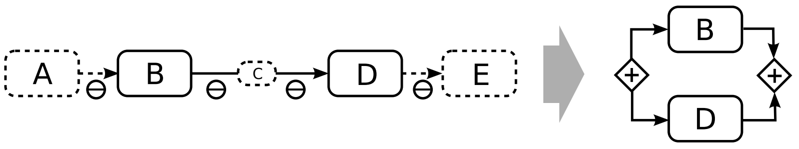


Figure 16: Change pattern: Parallelization

Example The invoice creation activity which has been performed after the packing activity is arranged in parallel to the packing an picking process to reduce devlivery time (time-to-customer).

Category Adaption patterns

Related change patterns `Serial` is reverse.

4.10 Pattern `Alter`: Alternation

Description Activity elements that are arranged as a sequence are now rearranged as alternative branches. For this purpose alternative gateways are inserted and associated sequence flow edges must be deleted and added.

Illustration see figure 17.

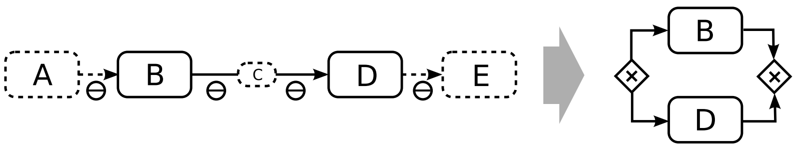


Figure 17: Change pattern: Alternation

Signature `add_gateway(G1)`, `add_gateway(G2)`, `delete_sequenceflow(A,B)`, `add_sequenceflow(G1,A)`, `add_sequenceflow(G1,B)`, `add_sequenceflow(A,G2)`, `add_sequenceflow(B,G2)`.

Rationale This pattern is applied when sequences of activities are changed because of constraints that allow only for exclusive execution of activities.

Example A payment process is changed in a way that mixing of payment methods (credit card payment and bank account withdrawal) for one invoice payable is replaced by a logic where either credit card payment or withdrawal is possible but not both.

Category Adaption patterns

Related change patterns

4.11 Pattern `Commut`: Commutation

Description Activity elements are replaced with each other with regard to their position in a sequence of activities.

Illustration see figure 18.

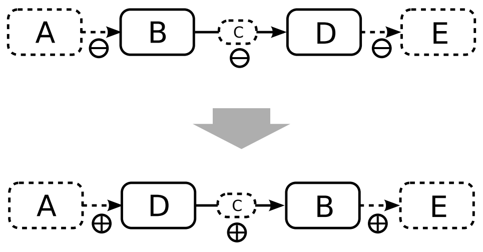


Figure 18: Change pattern: Commutation

Signature `delete_sequenceflow(B,D)`, `add_sequenceflow(D,B)`.

Rationale This pattern is usually applied when it turns out that the sequence of activities performed is not optimal to achieve a predefined goal. Rearranging two activities

Example Exchanging invoice creation activity with creditability check as it turns out that exact amount including taxes and shipping fees is needed beforehand to perform a valid creditability check.

Category Adaption patterns

Related change patterns `Commut` is reverse.

5 Developing algorithms for automated detection of change patterns in process model revisions

In this section we describe several algorithms that we designed for the purpose of detecting change patterns in a set of atomic change operations. One of these Algorithm 1 computes the difference between two succeeding model revisions. Algorithm 2 is the main routine that inspects the sets of change operations and provides entry points for sub routines that are able to check for the existence of a particular change pattern as specified in section 4.

5.1 Basic algorithms for preprocessing

Algorithm 1 processes succeeding revisions of a process model r_i, r_{i+1} and compares them to identify added, deleted and unchanged model elements. Model elements are distinguished into $nodes$ and $edges$. The variables holding the revisions of a process model are collections of model elements. The comparison is performed through a nested loop that compares element by element. We do this in both directions to detect both added elements – those that are not existing in r_i – and deleted elements – those that do not exist anymore in r_{i+1}. In the course of doing so we also identify those elements that are identical and have not been changed at all. Finally we obtain collections for added elements $nodes_{added}, edges_{added}$, deleted elements $nodes_{deleted}, edges_{deleted}$ and unchanged elements $nodes_{identical}, edges_{identical}$. These collections represent the model diff and are used as input for all subsequent processing.

```
//declare data variables
data: $r_i, r_{i+1}$ as list; $nodes_{deleted}$, $nodes_{identical}$, $edges_{added}$, $edges_{deleted}$ as list;
//loop through elements
foreach $m_i$ in $r_i$ do
    $is\_found$ := false;
    foreach $m_{i+1}$ in $r_{i+1}$ do
        //check for identical elements
        if $m_i === m_{i+1}$ then
            if $m_i.type ==$ 'node' then append($nodes_{identical}$, $m_i$);
            if $m_i.type ==$ 'edge' then append($edges_{identical}$, $m_i$);
            $is\_found$ := true; break;
        end
    end
    //
    if not $is\_found$ then
        if $m_i.type ==$ 'node' then append($nodes_{deleted}$, $m_i$);
        if $m_i.type ==$ 'edge' then append($edges_{deleted}$, $m_i$);
    end
end
foreach $m_{i+1}$ in $r_{i+1}$ do
    $is\_found$ := false;
    foreach $m_i$ in $r_i$ do
        //check for identical elements
        if $m_i === m_{i+1}$ then
            $is\_found$ := true; break;
        end
    end
    //
    if not $is\_found$ then
        if $m_{i+1}.type ==$ 'node' then append($nodes_{added}$, $m_{i+1}$);
        if $m_{i+1}.type ==$ 'edge' then append($edges_{added}$, $m_{i+1}$);
    end
end
```

Algorithm 1: Main algorithm to compute difference from revisions

Algorithm 2 takes the previously obtained diff (set of collections of unchanged, added, deleted nodes and edges) as input and searches for subsets of nodes which are somehow connected and therefore are candidates for a change pattern. To narrow the search space we assume that both extensions of a model or reductions either refer to a single node or to a pair of nodes from the set of identical and unchanged nodes $n_{identical}$. Thus changes that do not refer to any existing part of a model are omitted. According to our taxonomy of change patterns we distinguish between extension to the right where a model is extended by appending a sequence of activity nodes to a node without outgoing edges (a terminal node) and extensions to the left where a model is extended by prepending a sequence of activities to a node without ingoing edges (initial node) and insertions where a model is extended by inserting a sequence of activities between two succeeding activity nodes.

In algorithm 2 we refer to algorithm 3 which we use to identify subsets of connected model elements from a model diff. The algorithm loops through all model elements (added and deleted) and searches for an edge that refers to the reference node(s) through its source or target property. Subsequently it searches for the next node that has been referred to by the source or target property of the edge,

```
//declare data variables
data: $n_{source}$, $n_{target}$ as object; $type$ as string; $nodes_{added}$, $nodes_{deleted}$, $nodes_{identical}$, $edges_{added}$, $edges_{deleted}$ as list
//loop through all identical and unchanged nodes
foreach $n_1$ in $nodes_{identical}$ do
    //if node is a terminal node (no outgoing edges)..
    if is_terminal($n_1$) then
        //..check for extension patterns to right
        //..get traces of added and deleted elements
        $traces_{added} := get\_traces\_between\_two\_nodes(n_1, null, e_{added})$
        $traces_{deleted} := get\_traces\_between\_two\_nodes(n_1, null, e_{deleted})$
        //check for change pattern Ext.Ser.R
        ...see algorithm 5
        //check for other change patterns
        ...
    end
    //if node is a inital node (no ingoing edges)..
    if is_initial($n_1$) then
        //..check for extension patterns to left
        //..get traces of added and deleted elements
        $traces_{added} := get\_traces\_between\_two\_nodes(null, n_1, e_{added})$
        $traces_{deleted} := get\_traces\_between\_two\_nodes(null, n_1, e_{deleted})$
        //check for change pattern Ext.Ser.L
        ...see algorithm 6
        //check for other change patterns
        ...
    end
    //..  and pairwise combinations of identical and unchanged nodes
    foreach $n_2$ in $nodes_{identical}$ do
        //get traces..
        if $n_1 ! == n_2$ then
            //..of added and deleted elements
            $traces_{added} := get\_traces\_between\_two\_nodes(n_1, n_2, e_{added})$
            $traces_{deleted} := get\_traces\_between\_two\_nodes(n_1, n_2, e_{deleted})$
            //check for change pattern Ext.Ser.Ins
            ...see algorithm 4
            //Check for other change patterns
            ...
        end
    end
end
```

Algorithm 2: Main algorithm to detect change patterns from a revision

then it searches for the next edge and so on. Basically we use a depth-first approach (see for example in [Even, 2011] p. 41) that recursively steps through a directed graph and returns all possible paths as a collection of traces. Traces are themselves collections of connected model elements (sequences of activity nodes). Having a trace collection for each added or deleted model part it is possible to examine these traces for the occurrence of a signature that can be used to identify a change pattern. From a computational point of view this algorithm is the most resource intensive part.

5.2 Exemplary algorithms for detection of change patterns

In this section we present exemplary algorithms that check each collection of traces for the occurrence of a pattern signature. As explained previously a signature is defined as a set of change operations that is at least needed to identify a change pattern. To accomplish this task we first check for the type of node the trace refers to. In the case of a Serial Insert pattern this means that we have to check each pair of nodes if they are of type ‘activity’ as a serial insert between gateway nodes would be a Complete Additional Alternative `Ext.Add.Alt.Comp` or Parallel `Ext.Add.Par.Comp` Extension pattern.

Algorithm 4 shows a routine that checks for the existence of a Serial Insertion `Ext.Ser.Ins` pattern. For a pair of identical and unchanged nodes we check whether a single sequence flow edge (trace-length equals 1) exists that has been deleted and refers to the two nodes. To be precise we check if the first edge has node n_1 as its source and whether the last edge in the trace has node n_2 as its target. Next we check whether a trace with trace-length greater than 1 exists that has been subsequently added between the two nodes.

```
function $get\_traces\_between\_two\_nodes(n_1,\ n_2$ as object, $edges$ as list$)$
    data: $traces$ as list, $trace$ as string
    foreach $e$ in $edges$ do
        //check if an outgoing edge exists that refers to left node
        if $e.source == n_1$ then
            //if last element in trace or terminal
            if $e.target == n_2 \mid\mid is\_terminal(e.target)$ then
                //append last edge to trace
                $append(trace, e)$
                //append trace to collection of traces
                $append(traces, trace)$
            //if not last element in trace
            else
                //append identified edge to trace
                $append(trace, e)$
                //append target node of edge as well to trace
                $append(trace, e.target)$
                //recursively look for next elements in trace
                $get\_traces\_between\_two\_nodes(e.target, n_2)$
            end
        end
    end
    return traces
end
```

Algorithm 3: Algorithm to get added or deleted activity sequences (traces) between two nodes

```
//check for change pattern Ext.Ser.Ins
if $length(traces_{deleted}) > 0$ and $length(traces_{added}) > 0$ then
    //check context nodes are of type activity
    if $n_1.type == 'activity'$ and $n_2.type == 'activity'$ then
        //check if exactly one trace exists that contains a single sequence flow edge
        foreach $trace$ in $traces_{deleted}$ do
            if $length(trace) == 1$ then
                $deletion\_pattern\_exists :=$ true
                break
            end
        end
        //if deletion pattern has been found, ..
        if $deletion\_pattern\_exists$ then
            foreach $trace$ in $traces_{added}$ do
                //check if exactly one trace exists that contains at least two sequence flow edges and one activity node
                if $length(trace) > 1$ then
                    //collect detected change pattern
                    $append(patterns, array('Ex.Ser.Ins', n_1, n_2, traces_{added}, traces_{deleted}))$
                    break
                end
            end
        end
    end
end
```

Algorithm 4: Algorithm to detect a `Ext.Ser.Ins` change pattern

Algorithm 5 shows a routine that checks for the existence of a Serial Extension to Right `Ext.Ser.R` pattern. For a terminal node n from $nodes_{identical}$ we check whether it is of type 'activity' and if at least one trace exists that has been added and refers to the node n. The trace needs to contain at least more than one model element. In algorithm 6 a similar routine is illustrated for model parts that have been prepended to an initial activity node. We do not explain this algorithm here as it is analogue to 5.

6 Implementation

The previously described algorithms were implemented in a publicly available open source process modeling environment `PWiki`[6]. `PWiki` is a research prototype and as the name suggests is designed to support collaborative distributed process modeling in the cloud Erol [2012]. The central object of work in `PWiki` is a page. Pages are created, edited, deleted and linked to each other. The total

[6]https://launchpad.net/weasel2

```
   //check for change pattern Ext.Ser.R
   //check if terminal node is of type activity
1  if $n.type == 'activity'$ then
       //check if number of traces found for node is not empty
2      if $length(traces_{added}) > 0$ then
           //collect detected change pattern
3          $append(patterns, array('Ex.Ser.R', n, null, traces_{added}, null))$
4          break
5      end
6  end
```

Algorithm 5: Algorithm to detect a `Ext.Ser.R` change pattern

```
   //check for change pattern Ext.Ser.L
   //check if terminal node is of type activity
1  if $n.type == 'activity'$ then
       //check if number of traces found for node is not empty
2      if $length(traces_{added}) > 0$ then
           //collect detected change pattern
3          $append(patterns, array('Ex.Ser.L', null, n, traces_{added}, null))$
4          break
5      end
6  end
```

Algorithm 6: Algorithm to detect a `Ext.Ser.L` change pattern

of linked pages forms a body of knowledge which can be dynamically extended by arbitrary authors anytime from anywhere. Pages in `PWiki` are extended with a process model editor named `graphel`[7] that allows for modeling processes according to different process modeling notations (e.g. BPMN 2.0, EPC). To ensure traceability and recoverability of changes in `PWiki` each page maintains a revision history that reveals the date and time of a page revision, the author and if maintained by the author comments regarding the changes applied. For the purpose of validating our approach we extended this revision history feature with the capability of detecting change patterns in model revisions. Thus providing automatically created meaningful annotations of model revisions. By now the feature has been implemented for BPMN 2.0 based process models and covers selected extension patterns. Namely, `Ext.Ser.R`, `Ext.Ser.L`, `Ext.Ser.Ins`, `Ext.Par.Ini`, `Ext.Par.Add`, `Ext.Alt.Ini`, `Ext.Alt.Add`

6.1 Support for standards based modeling languages

Supporting change pattern detection from BPMN 2.0 based process model diffs requires some extensions to the above presented approach. BPMN 2.0 is a rather complex notation based on a comprehensive meta model that covers all aspects for describing software enabled business processes. Even the set of basic modeling elements of BPMN 2.0 covers twelve elements, the extended set of elements covers a multiple of twelve elements including all variations of base elements that are necessary to describe business processes. However in the first phase of our implementation we included only those elements that are needed to describe the control flow of a process. Elements to describe other aspects of a process (e.g. resources, data) are not included, see figure 21. For implementing the above general detection algorithms the introduction of additional elements required the differentiation between different types of activities (e.g. activities, tasks), different types of events (e.g. start, end and intermediate) events and different types of gateways (e.g. exclusive versus inclusive and parallel gateways). For example, within the exemplary algorithm `Ext.Ser.Ins` (see algorithm 4) this led to a more complex case-handling in line 2 where changes between different types of activity nodes needed to be handled differently.

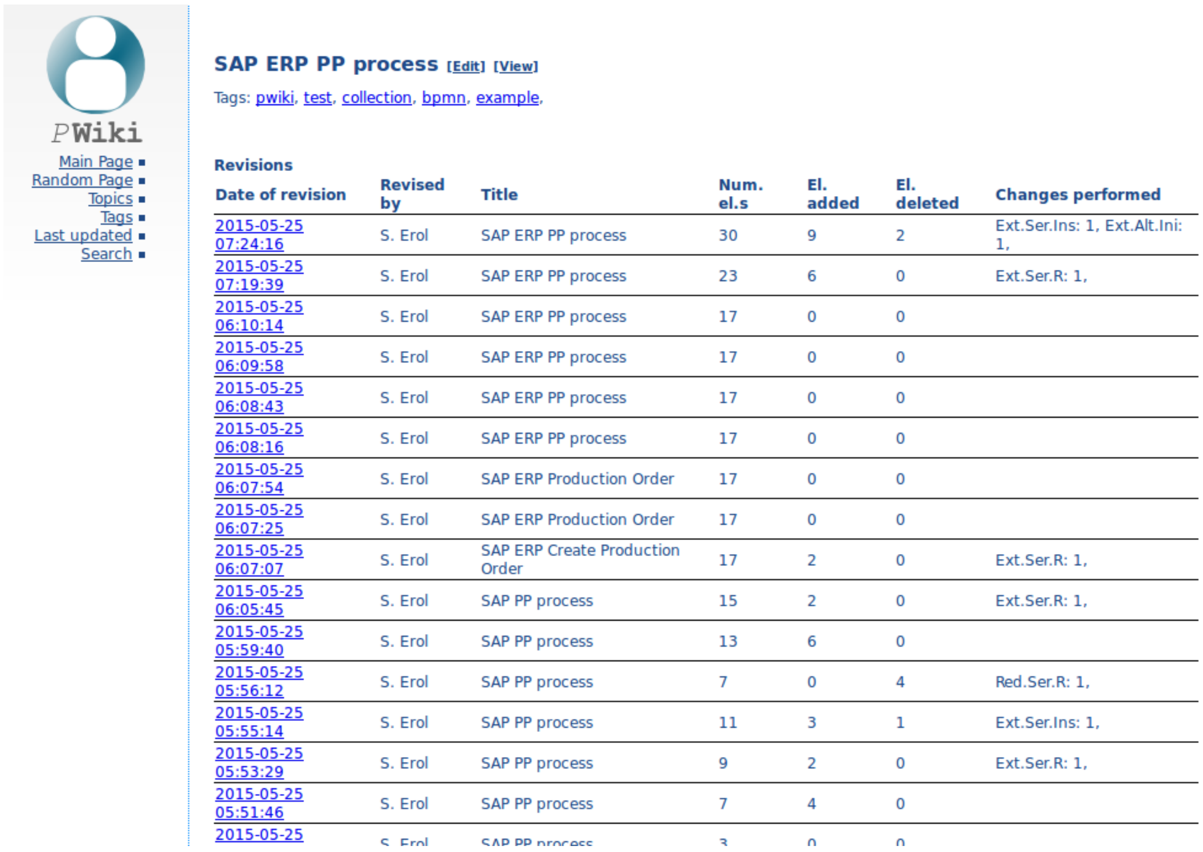
In figure 22 a screenshot of the process model editor is shown that is part of the process modeling

[7] https://launchpad.net/graphel

environment `PWiki`. As can be seen from the screenshot the model editor is embedded in a HTML form page that is used to create process model pages (see figure 23) including meta information such as the title of the model, keywords and data about the time, author of creation and last modification. Each of which is important information for maintaining a related revision history.

6.2 Visual presentation of change patterns

Figure 19 shows the revision history of a process model page in tabular from. It includes information on the date and time of the creation of the revision, the author and the title of the page. It reveals as well information on the number of model elements (nodes an edges) the current revision consists of and the number of elements added and deleted. The latter information is the result of model comparison performed before detection of change patterns. In the last column the detected change pattern with their respective short names can be seen and number of occurrence can be seen. In some rows one can see empty cells – no change pattern detected – although elements have been added or deleted. This is due to the fact that some of the predefined criteria has not been fulfilled to detect a change pattern. In other words that individual atomic change operations have been made that could not be related to some of the change patterns from our taxonomy. I considered these cases as candidates for further examination as they could potentially be special cases that where not considered beforehand. Other rows from the revision history show that the process model itself has not been changed but the title which as well resulted in empty cells regarding the change patterns.



SAP ERP PP process [Edit] [View]

Tags: pwiki, test, collection, bpmn, example,

Revisions

Date of revision	Revised by	Title	Num. el.s	El. added	El. deleted	Changes performed
2015-05-25 07:24:16	S. Erol	SAP ERP PP process	30	9	2	Ext.Ser.Ins: 1, Ext.Alt.Ini: 1,
2015-05-25 07:19:39	S. Erol	SAP ERP PP process	23	6	0	Ext.Ser.R: 1,
2015-05-25 06:10:14	S. Erol	SAP ERP PP process	17	0	0	
2015-05-25 06:09:58	S. Erol	SAP ERP PP process	17	0	0	
2015-05-25 06:08:43	S. Erol	SAP ERP PP process	17	0	0	
2015-05-25 06:08:16	S. Erol	SAP ERP PP process	17	0	0	
2015-05-25 06:07:54	S. Erol	SAP ERP Production Order	17	0	0	
2015-05-25 06:07:25	S. Erol	SAP ERP Production Order	17	0	0	
2015-05-25 06:07:07	S. Erol	SAP ERP Create Production Order	17	2	0	Ext.Ser.R: 1,
2015-05-25 06:05:45	S. Erol	SAP PP process	15	2	0	Ext.Ser.R: 1,
2015-05-25 05:59:40	S. Erol	SAP PP process	13	6	0	
2015-05-25 05:56:12	S. Erol	SAP PP process	7	0	4	Red.Ser.R: 1,
2015-05-25 05:55:14	S. Erol	SAP PP process	11	3	1	Ext.Ser.Ins: 1,
2015-05-25 05:53:29	S. Erol	SAP PP process	9	2	0	Ext.Ser.R: 1,
2015-05-25 05:51:46	S. Erol	SAP PP process	7	4	0	
2015-05-25	S. Erol	SAP PP process	3	0	0	

Figure 19: PWiki revision history feature

7 Related work

Our work mainly relates to three other approaches that we like to mention here. Küster et al. [2008] introduced so called compound changes which represent minimal meaningful sets of atomic change operations. Additionally, so called sequences of compound changes are introduced that represent sets of dependent change operations where for each change operation at least one dependency exists. Change operations may not belong to more than one sequence and may not have any dependency with change operations from other sequences. However, the authors limited their approach to a specific meta-model and a predefined set of compound changes that do not reflect the rationale of change patterns applied by modelers. The approach presented focuses mainly on the problem of detecting compound model changes from process model comparisons in scenarios where a change log (a timely order of changes) does not exist. They show as well how to present these logically connected changes to a modeler during merging model revisions.

The first explicit investigation of change patterns in the context of process modeling is provided by Weber et al. [2008]. In their work they classify and describe process model changes through a pattern language. Their change patterns are the result of two empirical studies of 59 process models and their documented adaptions. They also describe the formal semantics of process model patterns in [Rinderle-Ma et al., 2008] and suggest a set of software features to support typical process model changes according to the patterns identified. Their work differs in several ways. First their source of data for identifying change patterns is not clearly documented as they do not give evidence of the process model revisions studied. Second, they do not build a taxonomy – a classification of change patterns and do not explain relations between them sufficiently. Third, they do not provide a means to detect a change patterns although they have formalized their semantics in a comprehensive and rigorous way.

Langer et al. [2013] investigated change patterns (they call them diff patterns) in the context of UML[8] class models. They provide fundamental work with regard to specifying change patterns through so called signatures - sets of primitive change operations that are used to identify a particular change pattern. They also provide algorithms for detecting change patterns in a three-phased approach. Accordingly, In the first phase a model diff is checked for the occurrence of change patterns through so called signatures which in contrary to our approach reflect processable representations of the involved set of atomic change operations and the diff model. In the next steps change patterns are checked whether corresponding pre- and postconditions exist within the original revision and the final revision that prove the actual application of a change.

8 Conclusion and Outlook

In this report we document recent research that conducted as part of a larger research project that aims at investigating concepts, methods and tools to facilitate model-driven engineering of organizational processes. The approach described here aims at providing a taxonomy of typical changes to process models and their rationale. We used a pattern language to comprehensively identify and describe change patterns observed in a large collection of process models and their revision histories. The pattern language used consists of a graphical language to describe these patterns unambiguously and a pattern template to capture the rationale and relations between change patterns. We also postulated a set of algorithms that shows how such patterns can be detected from

[8]http://www.uml.org

process model revision comparisons where no timely order of involved atomic changes (a change log) exists.

The contribution of this research effort is therefore threefold:

- We provide an empirically grounded taxonomy of change patterns in process modeling.
- We provide a language to describe the properties of such change patterns and their rationale which is important with regard to software-based modeling support.
- We provide algorithms that show how such change patterns can be detected from model comparisons in the absence of a change log.

Our future research efforts will aim at providing algorithmic implementations of all patterns and a modeling environment that can be used to generate respective algorithms automatically. What remains as well open is the evaluation of the algorithms so far. For this purpose we plan to run the algorithms against the large corpus of process models we used to identify patterns. This will close the loop of manual identification to formalization, validation and evaluation of algorithmic performance.

9 References

Alexander, C. (1964). *Notes on the Synthesis of Form*, volume 5. Harvard University Press.

Davies, I., Green, P., Rosemann, M., Indulska, M., and Gallo, S. (2006). How do practitioners use conceptual modeling in practice? *Data and Knowledge Engineering*, 58(58):358–380.

Dijkman, R., Rosa, M., and Reijers, H. (2012). Managing large collections of business process models—current techniques and challenges. *Computers in Industry*, 63(2):91.

Erol, S. (2012). *Design and Evaluation of a Wiki-based Collaborative Process Modeling Environment*. PhD thesis, WU Vienna.

Erol, S. and Neumann, G. (2013). Handling concurrent changes in collaborative process model development: a change-pattern based approach. In *Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2013 17th IEEE International*, pages 250–257. IEEE.

Even, S. (2011). *Graph algorithms*. Cambridge University Press.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

Giaglis, G. M. (2001). A taxonomy of business process modeling and information systems modeling techniques. *International Journal of Flexible Manufacturing Systems*, 13(2):209–228.

Küster, J., Gerth, C., Forster, A., and Engels, G. (2008). Detecting and resolving process model differences in the absence of a change log. In *Proc. 6th Int. conf. on Business Process Management. BPM.*, volume 5240, page 244. Springer.

La Rosa, M., Reijers, H., Van der Aalst, W., Dijkman, R., Mendling, J., Dumas, M., and García-Bañuelos, L. (2011). Apromore: An advanced process model repository. *Expert Systems with Applications*, 38(6):7029–7040.

Langer, P., Wimmer, M., Brosch, P., Herrmannsdörfer, M., Seidl, M., Wieland, K., and Kappel, G. (2013). A posteriori operation detection in evolving software models. *Journal of Systems and Software*, 86(2):551–566.

Mens, T. (2002). A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28:449–461.

OMG (2011). Business Process Model and Notation (BPMN) Version 2.0. Technical report, OMG.

Rinderle-Ma, S., Reichert, M., and Weber, B. (2008). On the formal semantics of change patterns in process-aware information systems. In *Conceptual Modeling-ER 2008*, pages 279–293. Springer.

Scheer, A. W. (1998). *ARIS – vom Geschäftsprozess zum Anwendunssystem*. Springer.

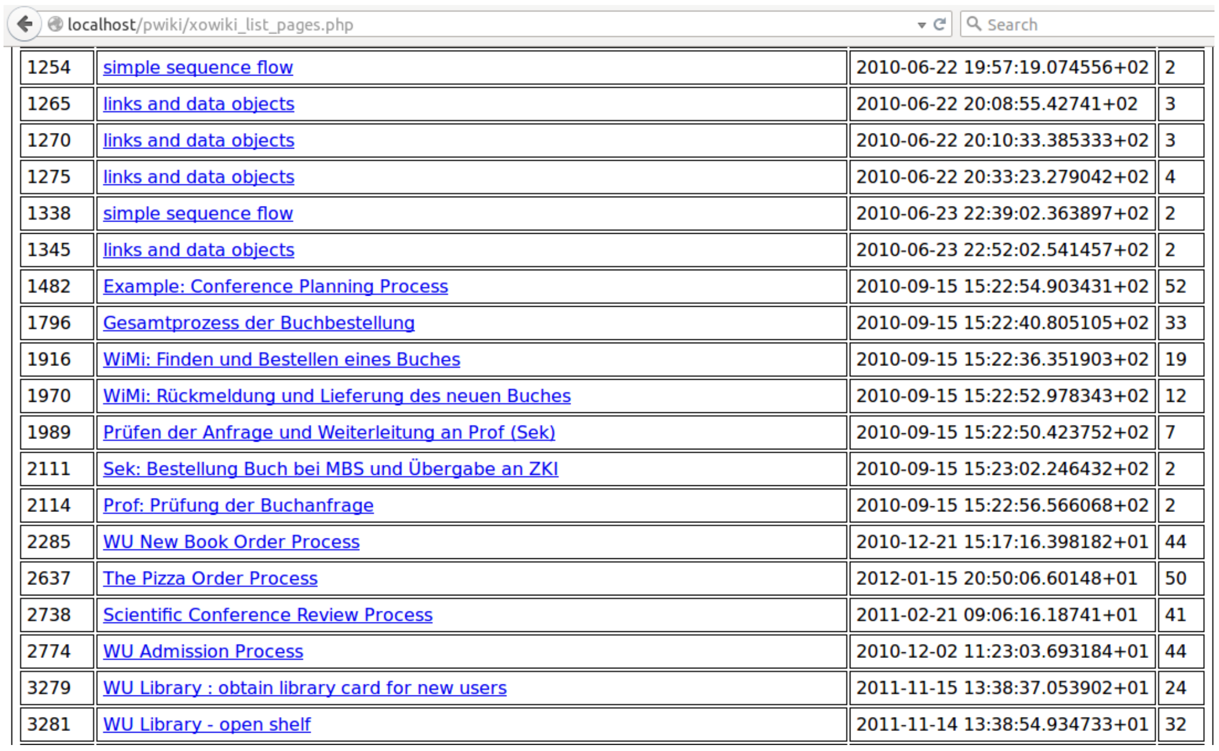
van der Aalst, W. (1999). Formalization and verification of event-driven process chains. *Information and Software technology*, 41(10):639–650.

Weber, B., Reichert, M., Mendling, J., and Reijers, H. A. (2011). Refactoring large process model repositories. *Computers in Industry*, 62(5):467–486.

Weber, B., Reichert, M., and Rinderle-Ma, S. (2008). Change patterns and change support features–enhancing flexibility in process-aware information systems. *Data & knowledge engineering*, 66(3):438–466.

Weber, B., Rinderle, S., and Reichert, M. (2007). Change patterns and change support features in process-aware information systems. In *Advanced IS Engineering*, pages 574–588. Springer.

10 Appendix



1254	simple sequence flow	2010-06-22 19:57:19.074556+02	2
1265	links and data objects	2010-06-22 20:08:55.42741+02	3
1270	links and data objects	2010-06-22 20:10:33.385333+02	3
1275	links and data objects	2010-06-22 20:33:23.279042+02	4
1338	simple sequence flow	2010-06-23 22:39:02.363897+02	2
1345	links and data objects	2010-06-23 22:52:02.541457+02	2
1482	Example: Conference Planning Process	2010-09-15 15:22:54.903431+02	52
1796	Gesamtprozess der Buchbestellung	2010-09-15 15:22:40.805105+02	33
1916	WiMi: Finden und Bestellen eines Buches	2010-09-15 15:22:36.351903+02	19
1970	WiMi: Rückmeldung und Lieferung des neuen Buches	2010-09-15 15:22:52.978343+02	12
1989	Prüfen der Anfrage und Weiterleitung an Prof (Sek)	2010-09-15 15:22:50.423752+02	7
2111	Sek: Bestellung Buch bei MBS und Übergabe an ZKI	2010-09-15 15:23:02.246432+02	2
2114	Prof: Prüfung der Buchanfrage	2010-09-15 15:22:56.566068+02	2
2285	WU New Book Order Process	2010-12-21 15:17:16.398182+01	44
2637	The Pizza Order Process	2012-01-15 20:50:06.60148+01	50
2738	Scientific Conference Review Process	2011-02-21 09:06:16.18741+01	41
2774	WU Admission Process	2010-12-02 11:23:03.693184+01	44
3279	WU Library : obtain library card for new users	2011-11-15 13:38:37.053902+01	24
3281	WU Library - open shelf	2011-11-14 13:38:54.934733+01	32

Figure 20: Screenshot of list of models that were used to identify and classify changes. The models are publicly available at http://www.erol.at/pwiki/xowiki_list_pages.php

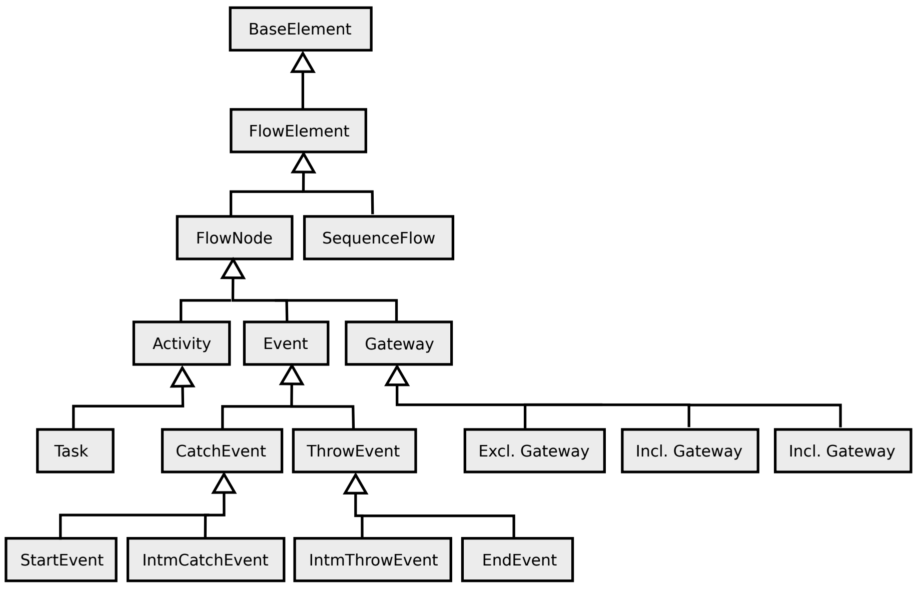
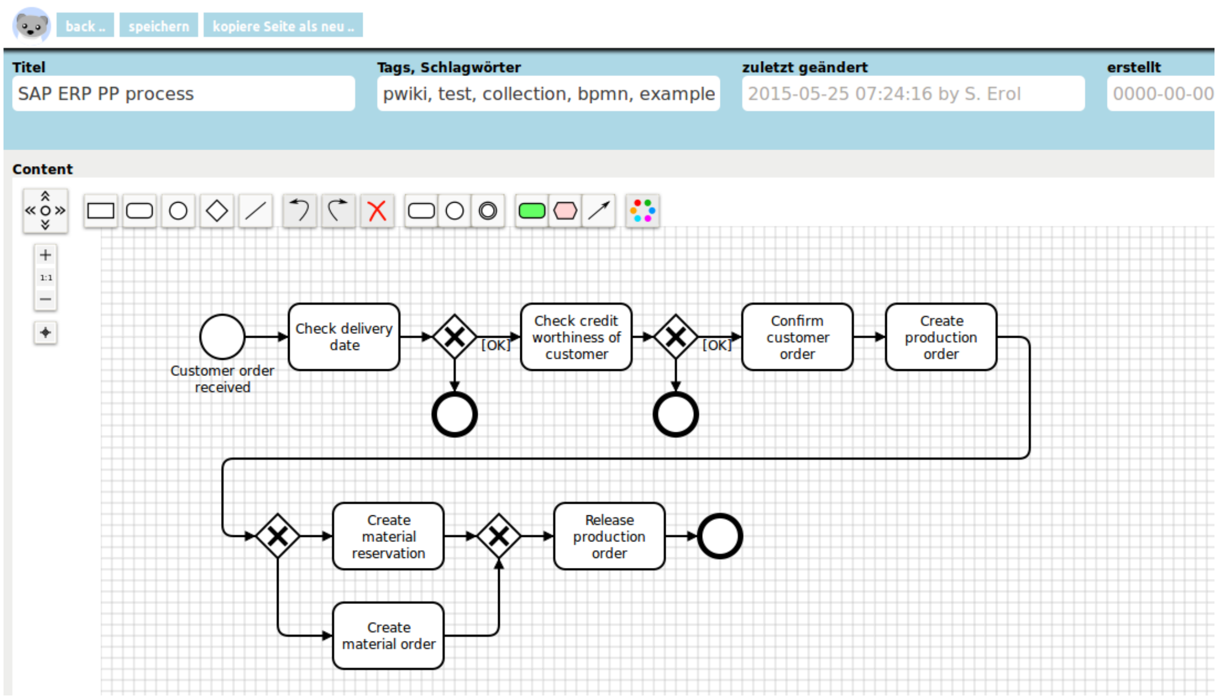


Figure 21: Implemented part of BPMN 2.0



Titel: SAP ERP PP process

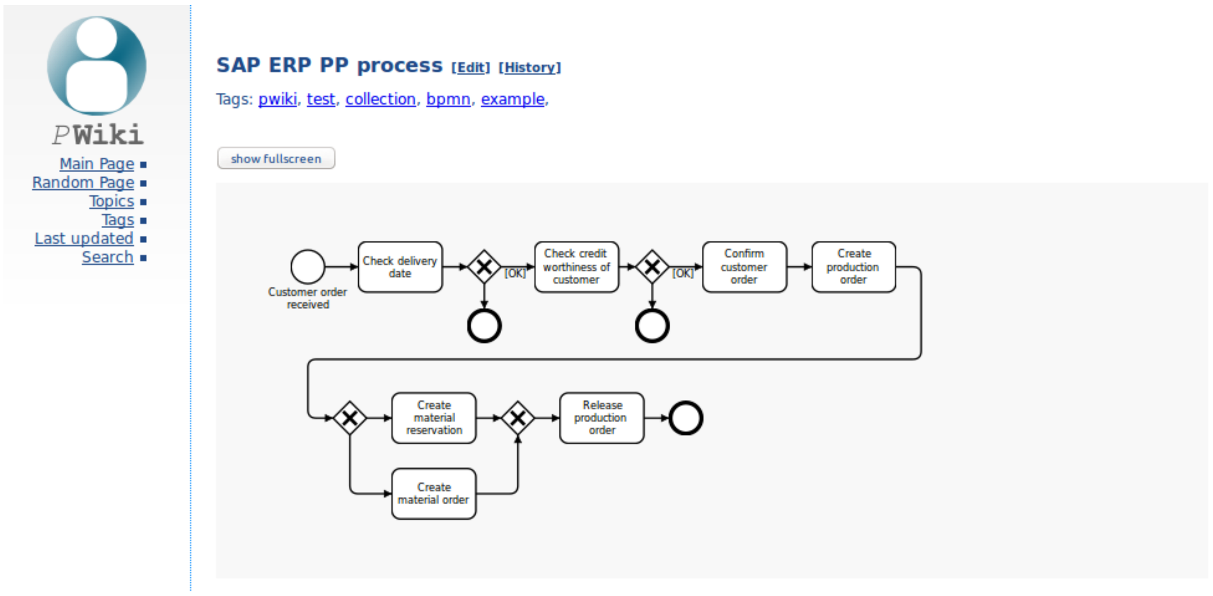
Tags, Schlagwörter: pwiki, test, collection, bpmn, example

zuletzt geändert: 2015-05-25 07:24:16 by S. Erol

erstellt: 0000-00-00

Content

Figure 22: PWiki process model editor



SAP ERP PP process [Edit] [History]

Tags: pwiki, test, collection, bpmn, example,

Figure 23: PWiki process model page

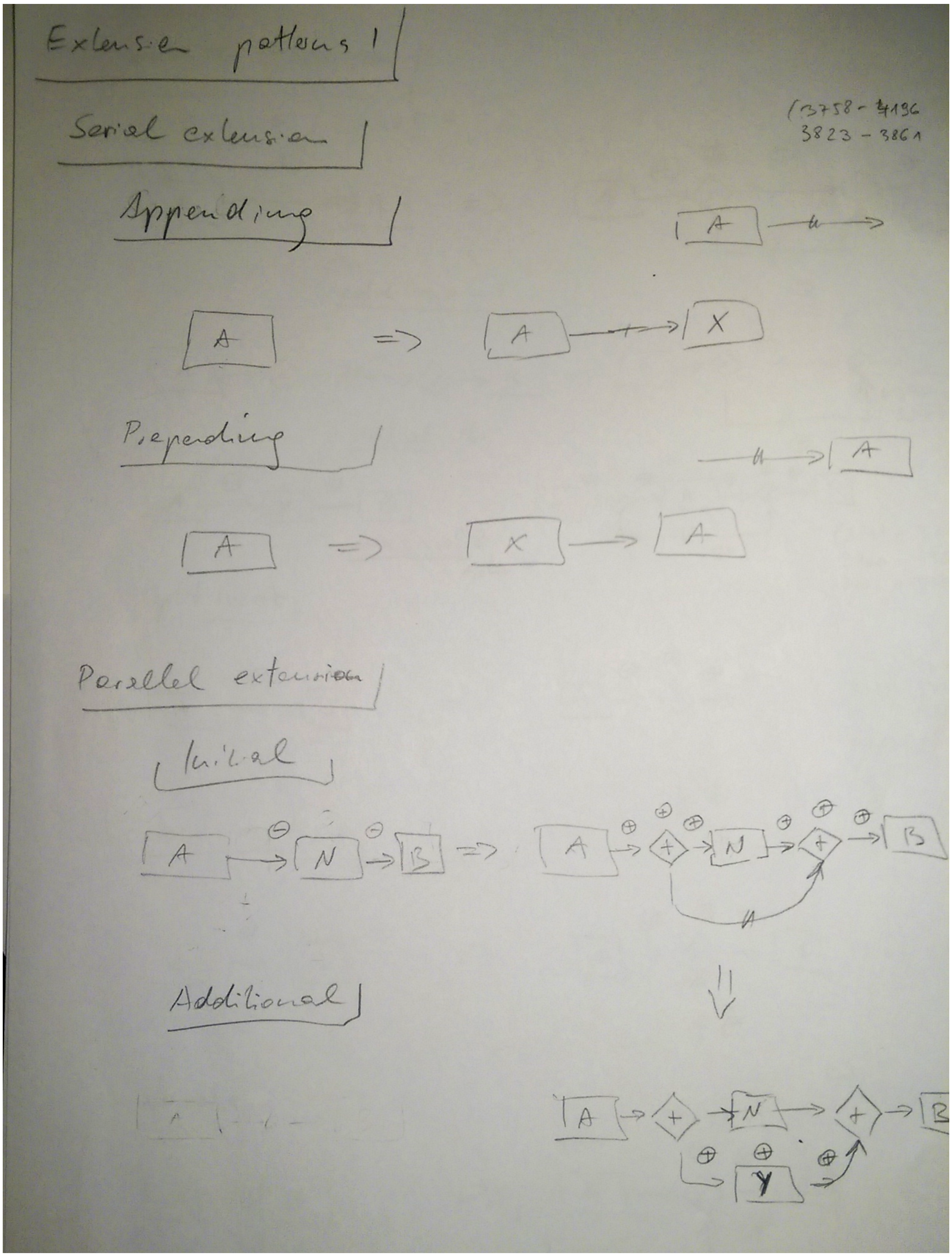


Figure 24: Paper card used during studying and recording model revisions