

# Keeping the Cost of Process Change Low through Refactoring

Barbara Weber<sup>1</sup> and Manfred Reichert<sup>2</sup>

<sup>1</sup>Quality Engineering Research Group, University of Innsbruck, Austria  
Barbara.Weber@uibk.ac.at

<sup>2</sup>Information Systems Group, University of Twente, The Netherlands  
m.u.reichert@cs.utwente.nl

**Abstract.** With the increasing adoption of process-aware information systems (PAIS) large process model repositories have emerged. Over time respective models have to be re-aligned to the real world business processes through customization or adaptation. This bears the risk that model redundancies are introduced and complexity is increased. If no continuous investment is made in keeping models simple, changes are becoming increasingly costly and error-prone. Although refactoring techniques are widely used in software engineering to address related problems, this does not yet constitute state-of-the art in business process management. Consequently, process designers either have to refactor process models by hand or can not apply respective techniques at all. In this paper we propose a set of techniques for refactoring large process repositories, which are behaviour-preserving. The proposed refactorings enable process designers to effectively deal with model complexity by making process models easier to change, less error-prone and better understandable.

**Keywords:** Process-aware Information Systems, Process Model Quality, Process Model Refactoring, Cost of Process Change

## 1 Introduction

Process-aware Information Systems (PAIS) offer promising perspectives for enterprise computing and are increasingly used to support business processes at an operational level [1]. In contrast to data- or function-oriented information systems (IS), PAIS strictly separate process logic from application code, relying on explicit *process models* which provide the schemes for process execution. This allows for a separation of concerns, which is a well established principle in computer science to increase maintainability and to reduce cost of change [2].

With the increasing adoption of PAIS large process repositories have emerged. Over time corresponding process models have to be adapted at different levels to meet new business, customer and regulatory needs, and to ensure that PAIS remain aligned with the processes as executed in the real world. Typical adaptations include the customization of (reference) process models to specific needs

of a customer [3, 4] or – at the operational level – the adaptation of running process instances to cope with exceptional situations [5]. Like software programs degenerate when adding more and more code or introducing changes by different developers [6], process adaptations bear the risk that model repositories are becoming increasingly complex and difficult to maintain over time.

In software engineering (SE) refactoring techniques have been widely used to address related problems and to ensure that code bases remain maintainable over time [7, 8]. Refactoring allows programmers to restructure a software system without altering its behaviour. Refactoring is typically used to improve code quality by removing duplication, improving readability, simplifying software design, or adding flexibility [9]. Examples of SE refactoring techniques include the renaming of a class to foster understandability or the extraction of a method from an existing code block to remove redundant code fragments.

Process modeling is often referred to as *programming in the large* [10, 11]. Thereby, a process schema is comparable to a software program specifying the inputs and outputs of activities as well as the control and data flow between them. Despite these similarities refactoring is not yet established in the field of business process management (BPM) and existing process modeling tools only provide limited refactoring support. Consequently, process designers either have to refactor process models by hand or can not apply respective techniques at all.

This paper adopts SE refactoring techniques to the needs of process modeling and complements them with additional refactorings specific to BPM. In particular, we describe techniques suitable for refactoring large process repositories, where we can find both collections of inter-related process models and process variants derived from generic models (e.g., reference process models). The former consist of a set of models, which may refer to each other (e.g., a parent process refers to child process) resulting in *model trees*. In contrast, *process variants* are part of a *process model family*, and are derived from a *generic process model* through a sequence of adaptations. This approach is often referred to as *model customization* or *configuration* [3, 4]. Like in SE, tool support is essential as a refactoring applied to one model might require changes in other models as well. As a contribution towards more effective tool support, we use formal notions to reason about the validity of the proposed refactorings.

In this paper we focus on refactoring techniques for control flow. For each proposed refactoring we describe its intent, give examples for its applicability and use (similar to *code smells* in SE [8]), and discuss its effects in respect to process model quality metrics (e.g., measuring control flow complexity) [12, 11]. None of the proposed techniques is dependent on a specific process meta-model.

Section 2 provides background information. Section 3 then gives an introduction into refactoring of process models and presents refactoring techniques for process composition. Section 4 suggests a refactoring to effectively deal with generalization and Section 5 introduces advanced refactorings considering process history data. Related work is discussed in Section 6. Finally, Section 7 concludes with a summary and an outlook.

## 2 Background Information

In this section we describe basic concepts and notions used in this paper.

### 2.1 Basic Concepts and Notions

A PAIS is a specific type of information system which provides process support functions and allows for the separation of process logic and application code. At *build-time* process logic has to be explicitly defined in a *process schema*, while at *run-time* the PAIS orchestrates processes according to their defined logic.

For each business process to be supported, a *process type* represented by a *process schema*  $S$  has to be defined. In the following, a process schema corresponds to a directed graph, which comprises a set of *nodes* – representing *activities* or *control connectors* (e.g. XOR-Split, AND-Join) – and a set of *control edges* between them. The latter specify precedence relations. Further, activities can be atomic or complex. While an *atomic activity* is associated with an invocable application service, a *complex activity* contains a sub process or, more precisely, a reference to a (sub) process schema  $S'$ . This allows for the hierarchical decomposition of schemes resulting in a *process model tree* (cf. Fig. 1a). Generally, different schemes  $S_1 \dots S_n$  may refer to a (sub) process schema  $S'$ . Fig. 1a shows a schema  $S$  modeled in BPMN notation [13] consisting of seven nodes. Thereby, A, B and D are atomic activities, C and E are complex activities referring to (sub) process schemes  $S1$  and  $S2$  respectively, and XOR-split and XOR-Join are control connectors.  $S2$  itself refers to schema  $S3$  resulting in a process model tree with *depth* three.

Process schemes can either be created from scratch or through configuration, i.e., customization of a *generic process model* (e.g., a reference model). From such a generic model several *process variants* (each with its schema) may be derived based on a restricted set of change operations [5, 14]. Thereby, for a given variant we denote the set of change operations needed to transform the generic model into the variant as *bias*. Usually, the aim is to minimize the number of operations required in this context. The total set of all variants derived from a generic process model is called *process model family*. Fig. 1b shows a generic process schema  $S_G$  and four variants  $V_1, \dots, V_4$  derived from it. For example, the transformation of  $S_G$  to  $V_1$  requires deletion of Activity  $G$ .

Most refactoring techniques are not only applicable to activities, but also to sub process graphs with single entry and exit nodes (also denoted as *hammocks* [15]). We use the term *process fragment* as generalizing concept for all these granularities; e.g., in Fig. 1b the sub-graph of schema  $S$  containing Activities B, C, and D and the two control connectors constitutes a hammock. Based on schema  $S$ , at run-time new *process instances* can be created and executed, which is reflected by the instance *trace*.

**Definition 1 (Execution Trace).** *Let  $\mathcal{PS}$  be the set of all process schemes and let  $\mathcal{A}$  be the total set of activities (or more precisely activity labels) based on which schemes  $S \in \mathcal{PS}$  are specified (without loss of generality we assume unique*

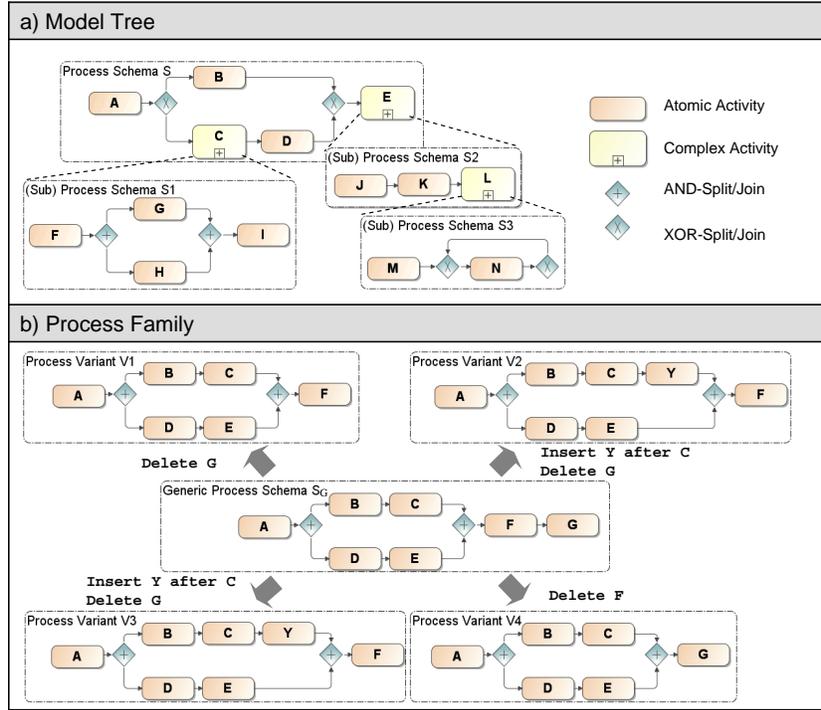


Fig. 1. Core Concepts

labelling of activities). Let further  $Q_S$  denote the set of all possible execution traces producible on schema  $S \in \mathcal{PS}$ . A trace  $\sigma \in Q_S$  is then given by  $\sigma = \langle a_1, \dots, a_k \rangle$  (with  $a_i \in \mathcal{A}$ ) where the temporal order of  $a_i$  in  $\sigma$  reflects the order in which activities  $a_i$  were completed over  $S$ .

For example,  $\sigma_1 = \langle A, B, D, C, E, F \rangle$  and  $\sigma_2 = \langle A, B, C, D, E, F \rangle$  both constitute traces producible by process variant  $V_1$  in Fig. 1b.

Schemes  $S$  and  $S'$  are called *trace equivalent* if and only if the same set of execution traces can be produced based on  $S$  and  $S'$ .

**Definition 2 (Trace Equivalence).** Two process schemes  $S$  and  $S'$  are trace equivalent iff  $Q_S = Q_{S'}$ .

To determine whether two (hierarchically) composed process schemes  $S$  and  $S'$  are trace equivalent, the respective process model trees need to be expanded. For this, each complex activity needs to be replaced by the (sub) process schema it refers to. Consequently, the trace of an activity does not contain the complex activity directly, but the trace of the associated sub process. A possible execution trace for schema  $S$  in Fig. 1a is  $\sigma_1 = \langle A, B, J, K, M, N \rangle$ .

## 2.2 Quality Metrics for Business Process Models

We describe selected *quality metrics* for process models, and use them in the following to discuss effects of the proposed refactorings (cf. Fig. 2). In SE, metrics have been used since the 60s to measure software quality. Main purpose is to improve software design, resulting in better understandable and maintainable code [16, 17]. BPM research has recently started to adopt quality metrics to specific needs of process modeling [10, 11, 18, 19] and to empirically validate these metrics [10, 12]. We use these metrics and complement them with further ones measuring cost of process change in terms of *change distance* [20] (cf. Fig. 2).

Quality Metrics for Business Processes		
Let $S = (N, E)$ be a process model with $N$ denoting the set of nodes and $E$ the set of edges.		
Metric	Description	Metrics calculated for Fig. 1
<b>Size</b> [11, 18]	$Size(S) =  N $ measures the number of nodes in process schema $S$	$Size(S) = 7$
<b>Process Depth</b> [18]	$Levels(S)$ = number of process levels of the model tree with $S$ as root	$Levels(S) = 3$
<b>Control-Flow Complexity</b> [10]	Let ANDSplits, XORSplits and ORSplits denote the node sets of $S$ comprising respective split nodes. Let further $\delta^+(n)$ denote the number of direct successors of node $n$ (number of control edges outgoing from $n$ ). Then: $CFC(S) =  ANDSplits  + \sum_{c \in XORSplits} \delta^+(c) + \sum_{c \in ORSplits} (2^{\delta^+(c)} - 1)$ is the sum over all connectors weighted by their potential combinations of states after the split	$CFC(S) = 2$
<b>Change Distance</b> [20]	$Dist(S_1, S_2)$ : Minimal number of high-level change operations (e.g., MOVE Activity) needed to transform schema $S_1$ to schema $S_2$	$Dist(S_G, V_1) = 1$

Fig. 2. Selected Quality Metrics for Process Models

Though refactoring decisions have to be made by designers, quality metrics help them to identify quality problems or deviations from the norm. Refactoring techniques can be applied to improve the quality of a process model. Its effects on model quality, in turn, can be measured based on respective quality metrics. As the application of a particular refactoring may affect several schemes it is not sufficient to look only at the quality metrics of a single schema in isolation, but to apply metrics to the entire collection of schemes as well. For this purpose we introduce functions *sum* and *avg*, which we will use later on for comparing process models before and after refactorings.

$$sum : 2^{\mathcal{PS}} \times Metrics \times Params \mapsto \mathbb{N}_0 \text{ with } sum(mset, m, p) := \sum_{S \in mset} m(S, p)$$

$$avg : 2^{\mathcal{PS}} \times Metrics \times Params \mapsto \mathbb{R}_0^+ \text{ with } avg(mset, m, p) := \frac{sum(mset, m, p)}{|mset|}$$

For example, the total change distance for the process family depicted in Fig. 1b is  $sum(\{V_1, \dots, V_4\}, Dist, S_G) = 6$ , while the average change distance is 1.5.

### 3 Refactorings for Process Composition

This paper describes 11 refactoring techniques which allow process designers to improve the quality of process models (cf. Fig. 3). In our context refactorings constitute model transformations based on well-defined change patterns [14, 21], which are behaviour-preserving if certain pre- and postconditions are met. We use *trace equivalence* (cf. Def. 2) as formal notion for most refactorings to ensure that no errors are introduced through their application. If for a model tree with root  $S_i$ , which shall be refactored, the same trace sets can be produced before and after the respective refactoring process behaviour will be preserved.

We divide our refactorings into *basic* ones, which can be applied to a single schema, and *composed refactorings* applicable to a collection of inter-related process schemes. Basic refactorings transform a schema  $S$  into a new schema  $S'$  by applying a refactoring operation  $\text{op}$ . This transformation might also imply changes of a model tree if, for example, a fragment is extracted from a process model and replaced by a reference to a sub process. Composed refactorings, in turn, refer to a collection of process schemes  $S_1 \dots S_n$  and apply basic refactorings to them if they meet the respective pre-conditions.

For each of the proposed refactorings we describe its intent, give examples, provide a description of the refactoring operation (with pre- and postconditions) and its implementation, and describe their effects on selected quality metrics. We organize our refactorings into three groups. The first one contains *refactorings for process composition* and is introduced in this section. The second one suggests a refactoring to effectively deal with *generalization* (cf. Section 4). The third group describes model *refactorings considering process history data* (cf. Section 5).

First we describe 8 **refactorings for process composition**. Refactoring *RF1 (Rename Activity)* can be applied when the name of an activity is not intention revealing and *RF2 (Rename Process Schema)* allows altering the name of a schema. Using *RF3 (Substitute Process Fragment)* process designers can substitute a fragment within a schema by another one which is simpler in structure, but has the same behaviour. *RF4 (Extract Process Fragment)* allows extracting a process fragment into a sub process to remove model redundancies, to foster reuse, and to reduce the size of a schema. Applying *RF5 (Replace Process Fragment by Reference)* a process fragment can be replaced by a complex activity referring to a (sub) process schema containing the respective fragment. *RF6 (Inline Process Fragment)* can be applied to collapse the hierarchy by inlining a fragment. *RF7 (Re-Label Collection)* is a composed refactoring, which supports re-labelling of certain activities within an entire process collection. Finally, *RF8 (Remove Redundancies)* allows for combined use of RF4 and RF5 to remove redundant fragments from multiple schemes in a model collection at once.

**RF1 (Rename Activity).** RF1 allows altering the name of an activity  $x$  to  $y$  if  $x$  is not intention revealing. RF1 is comparable to the *Rename Method* refactoring in SE [8]. Renaming an activity does not alter the behaviour of the schema  $S$  as only labels are changed. However, the notion of trace equivalence is not suitable in this context. Instead, we use a correctness notion based on the *Replace Process Fragment* change pattern [14, 21]. For each trace  $\sigma$  produced on

Refactoring Catalogue		
Name	Refactoring Operation	Short description of refactoring
Refactorings for Process Composition		
RF1: Rename Activity	<code>renameActivity(s, x, y)</code>	Changes the name of an activity from $x$ to $y$ in schema $S$ <i>Pre-Condition:</i> No activity from $S$ is labelled with $y$
RF2: Rename Process Schema	<code>renameSchema(s, s')</code>	Renames schema from $S$ to $S'$ and updates all references to $S$ <i>Pre-condition:</i> There exists no schema with label $S'$ in the repository
RF3: Substitute Process Fragment	<code>substituteFragment(s, g, g')</code>	Substitutes sub-graph $G$ in $S$ by sub-graph $G'$ <i>Pre-condition:</i> $G$ and $G'$ constitute hammocks and are trace equivalent
RF4: Extract Process Fragment	<code>extractFragment(s, g, x, s')</code>	Extracts sub-graph $G$ in $S$ and substitutes it with complex activity $x$ referring to $S'$ <i>Pre-condition:</i> There is no activity with label $x$ in $S$ ; $G$ is a hammock
RF5: Replace Process Fragment by Reference	<code>replaceFragment(s, g, x, s')</code>	Substitutes sub-graph $G$ in $S$ by complex activity $x$ referring to schema $S'$ <i>Pre-condition:</i> No activity from $S$ is labelled with $x$ ; $G$ is a hammock, and $G$ and $S'$ are trace equivalent
RF6: Inline Process Fragment	<code>inlineFragment(s, x)</code>	Inlines the sub process schema activity $x$ refers to in $S$ and deletes the respective sub process schema, if it is unused after the refactoring <i>Pre-condition:</i> Activity $x$ is a complex activity
RF7: Re-label Collection	<code>relabelCollection(c, x, y)</code>	Applies RF1 to every schema $S_1, \dots, S_n$ in model collection $C$ where $x \in S_i$
RF8: Remove Redundancies	<code>removeRedundancies(c, g, x, s')</code>	Applies RF4 to the first schema $S_i$ in model collection $C$ meeting the pre-conditions and RF5 to all other schemes
Refactoring for Generalization		
RF9: Generalize Variant Changes	<code>generalizeVariantChanges(s_g, VariantSet, ChangeSet)</code>	Generalizes variant changes by applying changes from <code>ChangeSet</code> to generic model $S\_G$ and by re-linking all variants from <code>VariantSet</code> to the new generic model $S\_G'$ (i.e., their biases are re-calculated with respect to $S\_G'$ )
Refactorings Considering Process History Data		
RF10: Remove Unused Branch	<code>removeUnusedBranch(s, g)</code>	Removes an unused branch $G$ from schema $S$ . <i>Pre-condition:</i> $G$ constitutes a branch within a conditional branching, which was not entered when executing instances of $S$ .
RF11: Pull Up Instance Change	<code>pullUpInstChange(s, InstSet, ChangeSet)</code>	Pulls frequent changes that happened at the process instance level up to the type level schema $S$ . Change are described in terms of a set of applied change operations.

Fig. 3. Refactoring Catalogue

schema  $S$  with an entry of  $x$  there exists a respective trace  $\mu$  on  $S'$  which is identical to  $\sigma$ , except that for every  $x$  in  $\sigma$  a  $y$  in  $\mu$  can be found at the same position. Applying RF1 does not have effects on the quality metrics described in Fig. 2. However, names which reveal the intention of process designers more clearly improve understandability of the model and consequently result in decreased cost of change and reduced errors [22].

**RF2 (Rename Process Schema.)** RF2 allows designers to rename a schema  $S$  to  $S'$ . A similar refactoring in SE is *Rename Class* [23]. To guarantee that RF2 does not alter process behaviour, all references to  $S$  are updated. Obviously, equivalence can be used as formal notion for RF2 ensuring that the behaviour of the model collection remains unchanged. Like RF1 this refactoring does not affect quality metrics, but improves model clarity.

**RF3 (Substitute Process Fragment).** RF3 allows substituting a fragment by another one with simpler structure, but the same behaviour. Applying RF3 requires both fragments to contain activities with the same labelling. The *Substitute Algorithm* refactoring [8] known from SE is comparable to RF3. Scenarios in which RF3 is useful include unnecessarily complex parallel branchings (cf. Fig. 4a) or unneeded control edges due to transitive relations. RF3 can be implemented based on change pattern *Replace Process Fragment* [14, 21]. As formal criterion trace equivalence can be used (cf. Def. 2). Substituting a fragment by a simpler one allows designers to improve model quality along several dimensions: by removing unnecessary parallel branchings and edges not only model clarity is increased, but also size and control-flow complexity (CFC) are decreased.

**RF4 (Extract Process Fragment).** RF4 can be used to extract a process fragment from a schema  $S$ , e.g., to eliminate redundant fragments or to reduce size of  $S$ . The fragment to be extracted must constitute a hammock. The intent of RF4 is similar to *Extract Method* [8] as known from SE. It results in the creation of a new (sub) process schema  $S1$  containing the respective fragment. In addition, the fragment is replaced by a complex activity referring to  $S1$ . As formal notion for reasoning about behaviour preservation trace equivalence is used again. RF4 can be implemented based on change pattern *Extract Process Fragment* [14, 21]. In many cases extracting parts of a schema results in a reduced CFC (cf. Fig. 5). Similarly, in SE the *Extract Method* refactoring is suggested as remedy for high cyclomatic complexity [24]. RF4 can also be used to reduce the size of large schemes and the overall number of nodes in the process repository by removing redundancies. Further, removing redundancies also reduces cost of future process changes as the same changes do not have to be performed at multiple places.

**RF5 (Replace Process Fragment by Reference).** RF5 is used to replace a process fragment by a complex activity referring to a trace equivalent (sub) process schema. RF5 is often used in combination with RF4. It can be implemented based on change pattern *Replace Process Fragment* [14, 21]. Regarding quality metrics similar considerations hold than for RF4.

**RF6 (Inline Process Fragment).** RF6 can be used to collapse the hierarchy of a model by inlining the process fragment, e.g., if it is not justifying its

induced overhead. Similarly, in SE *Inline Method* [8] allows programmers to inline the body of a method. By inlining a fragment  $S1$  into  $S$  the complex activity referring to  $S1$  is substituted by the fragment corresponding to  $S1$ . Again trace equivalence can be used as a formal notion. RF6 can be implemented based on the *Inline Process Fragment* change pattern [14, 21]. In particular, RF6 allows designers to collapse the hierarchy of a process model tree resulting in a decrease of levels. Note that metrics Size and CFC might increase when applying RF6.

**RF7 (Re-Label Collection).** RF7 is a composed refactoring based on RF1 and can be used for re-labelling a particular activity in all schemes of a model collection. For this, RF1 is applied to all schemes containing the activities to be re-labelled.

**RF8 (Remove Redundancies).** RF8 is a composed refactoring based on RF4 and RF5. It can be applied to a collection of schemes  $S_1 \dots S_n$  to remove redundancies. For this, RF4 is applied to one of these schemes to extract the redundant fragment. To all other schemes, RF5 is applied for replacing the respective fragment by a reference to the (sub) process schema created before.

**Example.** *Fig. 4 shows the combined usage of the basic refactorings described so far. For instance, for schema  $S$  activity  $A$  is renamed to  $A'$  using RF1. RF2 is used to rename schema  $S3$  to  $S3'$ . As process schemes  $S$  and  $S1$  contain complex Activity  $M$  referring to  $S3$  the references in  $M$  need to be updated to  $S3'$ . A further refactoring option is given by schemes  $S$ ,  $S1$  and  $S2$ , all containing a process fragment with same behaviour. However, the fragment in schema  $S$  has a more complex structure than the ones in  $S1$  and  $S2$ . First, RF3 is used to replace the fragment in  $S$  with the one of  $S1$  or  $S2$ . Next, RF4 is applied to either  $S$ ,  $S1$  or  $S2$  to extract the redundant process fragment to a (sub) process schema  $S5$ . Finally, RF5 is applied to the two other schemes to replace the respective fragment by a reference to  $S5$ . Instead of RF4 and RF5 the composed refactoring RF8 could be used alternatively. Schema  $S4$  only consists of a single activity and is therefore inlined in schema  $S2$  using RF6.*

**Effects on Quality Metrics.** Usually, refactorings are not applied in isolation, but in combination with other refactorings and with respect to a collection of models. Consequently, refactoring has an impact on the collection of process models. For example, in Fig. 4 the combined use of refactorings RF3, RF4, RF5 and RF6 reduces the total number of nodes in the given model collection from 34 to 20 and also decreases average CFC of the schemes by factor 1:4 (cf. Fig. 5). Note that in all cases no changes of the model behaviour have been performed. In particular, application of RF3 allows for the removal of two unnecessary connector nodes, reducing size by two and CFC by one; RF4 and RF5 remove existing redundancies leading to an additional saving of 11 nodes. Finally, RF6 reduces size by one.

As illustrated in Fig. 5 the proposed refactorings do not only result in smaller and less complex models, but also reduce costs of future changes by removing redundancies. For example, assume that activity  $D$  in Fig. 4 shall be replaced by a sequence consisting of activities  $D1$  and  $D2$ . Without the described refactoring this change would require to modify schemes  $S$ ,  $S1$  and  $S2$  by ap-

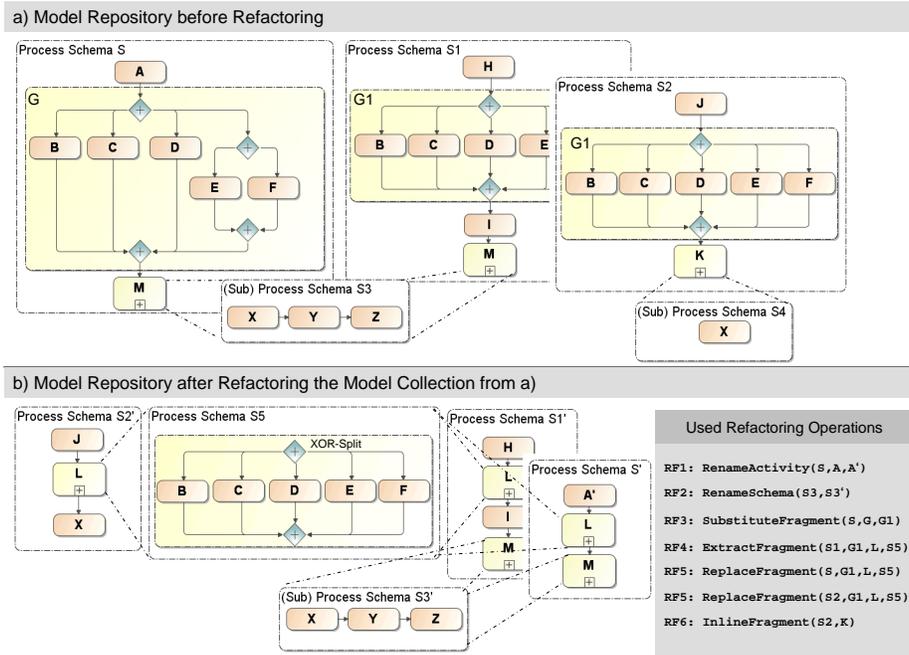


Fig. 4. Refactorings for Process Composition

plying three change operations to each of these schemes resulting in a total change distance of 9. In contrast, considering the refactoring only schema  $S5$  needs to be modified ( $\text{Delete}(S5, G)$ ,  $\text{SerialInsert}(S5, D1, \text{XOR-Split})$  and  $\text{SerialInsert}(S5, D2, D1)$ ) reducing the total change distance by 66,67 % to 3. Removing redundancies does not only result in smaller change distance, but also reduces the risk of introducing inconsistencies or errors. Finally, the exact change distance depends on the intended change and the used meta-model.

The scenario depicted in Fig. 4 constitutes a toy example. For more realistic scenarios refactoring effects become greater as our case studies in domains like healthcare and automotive engineering revealed. When elaborating 30 process models of a Women’s hospital, for example, we detected redundancies in more than 60% of the models [25]. Particularly, larger models with more than 20 activities often contained redundant process fragments (e.g., for making appointments with medical units or for exchanging medical reports). As we learned, these redundancies could be abolished using the proposed refactorings. Due to lack of space we omit further details.

## 4 Refactoring for Generalization

Another challenge is to manage the process variants belonging to a process family (cf. Fig. 1b). Usually, respective variants are derived from a generic schema  $S_G$

Before Refactoring (Fig. 4a)				After Refactoring (Fig. 4b)			
	Size	CFC	Levels		Size	CFC	Levels
S	11	2	2	S	3	0	2
S1	10	1	2	S1	4	0	2
S2	9	1	2	S2	3	0	2
S3	3	0	-	S3	3	0	-
S4	1	0	-	S4			
S5				S5	7	1	-
Sum	34	4		Sum	20	1	
Avg.	6.8	0.8		Avg.	4	0.2	

**Fig. 5.** Effects on Quality Metrics (with respect to Fig. 4)

by applying a set of change operations to it. In general, configuration of new variants and adaptation of existing variants can be done most effectively when the average change distance (cf. Section 2) between generic schema  $S_G$  and its variants  $V_1, \dots, V_n$  is minimal (i.e., the average number of change operations needed to transform  $S_G$  to  $V_i$  is minimal). However, to keep the average change distance small, continuous efforts have to be made to evolve the generic model over time. Otherwise, more and more redundant changes have to be performed to different variants to keep them aligned with the real-world processes. Though respective variants are often similar, slight differences make refactorings RF4 and RF5 inapplicable in many situations. Therefore, an additional refactoring technique is needed, which supports designers in maintaining generic models.

**RF9 (Generalize Variant Changes).** RF 9 allows designers to pull changes, which are common to several variants, up to the generic model (similar to *Pull Up Method* and *Push Down Method* in SE [8]). This allows removing redundancies and decreasing cost of future changes. As example consider Fig. 1b, which shows a generic model  $S_G$  and four variants  $V_1, \dots, V_4$  derived from it. Analysis of  $S_G$  and its variants shows that Activity G has been deleted for 3 of the 4 variants. Refactoring `GeneralizeVariantChanges( $S_G, \{V_1, \dots, V_4\}, \{Delete(G)\}$ )` can be applied to generalize the respective change by pulling the deletion of G up to the generic model  $S_G$  (not shown in Fig. 1b). As Activity G is deleted from the generic model, G needs to be inserted in variant  $V_4$  to keep the behaviour of variant  $V_4$  unchanged. This results in a reduction of the total change distance from 6 to 4 and a decrease of the average change distance from 1.5 to 1.0.

In case studies we conducted in healthcare, for example, we identified 10 variants for medical order handling with similar behaviour [25]. Though respective variants were similar, slight differences existed and redundant fragments could not be extracted to (sub) processes. However, by applying RF9 redundancies could be significantly reduced resulting in easier to configure and better maintainable variants.

The implementation of RF9 necessitates a sophisticated framework for coping with generic schemes and variants derived from them. First, advanced techniques for analyzing/mining process variants and for identifying variant changes to be pulled up to the generic model are needed. In MinADEPT [26], for example, a

generic model  $S'_G$  can be derived from a set of variants `VariantSet` such that the change distance between  $S'_G$  and the variants becomes minimal. Second, when applying RF9 the change operations in `ChangeSet` are applied to  $S_G$  resulting in a new version of the generic model  $S'_G$ . All variants in `VariantSet` need to be re-linked from  $S_G$  to  $S'_G$  and for each variant  $V_i \in \text{VariantSet}$  its bias must be re-calculated in respect to  $S'_G$  [27]. Third, effective techniques are needed for internally representing generic models, its variants and related biases (e.g., [20]). Note that RF9 does not alter variant behaviour. Applying the updated bias of a variant  $V_i$  to  $S'_G$  results in the same variant-specific schema as applying the old bias to  $S_G$ . Thus trace equivalence can be used as formal notion.

## 5 Refactorings Considering Process History Data

This section describes refactoring techniques, which become applicable when process models are executed by PAIS and historic data on process instances is available in execution and/or change logs [28, 20]. These logs can be analyzed and mined to discover potential refactoring options. In this context *RF10 (Remove Unused Branches)* allows process designers to remove unused paths from a process model and *RF11 (Pull Up Instance Change)* enables generalization of frequent instance changes by pulling them up to the process type level. Several process mining methods for discovering such situations already exist [28, 26]. We therefore do not look at mining techniques, but use them for realizing refactorings based on historical data.

**RF10 (Remove Unused Branches).** RF10 allows designers to remove unused process fragments from a schema  $S$ . It can be implemented based on change pattern *Delete Process Fragment* [14, 21] and on standard process mining techniques. Note that trace equivalence is not suitable as formal basis since the behaviour *producible* on the respective process schema is always altered by RF10. Therefore we use the notion of *compliance* in connection with RF10. It ensures that the *observed* behaviour remains unchanged. More precisely, RF10 can be applied to schema  $S$  if the related execution log  $L$  is compliant with the resulting schema  $S'$ ; i.e., each instance trace  $\sigma \in L$ , which was produced on  $S$ , is re-producible on  $S'$ . Obviously, compliance can be ensured when removing unused execution paths from a schema. RF10 is not applied automatically, but the designer has to ensure that the misalignment between model and log was not caused by design errors or an execution log not covering all relevant traces. Depending on the concrete application scenario the time window for which events from the log are considered can be narrowed. Applying RF10 decreases both model size and control flow complexity. Fig. 6a shows a schema  $S$  with its execution log comprising the traces of completed instances. When mining this log it will be discovered that the path with activities E and F was never executed. RF10 could then be applied to remove the unused fragment. This reduces size of  $S$  from nine to seven and CFC from three to two. After removing activities E and F all instances in the log are still compliant with schema  $S'$ .

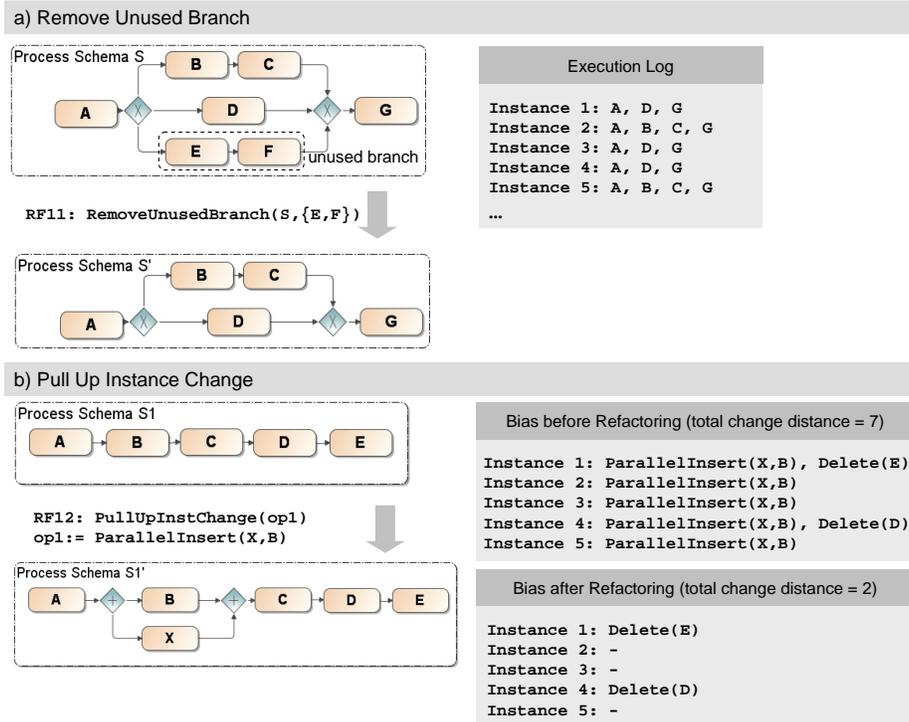


Fig. 6. Remove Unused Branch and Pull Up Instance Change Refactorings

**RF11 (Pull Up Instance Change).** RF11 can be used to generalize frequently occurring instance changes by pulling them up to the process type level (similar to RF9 where variant changes are generalized). Like for RF9 the overall goal is to reduce average and total change distance between type schema and instance-specific schemes; e.g., to learn from instance changes and to reduce the need for adapting future instances [27]. The implementation of RF11 is similar to RF9. In contrast to RF9, however, trace equivalence can not be used to ensure that no errors are introduced when applying RF11. By pulling changes from the instance level to the type level behaviour producible on the respective schema is always altered. Therefore, compliance is used as a formal notion like in RF10.

Fig. 6b shows a process schema  $S1$  and for each process instance  $I_1, \dots, I_5$  its deviation from schema  $S1$ . Activity  $X$  was inserted parallel to  $B$  for each of these instances. For  $I_1$  activity  $E$  was additionally deleted and for  $I_4$  activity  $D$  was deleted. To pull up the insertion of activity  $X$  (which is common to all instances) to the type level and to reduce the need for future instance adaptations RF11 could be applied. Using RF11 reduces the total change distance from  $\text{sum}(\{I_1, \dots, I_5\}, \text{Dist}, S1) = 7$  to  $\text{sum}(\{I_1, \dots, I_5\}, \text{Dist}, S1') = 2$ .

## 6 Related Work

Refactoring techniques for improving software design were first proposed by Opdyke [7]. He suggested a set of refactorings for C++ which are semantic preserving if certain preconditions are met. The first notable refactoring tool has been the Refactoring Browser [23] for Smalltalk, which automatically performs the refactorings proposed by Opdyke plus some additional techniques [29]. As all refactorings provided by this tool constitute behaviour-preserving transformations it is ensured that no errors or information losses are introduced. Tool support for languages like C++ and Java recently emerged. The provided refactorings are usually not provably behaviour-preserving. Therefore, refactorings need to be backed up by automated regression tests to detect behavioural changes in the software and to avoid errors [8].

Similar to program refactorings, model refactorings constitute transformations, which are behaviour-preserving if certain pre-/post-conditions are met. Existing approaches focus on UML model transformations [30], while the application of refactoring techniques to process models has not yet been elaborated in detail. In [31] first refactoring techniques for event-driven process chains (EPCs) are described. By introducing further control flow constructs EPCs can be simplified. In contrast, our refactorings are meta-model independent and do not require any advanced control-flow constructs. Refactoring techniques are also discussed in [32] in the context of model merging. The proposed transformations aim at improved process design, but are not necessarily behaviour-preserving.

This paper complements existing work dealing with *process redesign* [33] or *process adaptation* [5]. Both refactoring and *process redesign* [33] may require model transformations. However, scope of process redesign is much broader and goes beyond structural adaptations. Redesign is primarily business driven and aims to improve one or more performance dimensions of a process (e.g., time, quality, costs or flexibility) [33]. Therefore, process redesign often affects external quality of a PAIS and its results are visible to the customer. In contrast, refactoring techniques primarily impact the internal quality of the PAIS, ensure conceptual integrity, and foster maintainability. Similar to refactorings, *process adaptations* [5] refer to structural changes of a process schema (e.g., using change patterns) [14, 21, 5]. In contrast to refactorings, process adaptations are usually affecting the behaviour of a process model. We build upon existing research in this area and extend it to be applicable for process model refactorings.

Existing BPM tools only provide limited refactoring support. Renaming of activities and process schemes is supported by most tools (e.g., ARIS). However, more advanced refactoring support is missing.

## 7 Summary and Outlook

We proposed 11 refactoring operations specifically suited for the refactoring of large process repositories. These techniques allow process designers to better deal with model complexity and to make process models easier to change, less

error-prone and better understandable. With the increasing adoption of PAIS and the emergence of large process repositories systematic support for model management is getting increasingly important. We are currently working on a reference implementation of a tool for refactoring process models. In particular, we want to support users in both identifying refactoring options and in applying behaviour-preserving or compliance-ensuring refactorings. We further plan to integrate this with our previous work on change patterns [14, 21], model evolution [34], and process change mining [26] to provide integrated support for the management of process models throughout the entire process life cycle.

**Acknowledgements.** We would like to thank Werner Wild for his valuable feedback regarding this paper.

## References

1. Weske, M.: Business Process Management: Concepts, Methods, Technology. Springer (2007)
2. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
3. Rosemann, M., van der Aalst, W.: A Configurable Reference Modelling Language. Information Systems (2005)
4. Rosa, M.L., Lux, J., Seidel, S., Dumas, M., ter Hofstede, A.: Questionnaire-driven Configuration of Reference Process Models. In: Proc. CAiSE'07. (2007) 424–438
5. Reichert, M., Dadam, P.: ADEPT<sub>flex</sub> – Supporting Dynamic Changes of Workflows Without Losing Control. JIIS **10** (1998) 93–129
6. Parnas, D.: Software Aging. In: Proc: ICSE '94. (1994) 279–287
7. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, Univ. of Illinois (1992)
8. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley (2000)
9. Beck, K.: Extreme Programming Explained. Addison Wesley (2000)
10. Cardoso, J.: Process Control-Flow Complexity Metrics: An Empirical Validation. In: Proc. IEEE SCC'06. (2006) 167–173
11. Vanderfeesten, I., Cardoso, J., Mendling, J., Reijers, H., van der Aalst, W.: Quality Metrics for Business Process Models. In: 2007 BPM & Workflow Handbook. (2007)
12. Mendling, J.: Detection and Prediction of Errors in EPC Business Process Models. PhD thesis, Vienna Univ. of Economics and Business Administration (2007)
13. Object Management Group: Business Process Modeling Notation Specification. (2006)
14. Weber, B., Rinderle, S., Reichert, M.: Change patterns and change support features in process-aware information systems. In: Proc. CAiSE'07. (2007) 574–588
15. Zhang, F., D'Hollander, E.: Using Hammock Graphs to Structure Programs. IEEE ToSE **30** (2004) 231–245
16. McCabe, T.: A Complexity Measure. IEEE ToSE **2** (1976) 308–320
17. Yourdon, E., Constantine, L.: Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall, Yourdon Press (1979)
18. Nissen, M.: Redesigning Reengineering through Measurement-Driven Inference. MIS Quarterly **22** (1998) 509–534
19. Reijers, H., Vanderfeesten, I.: Cohesion and Coupling Metrics for Workflow Process Design. In: Proc. BPM'04. (2004) 290–305

20. Rinderle, S., Reichert, M., Jurisch, M., Kreher, U.: On Representing, Purging, and Utilizing Change Logs in Process Management Systems. In: Proc. BPM'06. (2006) 241–256
21. Weber, B., Rinderle, S., Reichert, M.: Change Support in Process-Aware Information Systems - A Pattern-Based Analysis. Technical Report TR-CTIT-07-76, University of Twente (2007)
22. Becker, J., Rosemann, M., von Uthemann, C.: Guidelines of Business Process Modeling. In: BPM'00. (2000) 30–49
23. Brant, J., Roberts, D.: Refactoring Browser: [st-www.cs.uiuc.edu/users/brant/refactoringbrowser/](http://st-www.cs.uiuc.edu/users/brant/refactoringbrowser/) (1999)
24. Glover, A.: Refactoring with Code Metrics, [www.ibm.com/developerworks/java/library/j-cq05306/](http://www.ibm.com/developerworks/java/library/j-cq05306/) (2006)
25. Reichert, M., Dadam, P., Schultheiss, B., Konyen, I.: Modeling and analysis of healthcare processes in a woman's hospital. project reports no. dbis-27, dbis-28, dbis-29, dbis-16, dbis-15, dbis-14, dbis-7, dbis-6, dbis-5 (1996-1997)
26. Günther, C., Rinderle, S., Reichert, M., van der Aalst, W.: Change Mining in Adaptive Process Management Systems. In: Proc. CoopIS'06. (2006) 309–326
27. Rinderle, S., Weber, B., Reichert, M., Wild, W.: Integrating Process Learning and Process Evolution - A Semantics Based Approach. In: Proc. BPM 2005. (2005) 252–267
28. Van der Aalst, W., van Dongen, B., Herbst, J.: Workflow Mining: a Survey of Issues and Approaches. *Data and Knowledge Engineering* (2003) 237–267
29. Roberts, D., Brant, J., Johnson, R.: A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems* (1997) 253–263
30. Sunye, G., Pollet, D., Traon, Y.L., Jezequel, J.: Refactoring UML Models. In: UML'01. (2001) 134–148
31. Fettke, P., Loos, P.: Refactoring von Ereignisgesteuerten Prozessketten. In: EPK'02. (2002) 37–49
32. Küster, J., Koehler, J., Ryndina, K.: Improving Business Process Models with Reference Models in Business-Driven Development. In: BPM'06 Workshops. (2006) 35–44
33. Reijers, H.: *Design and Control of Workflow Processes: Business Process Management for the Service Industry*. Springer (2003)
34. Rinderle, S., Reichert, M., Dadam, P.: Correctness Criteria for Dynamic Changes in Workflow Systems – A Survey. *DKE* **50** (2004) 9–34