Merging Business Process Models (Extended Version)

Marcello La Rosa[1], Marlon Dumas[2], Reina Käärik[2], and Remco Dijkman[3]

[1] Queensland University of Technology, Australia
m.larosa@qut.edu.au
[2] University of Tartu, Estonia
{marlon.dumas,reinak}@ut.ee
[3] Eindhoven University of Technology, The Netherlands
r.m.dijkman@tue.nl

Abstract. This paper addresses the following problem: given two or more business process models, create a process model that is the union of the process models given as input. In other words, the behavior of the produced process model should encompass that of the input models. The paper describes an algorithm that produces a single configurable process model from an arbitrary collection of process models. The algorithm works by extracting the common parts of the input process models, creating a single copy of them, and appending the differences as branches of configurable connectors. This way, the merged process model is kept as small as possible, while still capturing all the behavior of the input models. Moreover, analysts are able to trace back from which original model(s) does a given element in the merged model come from. The algorithm has been prototyped and tested against process models taken from several application domains.

1 Introduction

In the context of company mergers and restructurings, it often occurs that multiple alternative processes, previously belonging to different companies or units, need to be consolidated into a single one in order to eliminate redundancies and to create synergies. To this end, teams of business analysts need to manually compare similar process models so as to identify commonalities and differences, and to create integrated process models that can be used to drive the process consolidation effort. This process model merging effort is tedious, time-consuming and error-prone. In one instance reported in this paper, it took a team of five analysts and 600 man-hours to merge 25% of an end-to-end process model.

In this paper, we consider the problem of automatically merging process models under the following requirements:

1. The behavior of the merged model should subsume that of the input models.
2. Given an element in the merged process model, analysts should be able to trace back from which process model(s) the element in question originates.

3. One should be able to derive the input process models from the merged one.

The main contribution of the paper is an algorithm that takes as input a collection of process models and generates a *configurable process model* [10]. A configurable process model is a modeling artifact that captures a family of process models in an integrated manner and that allows analysts to understand what these process models share, what are their differences, and why and how these differences occur. Given a configurable process model, analysts can derive individual members of the underlying family of processes by means of a procedure known as *individualization*. We contend that configurable process models are a suitable output for a process merge algorithm, because they provide a mechanism to fulfill the second and third requirements outlined above.

The proposed algorithm has been implemented and tested on several collections of process models taken from different domains. In addition to providing evidence on the correctness of the algorithm and its implementation, these tests show that the process merge algorithm produces compact models and that it scales up to process models containing hundreds of nodes.

The paper is structured as follows. Section 2 introduces the notion of configurable process model as well as a notion of similarity of process models, which is used as a basis for the merge algorithm. Section 3 presents the process merge algorithm. Section 4 reports on the implementation and evaluation of the algorithm. Finally, Section 5 discusses related work and Section 6 draws conclusions.

2 Preliminaries

This section introduces two basic ingredients of the proposed process merging technique: a notation for configurable process models and a technique to match the elements of a given pair of process models, so that we know which pairs of process model elements should be considered as equivalent when merging.

2.1 Configurable Business Process

There exist many notations to represent business processes, such as Event-driven Process Chains (EPC), UML Activity Diagrams (UML ADs) and the Business Process Modeling Notation (BPMN). In this paper we abstract from any specific notation and represent a business process model as a directed graph with labeled nodes. Specifically, we define a *business process graph* G as a set of pairs of process model nodes—each pair denoting a directed edge. A node n of G is a tuple $(id_G(n), \lambda_G(n), \tau_G(n), \eta_G(n))$ consisting of a unique identifier $id_G(n)$ (of type string), a label $\lambda_G(n)$ (of type string), a type $\tau_G(n)$ indicating the type of node, and a boolean $\eta_G(n)$ indicating whether the node is configurable or not. If there is no confusion, we will drop the subscript G from id_G, λ_G, τ_G and η_G. For a business process graph G, its set of nodes, denoted N_G, is $\bigcup\{\{n_1, n_2\} | (n_1, n_2) \in G\}$.

The available types of nodes depend on the language that is used. For example, the BPMN notation has nodes of type 'activity', 'event' and 'gateway'.

In the rest of this paper we will show examples using the EPC notation, which has three types of nodes: i) 'function' nodes, representing tasks that can be performed in an organization; ii) 'event' nodes, representing pre-conditions that must be satisfied before a function can be performed, or post-conditions that are satisfied after a function has been performed; and iii) 'connector' nodes, which determine the flow of execution of the process. Thus, for a given node n, $\tau_G \in \{\text{“}f\text{”}, \text{“}e\text{”}, \text{“}c\text{”}\}$ where “f” stands for function, “e” stands for event and “c” stands for connector. The label of a node of type “c” indicates the kind of connector. EPCs have three kinds of connectors: AND, XOR and OR. AND connectors either represent that after the connector, the process can continue along multiple parallel paths (AND-split), or that it has to wait for multiple parallel paths in order to be able to continue (AND-join). XOR connectors either represent that after the connector, a choice has to be made about which path to continue on (XOR-split), or that the process has to wait for a single path to be completed in order to be allowed to continue (XOR-join). OR connectors start or wait for multiple paths. Models G_1 and G_2 in Fig. 1 are two example EPCs.

A Configurable EPC (C-EPC) [10] is an EPC where some connectors are marked as configurable. A configurable connector can be configured by reducing its incoming branches (in the case of a join) or its outgoing branches (in the case of a split). The result will be a regular connector with a reduced number of incoming or outgoing branches. In addition, the type of a configurable OR can be restricted to a regular XOR or AND. After being configured, a C-EPC needs to be individualized by removing those branches that have been excluded for each configurable connector. Model CG in Fig. 1 is an example of C-EPC featuring a configurable XOR-split, a configurable XOR-join and a configurable OR-join, while the two models G_1 and G_2 are two possible individualizations of CG. G_1 can be obtained by configuring the three configurable connectors in order to keep all branches labeled “1”, and restricting the OR-join to an AND-join; G_2 can be obtained by configuring the three configurable connectors in order to keep all branches labeled “2” and restricting the OR-join to an XOR-join. Since in both cases only one branch is kept for the two configurable XOR connectors (either the one labeled “1” or the one labeled “2”), these two connectors are removed altogether during individualization. This is why they are not present in G_1 and G_2. For more details on the individualization algorithm, we refer to [10].

According to requirement (2) in Section 1, we need a mechanism to trace back from which variant a given element in the merged model originates. Coming back to the example in Fig. 1, the C-EPC model (CG) can also be seen as the result of merging the two EPCs (G_1 and G_2). The configurable XOR-split immediately below function “Shipment Processing” in CG, has two outgoing edges. One of them originates from G_1 (and we thus label it with identifier “1”) while the second originates from G_2 (identifier “2”). In some cases, an edge in the merged model originates from multiple variants. For example, the edge that emanates from event “Delivery is relevant for shipment” is labeled with both variants (“1” and “2”) since this edge can be found in both original models.

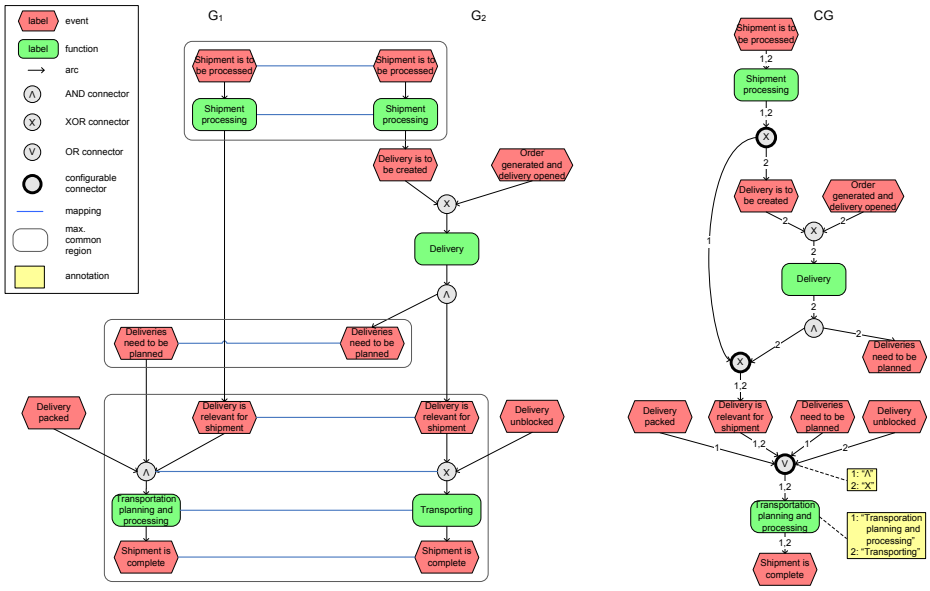

Fig. 1. Two business process models with a mapping, and their merged model.

Also, since nodes in the merged model are obtained by combining nodes from different variants, we need to capture the label of the node in each of its variants. For example, function "Transportation planning and processing" in CG stems from the merger of the function with the same name in G_1, and function "Transporting" in G_2. Accordingly, this function in CG will have an annotation (as shown in the figure), stating that its label in variant 1 is "Transportation planning and processing", while its label in variant 2 is "Transporting". Similarly, the configurable OR connector just above "Transportation planning and processing" in CG stems from two connectors: an AND connector in variant 1, and an XOR connector in variant 2. Thus an annotation will be attached to this node (as shown in the figure) which will record the fact that the label of this connector is "and" in variant 1, and "xor" in variant 2. In addition to providing traceability, these annotations enable us to derive the original process models by configuring the merged one, as per requirement (3) in Section 1. Thus, we define a concept of *Annotated Process Graph*, which attaches additional metadata to edges and nodes in a process graph. Also, we define some auxiliary notations which we will use when comparing two process graphs.

Definition 1 (Annotated Business Process Graph). *Let $\mathcal{I}$ be a set of identifiers to refer to business process models and $\mathcal{L}$ the set of labels that process model nodes can take. An annotated business process graph is a tuple (G, α_G, γ_G) where G is a business process graph, $\alpha_G : G \rightarrow \wp(\mathcal{I})$ is a function that maps each edge in G to a set of process graph identifiers, and $\gamma_G : N_G \rightarrow (\mathcal{I} \times \mathcal{L})$ is a function*

that maps each node in G to a set of pairs where each pair indicates a process graph identifier and the label of the node for that process graph.

Definition 2 (Preset, Postset, Transitive Preset, Transitive Postset). *Let G be a business process graph. For a node $n \in N_G$ we define the preset as $\bullet n = \{m | (m,n) \in G\}$ and the postset as $n\bullet = \{m | (n,m) \in G\}$. We call an element of the preset* predecessor *and an element of the postset* successor*. There is a* path *between two nodes $n \in N_G$ and $m \in N_G$, denoted $n \hookrightarrow m$, if and only if (iff) there exists a sequence of nodes $n_1, \ldots, n_k \in N_G$ with $n = n_1$ and $m = n_k$ such that for all $i \in 1, \ldots, k-1$ holds: $(n_i, n_{i+1}) \in G$. If $n \neq m$ and for all $i \in 2, \ldots, k-1$ holds $\tau(n_i) =$ "c", the path $n \overset{c}{\hookrightarrow} m$ is called a* connector chain*. The set of nodes from which a node $n \in N_G$ is reachable via a connector chain is defined as $\overset{c}{\bullet} n = \{m \in N_G | m \overset{c}{\hookrightarrow} n\}$ and is called the* transitive preset *of n via connector chains. The set of nodes that can be reached via a connector chain from n is $n \overset{c}{\bullet} = \{m \in N_G | n \overset{c}{\hookrightarrow} m\}$ and is called the* transitive postset *of n via connector chains.*

2.2 Matching Business Processes

The aim of matching two process models is to establish the best possible mapping between their nodes. What is considered to be the best mapping depends on a scoring function, called the *matching score*. The matching score we employ is related to the notion of graph edit distance [1]. We use this matching score because it performed well in several empirical studies [12, 2, 3].

Given two graphs and a mapping between their nodes, we compute the matching score in three steps. First, we compute the matching score of each mapping between two nodes as follows. Nodes of different types must not be mapped, splits must be matched with splits and joins must be matched with joins. Thus, a mapping between nodes of different types, or between a split and a join, has a matching score of 0. The matching score of a mapping between two functions or between two events can be measured by the similarity of their labels. To determine the similarity of node labels, we use the standard notion of *string edit distance* [8]. More advanced notions, e.g. [12, 2, 3] can be used as required.

The string edit distance of s and t is the minimal number of atomic string operations needed to transform s into t or vice versa. The atomic string operations are: inserting a character, deleting a character or substituting a character for another. The string edit similarity of s and t, denoted $\mathrm{Sed}(s,t)$ is one minus the string-edit distance. For example, the string edit distance between 'Transportation planning and processing' and 'Transporting' is 26: delete 'ion planning and process'. Consequently, the string edit similarity is $1.0 - \frac{26}{38} \approx 0.32$.

We cannot use the string edit distance to compute the similarity of control nodes, because the labels of control nodes should be interpreted atomically (i.e.: the difference between 'OR' and 'XOR' is the same as the difference between 'OR' and 'AND' even if the number of string operations needed to get from one to the other differs). Instead, we use *context similarity* to determine the similarity of control nodes. Context similarity is computed as the fraction of

nodes in the transitive presets and the transitive postsets that are mapped (i.e.: the contexts of the nodes), provided at least one mapping of transitive preset nodes and one mapping of transitive postset nodes exists.

Definition 3 (Context similarity). *Let G_1 and G_2 be two process graphs. Let $M : N_{G_1} \nrightarrow N_{G_2}$ be a partial injective mapping that maps nodes in G_1 to nodes in G_2. The context similarity of two mapped nodes $n \in N_{G_1}$ and $m \in N_{G_2}$, denoted $Ces(n, m)$ is defined as:*

$$Ces(n,m) = \frac{|M(\overset{c}{\bullet} n) \cap \overset{c}{\bullet} m| + |M(n \overset{c}{\bullet}) \cap m \overset{c}{\bullet}|}{max(|\overset{c}{\bullet} n|, |\overset{c}{\bullet} m|) + max(|n \overset{c}{\bullet}|, |m \overset{c}{\bullet}|)}$$

where M applied to a set yields the set in which M is applied to each element.

$Ces(n, m)$ returns 0 if no mapping exists between nodes in the transitive presets and in the transitive postsets of n and m. For example, the event 'Delivery is relevant for shipment' preceding the AND-join (via a connector chain of size 0) in model G_1 from Fig. 1 is mapped to the event 'Delivery is relevant for shipment' preceding the XOR-join in G_2. Also, the one function succeeding the AND-join (via a connector chain of size 0) in G_1 is mapped to the one function succeeding the XOR-join in G_2. Therefore, the context similarity of the two connectors is: $\frac{1+1}{3+1} = 0.5$. We can now define the similarity of two nodes as follows.

Definition 4 (Node matching score). *Let G_1 and G_2 be two (configurable) process graphs. The similarity of two mapped nodes $n \in N_{G_1}$ and $m \in N_{G_2}$, denoted $Sim(n, m)$ is defined as:*

$$Sim(n,m) = \begin{cases} Sed(n,m) & \text{if } \tau(n) = \tau(m) = f \text{ or } \tau(n) = \tau(m) = e \\ Ces(n,m) & \text{if } \tau(n) = \tau(m) = c \text{ and } \bullet n = \bullet m = 1 \text{ or } n\bullet = m\bullet = 1 \\ 0 & \text{otherwise} \end{cases}$$

Second, we count the number of *Node substitutions* (a node in one graph is substituted for a node in the other graph iff they are matched); *Node insertions/deletions* (a node is inserted into or deleted from one graph iff it is not matched); *Edge substitution* (an edge from node a to node b in one graph is substituted for an edge in the other graph iff node a is matched to node a', node b is matched to node b' and there exists an edge from node a' to node b'); and *Edge insertions/deletions* (an edge is inserted into or deleted from one graph iff it is not substituted).

Third, we return the weighted average of the fraction of inserted/deleted nodes, the fraction of inserted/deleted edges and the average score for node substitutions. More precisely, we define the matching score as follows.

Definition 5 (Matching score). *Let G_1 and G_2 be two process graphs and let M be their mapping function, where* $\mathrm{dom}(M)$ *denotes the domain of M and* $\mathrm{cod}(M)$ *denotes the codomain of M. Let also* $0 \leq \mathrm{wsubn} \leq 1$, $0 \leq \mathrm{wskipn} \leq 1$ *and* $0 \leq \mathrm{wskipe} \leq 1$ *be the weights that we assign to substituted nodes, inserted or deleted nodes and inserted or deleted edges, respectively.*

The set of substituted nodes, denoted subn, inserted or deleted nodes, denoted skipn, substituted edges, denoted sube, and inserted or deleted edges, denoted skipe, are defined as follows:

$$\begin{array}{ll} \text{subn} = \text{dom}(M) \cup \text{cod}(M) & \text{skipn} = (N_{G_1} \cup N_{G_2}) - \text{subn} \\ \text{sube} = \{(a,b) \in E_1 | (M(a), M(b)) \in E_2\} \cup & \text{skipe} = (E_1 \cup E_2) \setminus \text{sube} \\ \quad \{(a',b') \in E_2 | (M^{-1}(a'), M^{-1}(b')) \in E_1\} & \end{array}$$

The fraction of inserted or deleted nodes, denoted fskipn*, the fraction of inserted or deleted edges, denoted* fskipe *and the average distance of substituted nodes, denoted* fsubsn*, are defined as follows.*

$$\text{fskipn} = \frac{|\text{skipn}|}{|N_1|+|N_2|} \quad \text{fskipe} = \frac{|\text{skipe}|}{|E_1|+|E_2|} \quad \text{fsubn} = \frac{2.0 \cdot \Sigma_{(n,m) \in M} 1.0 - \text{Sim}(n,m)}{|\text{subn}|}$$

The matching score is defined as:

$$1.0 - \frac{\text{wskipn} \cdot \text{fskipn} + \text{wskipe} \cdot \text{fskipe} + \text{wsubn} \cdot \text{fsubn}}{\text{wskipn} + \text{wskipe} + \text{wsubn}}$$

For example, in Fig. 1 the node 'Delivery packed' and its edge to the AND-join in G_1 are inserted, and so are the node 'Delivery unblocked' and its edge to the XOR-join in G_2. The AND-join in G_1 is substituted by the second XOR-join in G_2 with a matching score of 0.5, while the node 'Transportation planning and processing' in G_1 is substituted by the node 'Transporting' in G_2 with a matching score of 0.32. Thus, the edge between 'Transportation planning and processing' and the AND-join in G_1 is substituted by the edge between 'Transporting' and the XOR-join in G_2, as both edges are between two substituted nodes. All the other substituted nodes have a matching score of 1.0. If all weights are set to 1.0, the total matching score for this mapping is $1.0 - \frac{\frac{7}{21} + \frac{11}{19} + \frac{2 \cdot 0.5 + 2 \cdot 0.68}{14}}{3} = 0.64$.

Definition 5 returns the matching score given a mapping. To determine the matching score between two business processes, we must exhaustively try all possible mappings and return the one with the highest matching score. Various algorithms exist to find the mapping with the highest matching score [2]. In this paper we use a greedy algorithm [2], since its computational complexity is much lower than that of an exhaustive algorithm, while having a high precision.

3 Merging Algorithm

The merge algorithm is defined over pairs of annotated process graphs. If we want to merge two or more process graphs, we first need to annotate every edge of each process graph with the identifier of the process graph, and every node with a pair indicating the process graph identifier and the label for that node. Then we can proceed to merge the annotated process graphs in a pairwise manner. We first present the basic merge algorithm and then discuss a set of reduction rules to simplify the merged process graph.

3.1 Basic Algorithm

Given two annotated process graphs G_1 and G_2 and their mapping M, the merge algorithm (Algorithm 1) starts by creating an initial version of the merged graph CG by doing an union of the edges of G_1 and G_2, excluding the edges of G_2 that are substituted. In this way for each matched node we keep the copy in G_1 only.

Algorithm 1: Merge

function Merge(Graph G_1, Graph G_2, Mapping M)
init
  Mapping mcr, Graph CG
begin
  $\mathsf{CG} \Leftarrow \mathsf{G}_1 \cup \mathsf{G}_2 \setminus (\mathsf{G}_2 \cap \mathit{sube})$
  foreach $(x, y) \in \mathsf{CG} \cap \mathit{sube}$ **do**
    $\alpha_{\mathsf{CG}}(x, y) \Leftarrow \alpha_{\mathsf{G}_1}(x, y) \cup \alpha_{\mathsf{G}_2}(\mathsf{M}(x), \mathsf{M}(y))$
  end
  foreach $\mathsf{n} \in N_{\mathsf{CG}} \cap \mathit{subn}$ **do**
    $\gamma_{\mathsf{CG}}(\mathsf{n}) \Leftarrow \gamma_{\mathsf{G}_1}(\mathsf{n}) \cup \gamma_{\mathsf{G}_2}(\mathsf{M}(\mathsf{n}))$
  end
  foreach $\mathsf{mcr} \in$ MaximumCommonRegions($\mathsf{G}_1, \mathsf{G}_2, \mathsf{M}$) **do**
    $\mathsf{FG}_1 \Leftarrow \{x \in \mathrm{dom}(\mathsf{mcr}) \mid \bullet x \cap \mathrm{dom}(\mathsf{mcr}) = \emptyset\}$
    foreach $\mathsf{fG}_1 \in \mathsf{FG}_1$ *such that* $|\bullet \mathsf{fG}_1| = 1$ *and* $|\bullet \mathsf{M}(\mathsf{fG}_1)| = 1$ **do**
      $\mathsf{pfG}_1 \Leftarrow \mathrm{Any}(\bullet\mathsf{fG}_1)$, $\mathsf{pfG}_2 \Leftarrow \mathrm{Any}(\bullet\mathsf{M}(\mathsf{fG}_1))$
      $\mathsf{xj} \Leftarrow$ new Node("c","xor",true)
      $\mathsf{CG} \Leftarrow (\mathsf{CG} \setminus (\{(\mathsf{pfG}_1, \mathsf{fG}_1), (\mathsf{pfG}_2, \mathsf{fG}_2)\})) \cup \{(\mathsf{pfG}_1, \mathsf{xj}), (\mathsf{pfG}_2, \mathsf{xj}), (\mathsf{xj}, \mathsf{fG}_1)\}$
      $\alpha_{\mathsf{CG}}(\mathsf{pfG}_1, \mathsf{xj}) \Leftarrow \alpha_{\mathsf{G}_1}(\mathsf{pfG}_1, \mathsf{fG}_1)$, $\alpha_{\mathsf{CG}}(\mathsf{pfG}_2, \mathsf{xj}) \Leftarrow \alpha_{\mathsf{G}_2}(\mathsf{pfG}_2, \mathsf{fG}_2)$
      $\alpha_{\mathsf{CG}}(\mathsf{xj}, \mathsf{fG}_1) \Leftarrow \alpha_{\mathsf{G}_1}(\mathsf{pfG}_1, \mathsf{fG}_1) \cup \alpha_{\mathsf{G}_2}(\mathsf{pfG}_2, \mathsf{fG}_2)$
    end
    $\mathsf{LG}_1 \Leftarrow \{x \in \mathrm{dom}(\mathsf{mcr}) \mid x \bullet \cap \mathrm{dom}(\mathsf{mcr}) = \emptyset\}$
    foreach $\mathsf{lG}_1 \in \mathsf{LG}_1$ *such that* $|\mathsf{lG}_1 \bullet| = 1$ *and* $|\mathsf{M}(\mathsf{lG}_1) \bullet| = 1$ **do**
      $\mathsf{slG}_1 \Leftarrow \mathrm{Any}(\mathsf{lG}_1\bullet)$, $\mathsf{slG}_2 \Leftarrow \mathrm{Any}(\mathsf{M}(\mathsf{lG}_1)\bullet)$
      $\mathsf{xs} \Leftarrow$ new Node("c","xor",true)
      $\mathsf{CG} \Leftarrow (\mathsf{CG} \setminus (\{(\mathsf{lG}_1, \mathsf{slG}_1), (\mathsf{lG}_2, \mathsf{slG}_2)\})) \cup \{(\mathsf{xs}, \mathsf{slG}_1), (\mathsf{xs}, \mathsf{slG}_2), (\mathsf{lG}_1, \mathsf{xs})\}$
      $\alpha_{\mathsf{CG}}(\mathsf{xs}, \mathsf{slG}_1) \Leftarrow \alpha_{\mathsf{G}_1}(\mathsf{lG}_1, \mathsf{slG}_1)$, $\alpha_{\mathsf{CG}}(\mathsf{xs}, \mathsf{slG}_2) \Leftarrow \alpha_{\mathsf{G}_2}(\mathsf{lG}_2, \mathsf{slG}_2)$
      $\alpha_{\mathsf{CG}}(\mathsf{lG}_1, \mathsf{xs}) \Leftarrow \alpha_{\mathsf{G}_1}(\mathsf{lG}_1, \mathsf{slG}_1) \cup \alpha_{\mathsf{G}_2}(\mathsf{lG}_2, \mathsf{slG}_2)$
    end
  end
  $\mathsf{CG} \Leftarrow$ MergeConnectors(M, CG)
  return CG
end

Next, we set the annotation of each edge in CG that originates from a substituted edge, with the union of the annotations of the two substituted edges in G_1 and G_2. For example, this produces all edges with label "1,2" in model CG in Fig. 1. Similarly, we set the annotation of each node in CG that originates from a matched node, with the union of the annotations of the two matched nodes in G_1 and G_2. In Fig. 1, this produces the annotations of the last two nodes of

CG—the only two nodes originating from matched nodes with different labels (the other annotations are not shown in the figure).

Next, we use function *MaximumCommonRegions* to partition the mapping between G_1 and G_2 into maximum common regions (Algorithm 2). A maximum common region (mcr) is a maximum connected subgraph consisting only of matched nodes and substituted edges. For example, given models G_1 and G_2 in Fig. 1, *MaximumCommonRegions* returns the three mcrs highlighted by rounded boxes in the figure. To find all mcrs, we first randomly pick a matched node that has not yet been included in any mcr. We then compute the mcr of that node using a breadth-first search. After this, we choose another mapped node that is not yet in an mcr, and we construct the next mcr.

We then postprocess the set of maximum common regions $MCRs$ to remove from each mcr those nodes that are at the beginning or at the end of one model, but not of the other (this step is not shown in Algorithm 2). Such nodes need not be merged, otherwise it would not be possible to trace back which original model they come from. For example, we do not merge event "Deliveries need to be planned" in Fig. 1 as this node is at the beginning of G_1 and at the end of G_2.

Algorithm 2: Maximum Common Regions

```
function MaximumCommonRegions(Graph G1, Graph G2, Mapping M)
init
  {Node} visited ⇐ ∅, {Mapping} MCRs ⇐ ∅
begin
  while exists c ∈ dom(M) such that c ∉ visited do
    {Node} mcr ⇐ ∅
    {Node} tovisit ⇐ {c}
    while tovisit ≠ ∅ do
      c ⇐ dequeue(tovisit)
      mcr ⇐ mcr ∪ {c}
      visited ⇐ visited ∪ {c}
      foreach n ∈ dom(M) such that ((c, n) ∈ G1 and (M(c), M(n)) ∈ G2) or
      ((n, c) ∈ G1 and (M(n), M(c)) ∈ G2) and n ∉ visited do
        enqueue(tovisit, n)
      end
    end
    MCRs ⇐ MCRs ∪ {mcr}
  end
  return MCRs
end
```

Once we have identified all mcrs, we need to reconnect them with the remaining nodes from G_1 and G_2 that are not matched. The way a region is reconnected depends on the position of its sources and sinks with respect to G_1 and G_2. A region's source is a node whose predecessors are not in the region or do not exist;

a region's sink is a node whose successors are not in the region or do not exist. If a source fG_1 in G_1 and its matched node have exactly one predecessor each, we insert a configurable XOR-join in CG to reconnect the two predecessors to the source. Similarly, if a sink lG_1 in G_1 and its matched node have exactly one successor each, we insert a configurable XOR-split in CG to reconnect the two successors to the sink. We also set the labels of the new edges in CG to track back the edges in the original models. This is depicted in Fig. 2. We use function *Node* to create a new connector and initialize its identifier and annotation, and function *Any* to return the element of a singleton set. The postprocessing of $MCRs$ guarantees that either both a source and its matched node have predecessors or none has, and similarly, that either both a sink and its matched node have successors or none has.

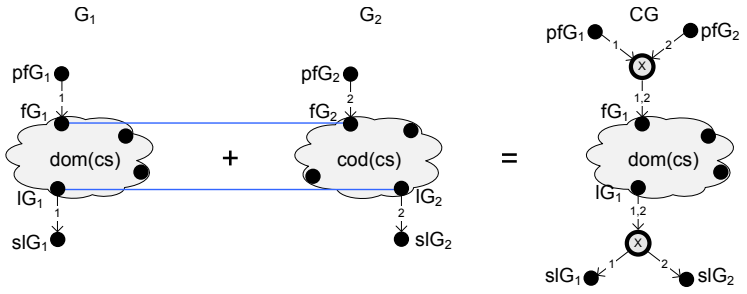

Fig. 2. Reconnecting a maximum common region to the nodes that are not matched.

In the example of Fig. 1, the first mcr is reconnected in CG via a new configurable XOR-split after its sink node "Shipment processing"; the second region is removed after postprocessing $MCRs$; and the third region is reconnected via a configurable XOR-join before its source node "Delivery is relevant for shipment".

The cases in which a source has multiple predecessors (i.e. it is a join) or a sink has multiple successors (i.e. it is a split) are covered by function *MergeConnectors* (Algorithm 3). This function is invoked in the last step of Algorithm 1 and it basically merges the preset and postset of matched connectors. Since every matched connector c in CG is copied from G_1, we need to reconnect to c the predecessors and successors of $M(c)$ that are not matched. We do so by adding a new edge between each predecessor or successor of $M(c)$ and c. If at least one such predecessor or successor exists, we make c configurable, and if there is a mismatch between the labels of the two matched connectors (e.g. one is "xor" and the other is "and"), we also change the label of c to "or". For example, the AND-join in the model G_1 of Fig. 1 is matched with the XOR-join that precedes function "Transporting" in G_2. The only non-matched predecessor of this XOR-join is event "Delivery unblocked". Thus, we reconnect the latter to the AND-join that precedes function "Transportation planning and processing" in CG via a new edge labeled "2". Also, we make c configurable and we change its label to "or", thus obtaining the model CG in Fig. 1.

Algorithm 3: Merge Connectors

function MergeConnectors(Mapping M, {Edge} CG)
init
  {Node} $\mathsf{S} \Leftarrow \emptyset$, {Node} $\mathsf{J} \Leftarrow \emptyset$
begin
  foreach $\mathsf{c} \in \mathrm{dom}(\mathsf{M})$ *such that* $\tau(\mathsf{c}) =$ "c" **do**
    $\mathsf{S} \Leftarrow \{\mathsf{x} \in \mathsf{M}(\mathsf{c})\bullet \mid \mathsf{x} \notin \mathrm{cod}(\mathsf{M})\}$
    $\mathsf{J} \Leftarrow \{\mathsf{x} \in \bullet\mathsf{M}(\mathsf{c}) \mid \mathsf{x} \notin \mathrm{cod}(\mathsf{M})\}$
    $\mathsf{CG} \Leftarrow (\mathsf{CG} \setminus \bigcup_{\mathsf{x}\in\mathsf{S}}\{(\mathsf{M}(\mathsf{c}),\mathsf{x})\} \cup \bigcup_{\mathsf{x}\in\mathsf{J}}\{(\mathsf{x},\mathsf{M}(\mathsf{c}))\}) \cup \bigcup_{\mathsf{x}\in\mathsf{S}}\{(\mathsf{c},\mathsf{x})\} \cup \bigcup_{\mathsf{x}\in\mathsf{J}}\{(\mathsf{x},\mathsf{c})\}$
    foreach $\mathsf{x} \in \mathsf{S}$ **do**
      $\alpha_{\mathsf{CG}}(\mathsf{c},\mathsf{x}) \Leftarrow \alpha_{\mathsf{G}_2}(\mathsf{M}(\mathsf{c}),\mathsf{x})$
    end
    foreach $\mathsf{x} \in \mathsf{J}$ **do**
      $\alpha_{\mathsf{CG}}(\mathsf{x},\mathsf{c}) \Leftarrow \alpha_{\mathsf{G}_2}(\mathsf{x},\mathsf{M}(\mathsf{c}))$
    end
    if $|\mathsf{S}| > 0$ *or* $|\mathsf{J}| > 0$ **then**
      $\eta_{\mathsf{c}} \Leftarrow$ true
    end
    if $\lambda_{\mathsf{G}_1}(\mathsf{c}) \neq \lambda_{\mathsf{G}_2}(\mathsf{M}(\mathsf{c}))$ **then**
      $\lambda_{\mathsf{CG}}(\mathsf{c}) \Leftarrow$ "or"
    end
  end
  return CG
end

3.2 Reduction Rules

After merging two process graphs, we can simplify the resulting graph by applying a set of reduction rules. These rules are used to reduce connector chains that may have been generated after inserting configurable XOR connectors. The idea is to improve the visual presentation of the merged process graph while preserving its behavior and its configuration options. These rules are: 1) remove redundant transitive edges between connectors, 2) merge consecutive splits/joins, and 3) remove trivial connectors (i.e. those connectors with one input edge and one output edge), that may have been generated after applying the first two rules. These rules are applied until a process graph cannot be further reduced.

Remove redundant transitive edges Function *RemoveRedundantEdges* (Algorithm 4) removes all redundant transitive edges. A redundant transitive edge is an edge whose source and target node are also connected via an alternative path made of a connector chain. Thus, the source of a redundant edge must be a split connector, while its target must be a join connector. Given two nodes m and n where m is a split, n is a join and m is connected to n via a redundant transitive edge, we first remove the redundant transitive edge. Next, we set the label of each edge in the connector chain to the union of the edge's label with the label of the redundant edge being removed. Finally, for each connector c in

the connector chain, we set its annotation with the union of its annotation and the annotation of m, we make it configurable and if there is a mismatch between its label and that of m, we change the label to "or". We use function *Alphabet* to retrieve the set of nodes in the connector chain. Fig. 3 shows the application of this rule.

Algorithm 4: Remove Redundant Edges

```
function RemoveRedundantEdges({Edge} CG)
begin
  foreach (m, n) ∈ CG such that |m •| > 1 and | • n| > 1 and exists a path
  p ∈ CG such that p = m ↪c n do
    CG ⇐ CG \ {(m, n)}
    foreach (x, y) ∈ CG such that x ∈ Alphabet(p) and y ∈ Alphabet(p) do
      α(x, y) ⇐ α(x, y) ∪ α(m, n)
    end
    foreach c ∈ Alphabet(p) such that τ(c) = "c" do
      γ(c) ⇐ γ(c) ∪ γ(m)
      η(c) ⇐ true
      if λ(c) ≠ λ(m) then
        λ(c) ⇐ "or"
      end
    end
  end
  return CG
end
```

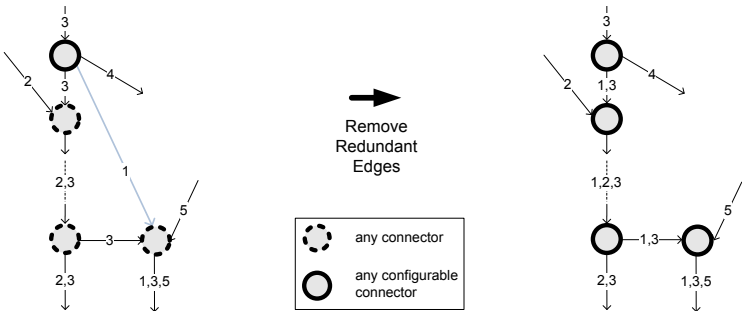

Fig. 3. Removing redundant transitive edges.

Merge consecutive splits/joins Function *MergeConsecutiveConnectors* (Algorithm 5) merges any two consecutive splits/joins into a single split/join connector. The preconditions for applying this rule are that there are no redundant transitive edges, and that at least one of the two connectors is a configurable

XOR that has been added by Algorithm 1. The latter condition is checked by function *IsAdded* which takes an edge as input and returns true if the edge's source or target is a configurable XOR added by the merge algorithm. Given an edge (m, n) where m and n are both splits, we merge m into n. First, we remove all edges connected to n and reconnect the successors of n to m via new edges. Next, we set the labels of the new edges with the labels of the edges being removed, and we set the annotation of m with the union of the annotations of both connectors. Finally, we set the annotation of m with the union of its annotation and the annotation of n, we make m configurable and if there is a mismatch between its label and that of n, we change the label to "or". The case of two consecutive joins is the opposite. Fig. 4 shows the application of this rule.

Algorithm 5: Merge Consecutive Connectors

function MergeConsecutiveConnectors({Edge} CG)
precondition no redundant transitive edges
begin
  foreach $(\mathsf{m}, \mathsf{n}) \in \mathsf{CG}$ *such that* $\tau(\mathsf{m}) = \tau(\mathsf{n}) =$ "c" *and* IsAdded$((\mathsf{m}, \mathsf{n}))$ = true **do**
    if $|\mathsf{m}\bullet| > 1$ *and* $|\mathsf{n}\bullet| > 1$ **then**
      $\mathsf{CG} \Leftarrow (\mathsf{CG} \setminus \{(\mathsf{m}, \mathsf{n})\} \cup \bigcup_{\mathsf{x} \in \mathsf{n}\bullet} \{(\mathsf{n}, \mathsf{x})\}) \cup \bigcup_{\mathsf{x} \in \mathsf{n}\bullet} \{(\mathsf{m}, \mathsf{x})\}$
      foreach $\mathsf{x} \in \mathsf{n}\bullet$ **do**
        $\alpha(\mathsf{m}, \mathsf{x}) \Leftarrow \alpha(\mathsf{n}, \mathsf{x})$
      end
      $\gamma(\mathsf{m}) \Leftarrow \gamma(\mathsf{m}) \cup \gamma(\mathsf{n})$
      $\eta(\mathsf{m}) \Leftarrow$ true
      if $\lambda(\mathsf{m}) \neq \lambda(\mathsf{n})$ **then**
        $\lambda(\mathsf{m}) \Leftarrow$ "or"
      end
    end
    else if $|\bullet\,\mathsf{m}| > 1$ *and* $|\bullet\,\mathsf{n}| > 1$ **then**
      $\mathsf{CG} \Leftarrow (\mathsf{CG} \setminus \{(\mathsf{m}, \mathsf{n})\} \cup \bigcup_{\mathsf{x} \in \bullet\mathsf{m}} \{(\mathsf{x}, \mathsf{m})\}) \cup \bigcup_{\mathsf{x} \in \bullet\mathsf{m}} \{(\mathsf{x}, \mathsf{n})\}$
      foreach $\mathsf{x} \in \bullet\mathsf{m}$ **do**
        $\alpha(\mathsf{x}, \mathsf{n}) \Leftarrow \alpha(\mathsf{x}, \mathsf{m})$
      end
      $\gamma(\mathsf{n}) \Leftarrow \gamma(\mathsf{m}) \cup \gamma(\mathsf{n})$
      $\eta(\mathsf{n}) \Leftarrow$ true
      if $\lambda(\mathsf{m}) \neq \lambda(\mathsf{n})$ **then**
        $\lambda(\mathsf{n}) \Leftarrow$ "or"
      end
    end
  end
  return CG
end

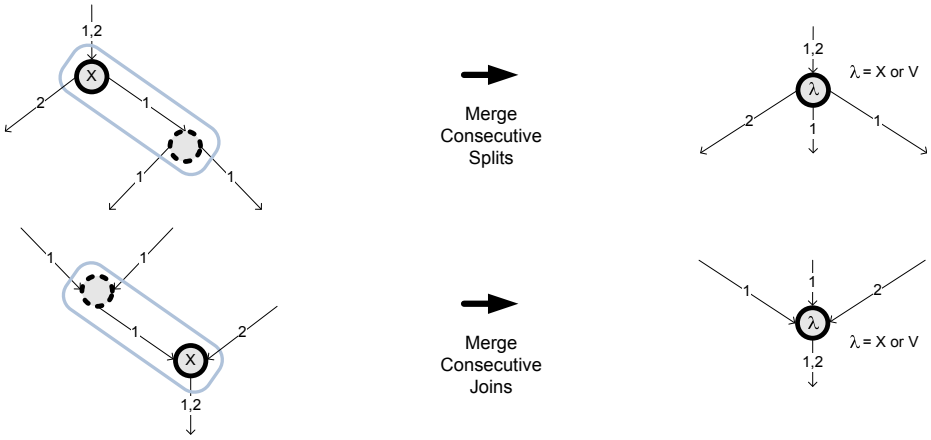

Fig. 4. Merging consecutive splits and joins.

Remove trivial connectors. Function *RemoveTrivialConnectors* (Algorithm 6) removes all connectors that have one input edge and one output edge only. Such connectors do not contain any needful information. Thus, we can get rid of them without losing any process behavior. The precondition for applying this rule is that the trivial connector must be configurable but it can be any configurable connector, i.e. not necessarily a configurable XOR that has been added during the merge. In fact, a trivial connector may be generated from applying *MergeConsecutiveConnectors* and *RemoveRedundantEdges*. We get rid of a trivial connector m by removing the edge from its single predecessor pm and the edge to its single successor sm. Next, we reconnect pm with sm with a new edge, whose annotation is set to the union of the annotations of the two edges being removed. Fig. 5 shows the application of this rule.

Algorithm 6: Remove Trivial Connectors

function RemoveTrivialConnectors({Edge} CG)
begin
  foreach $\mathsf{m} \in \mathsf{N_{CG}}$ *such that* $\tau(\mathsf{m}) =$ "c" *and* $|\bullet\,\mathsf{m}| = |\mathsf{m}\,\bullet| = 1$ *and* $\eta(\mathsf{m}) =$ true **do**
    $\mathsf{pm} = \mathrm{Any}(\bullet\mathsf{m})$, $\mathsf{sm} = \mathrm{Any}(\mathsf{m}\bullet)$
    $\mathsf{CG} \Leftarrow (\mathsf{CG} \setminus \{(\mathsf{pm}, \mathsf{m}), (\mathsf{m}, \mathsf{sm})\}) \cup \{(\mathsf{pm}, \mathsf{sm})\}$
    $\alpha(\mathsf{pm}, \mathsf{sm}) \Leftarrow \alpha(\mathsf{pm}, \mathsf{m}) \cup \alpha(\mathsf{m}, \mathsf{sm})$
  end
  return CG
end

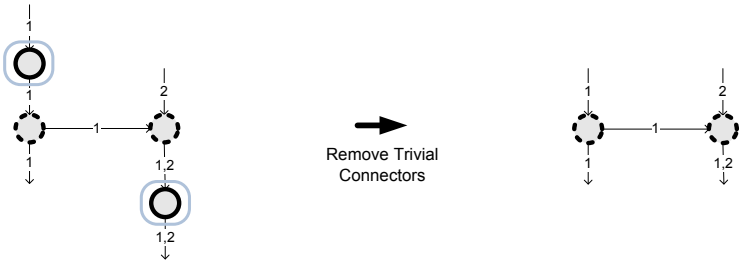

Fig. 5. Removing trivial connectors.

4 Evaluation

The proposed algorithm for process merging has been implemented as a tool that takes as input two EPCs represented in the EPML format, and produces a configurable EPC also represented in EPML. This process merging tool is freely available as part of the Synergia toolset for configurable process modeling, downloadable at `http://www.processconfiguration.com`.

To evaluate the merge operator and its implementation, we conducted a number of tests aimed at evaluating: (i) the basic properties of the operator; (iii) the size of the merged models; and (iii) the scalability of the operator.

Operator's properties. We first undertook to test that the merge operator is idempotent, commutative and associative. The first property means that the merger of a model with itself leads to itself (after excluding the annotations added during the merge). This is a property one would expect in any merge operator. The latter two properties are desirable as they entail that the operator can be used for multi-way merging, i.e. given a collection of process models, one can merge them in a pairwise manner and the order in which the merge operator is applied is not important. Finally, we sought to validate that the input models could be derived back from the merged model, as per the third requirements in Section 1.

To this end, we took a configurable process model for film post-production designed in collaboration with domain experts from the Australian Film, Television & Radio School. The model had 14 possible individualizations, which we generated using the Synergia tool. We tested idempotence by merging every individualized model (i.e. variant) with itself. We then checked commutativity and associativity by merging each subset of the 14 variants (i.e. every pair, triplet, etc.) in every possible order. When merging all 14 variants, we obtained the same configurable process model from which we had derived the 14 variants, thus confirming that the original models can be derived from the merged one.

Size of merged models. The second part of the evaluation aimed to compare the sizes of the merged models to the sizes of the input models. Size is a key factor affecting the understandability of process models and it is thus important that merged models are as compact as possible.

We took the SAP reference model, consisting of 604 EPCs, and constructed every pair of EPCs from among them. We then filtered out pairs in which a model was paired with itself and pairs for which the similarity between the models was less than 0.5. Thus, we were left with pairs of similar but non-identical EPCs. After this filtering step, we obtained 78 model pairs.

Next, we merged each of these model pairs and calculated the ratio between the size of the merged model and the size of the input models. This ratio is called the *compression factor* and is defined as $CF(G_1, G_2) = |CG|/(|G_1|+|G_2|)$, where G_1 and G_2 are two process graphs and CG is the result of merging G_1 and G_2. A compression factor of 1 means that the size of the merged model is equal to the sum of the sizes of the input models, which means that the merge operator merely juxtaposes the two input models side-by-side. A compression of 0.5 means that the size of the merged model is equal to the average size of the input models. This situation may occur when the merged models are very similar and thus the merged model is essentially equal to one of the input models.

Table 1 presents the results of merging the selected pairs of models from the SAP reference model. The first and second column show the size of the initial models. The third and fourth column show the size of the merged model and the compression factor before applying any reduction rules, while the remaining two columns show the size of the merged model and the compression factor after applying the reduction rules. The table shows that the reduction rules improve slightly the compression factor (average of 72% versus 75% before reduction rules), but the main compression is given by the merge algorithm itself. In other words, by identifying and factoring out common regions during the merge procedure, we already obtain models that are significantly smaller than those that one would obtain by simply juxtaposing the merged models against one another.

	Size 1	Size 2	Size merged	Compression	Merged after reduction	Compression after reduction
Min	3	3	5	0.5	5	0.5
Max	118	130	195	1.03	188	1.0
Avg.	22.59	23.03	32.28	0.75	30.82	0.72
Stdev	20.5	20.5	26.57	0.17	25.14	0.16

Table 1. Size statistics of merged SAP reference models.

The more two process models are similar, the smaller is the compression factor of the merged model. Figure 6 provides a scatter plot showing the compression factors (X axis) obtained for different similarity measures between the input models (Y axis). The solid line is the linear regression of the points.

Scalability. Finally, we conducted tests with real-life process models of large size in order to validate the scalability of the approach. To this end, we considered four model pairs of models. The first three pairs of models were provided by a large insurance company. The models capture processes for handling claims for motor incidents and for personal injury incidents. The first pair of models

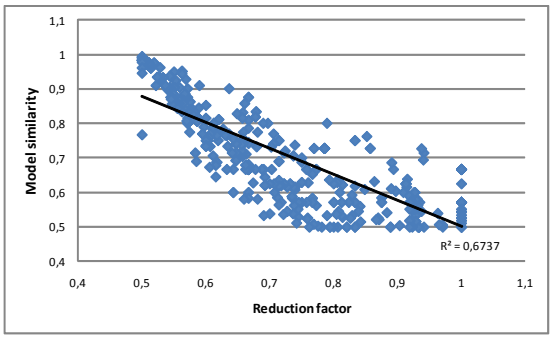

Fig. 6. Correlation between similarity of the input models and compression factor.

corresponds to the claim initiation phase (one model for motor incident and one for personal injury), the second pair of models are for claim lodgment and handling and the third pair are for payment of invoices associated to a claim. Together, these models cover the entire end-to-end process for claims handling. Each pair of models has a high similarity, but they diverge in certain points due to differences in the object of the claim (vehicle vs. personal injury). At the time the models were provided to us, a team of five analysts at the insurance company had been manually merging the process models. It took them 600 man-hours to merge 25% of the end-to-end process models. This manual effort was interrupted when the team learned that the effort could be automated using our approach.

A fourth pair of models was obtained from an agency specialized in applications for developing parcels of land (mainly for commercial purposes). One model captures how land development applications are handled in South Australia while the other captures the same process in Western Australia. The similarity between these two models was high because they cover the same process and were designed by the same analysts. However, due to regulatory differences, the models diverge in certain points.

Table 2 shows the sizes of the input models, the execution time of the merge operator and statistics related to the size of the merged models. The tests were conducted on a laptop with a dual core Intel processor, 1.8 GHz, 2 GB memory, running Microsoft Vista and SUN Java Virtual Machine version 1.6 (with 512MB of allocated memory). The results show that the merge operator can handle pairs of models with around 350 nodes each in a matter of seconds. Models with less than 50 nodes are merged in sub-second times – an observation supported by the execution times we observed when merging the pairs of SAP reference models. Table 2 also shows the size of the merged models and the compression factors for the insurance and land development processes. The numbers for the first and third model pair are in line with those observed for the SAP reference models. The compression factors for the second and third pairs are on the high end. Closer inspection of these pairs of models showed that they had strong differences.

Pair #	Size 1	Size 2	Merge time (in sec.)	Size merged	Compression	Merged after reduction	Compression after reduction
1	340	357	6.04	491	0,70	470	0.67
2	22	78	1.02	88	0,88	87	0.87
3	469	213	3.76	598	0,88	590	0,87
4	200	191	1.03	296	0,76	286	0,73

Table 2. Results of merging land development and insurance models.

5 Related Work

Gottschalk et al. [4] study the problem of merging pairs of EPCs. Their technique first constructs an abstraction of each EPC, namely a function graph, in which all connectors are removed and replaced with annotations attached to the edges. Function graphs are merged by means of set union. The connectors are then restituted by inspecting the annotations attached to the edges in the merged function graph. This approach does not address the second and third criteria in Section 1: there is no information in the generated EPCs allowing one to trace the origin of each element nor to derive the original models from the merged one. Also, they only merge two nodes if they have identical labels, whereas our approach supports approximate matching. Finally, they assume that the input models have a single start and a single end event and no connector chains.

Li et al. [9] propose another approach to merging process models. Given a set of similar process models (the "variants") their technique constructs a single model (the "generic" model) such that the sum of the *change distances* between each variant and the generic model is minimal. The *change distance* is the minimal number of change operations needed to transform one model into another. This work does not fulfill the three criteria in Section 1. The generic model does not necessarily subsume any of the initial variants and there is no information for tracing the origin of the elements in the generic model.

Sun et al. [11] describe yet another approach to process model merging in the context of Workflow nets. Their approach starts from a mapping between tasks in the two process models. Mapped tasks are copied into the merged model directly. On the other hand, regions where the two process models differ, are merged by applying a set of "merge patterns" (sequential, parallel, conditional and iterative). This procedure requires input from the modeler. The proposed technique does not fill any of the criteria in Section 1: when the sequential, parallel or iterative merge operators are used, the merged model does not subsume the initial variants. Also, the merged model does not provide traceability. Finally, their merging technique only works for block-structured process models.

Kuster et al. [6] outline requirements for a process merging tool. Their emphasis is on merging models in the context of version conflicts. Their envisaged merge procedure is not intended to be fully automated. Instead the aim is to assist modellers in resolving differences manually. In [5] they show how changes between pairs of models are detected and classified in their tool.

Ryndina et al. [7] propose a method for merging state machines describing the lifecycle of independent objects involved in a business process, into a single

UML activity diagram capturing the overall process. Because the aim is to merge partial (and disjoint) views of a process model, their technique significantly differs from ours. In [7], the problem of merging tasks that are similar but not identical is not posed. Similarly, the lifecycles to be merged are assumed to be consistent, which eases the merge procedure. Finally, they do not consider the traceability requirement formulated in Section 1.

6 Conclusion

The paper presented a merge operator that takes as input a pair of process models and produces a (configurable) process model. The operator ensures that the merged model subsumes the original model and that the original models can be derived back by individualizing the merged model. Additionally, the merged model is kept as compact as possible in order to enhance its understandability.

The merge operator was implemented and evaluated using process models from practice. The evaluation showed that the merge operator is idempotent, commutative and associative. The commutativity and associativity properties make the operator suitable for merging collections of three or more models. The evaluation also demonstrated that the operator can deal with models of realistic size (even hundreds of nodes per model) and that the merged models are significantly smaller than the sum of the sizes of the original models. For a set of 78 model pairs from the evaluation, the compression factor of the merged process models was 0.76, meaning that they are 76% of the size of the models from which they were derived.

The proposed merge operator creates a model that is a union of the input models. In some scenarios, we are not interested in the union of the input models, but rather in a "summarized version" of the input models showing the most frequently observed behaviour across the input models. In future work, we plan to define a variant of our merge operator that addresses this requirement. Also, the merge operator proposed in this paper assumes that the input models are "flat", i.e. they are not decomposed into sub-processes. When merging process models with sub-processes, we would ideally want to preserve the process decomposition. Addressing this limitation is a direction for future work.

References

1. H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 18(8):689–694, 1997.
2. R.M. Dijkman, M. Dumas, and L. García-Ba nuelos. Graph matching algorithms for business process model similarity search. In *Proceedings of the 7th International Conference on Business Process Management (BPM)*, LNCS. Springer, 2009.
3. R.M. Dijkman, M. Dumas, L. García-Ba nuelos, and Reina Käärik. Aligning business process models. In *Proceedings of the 13th IEEE EDOC Conference*. IEEE Press, 2009.

4. F. Gottschalk, W. M. P. van der Aalst, and M. H. Jansen-Vullers. Merging event-driven process chains. In *Proceedings of the Confederated International Conferences "On the Move to Meaningful Internet Systems"*, volume 5331 of *LNCS*, pages 418–426, Monterrey, Mexico, November 2008. Springer.
5. J. Malte Küster, C. Gerth, A. Förster, and G. Engels. Detecting and resolving process model differences in the absence of a change log. In *Proceedings of the 6th International Conference on Business Process Management*, volume 5240 of *LNCS*, pages 244–260, Milan, Italy, September 2008. Springer.
6. J.M. Küster, C. Gerth, A. Förster, and G. Engels. A tool for process merging in business-driven development. In *Proceedings of the Forum at the CAiSE'08 Conference*, volume 344 of *CEUR Workshop Proceedings*, pages 89–92, Montpellier, France, June 2008. CEUR-WS.org.
7. J.M. Küster, K. Ryndina, and H. Gall. Generation of business process models for object life cycle compliance. In *In Proceedings of the 5th International Conference on Business Process Management*, volume 4714 of *LNCS*, pages 165–181, Brisbane, Australia, September 2007. Springer.
8. I Levenshtein. Binary code capable of correcting deletions, insertions and reversals. *Cybernetics and Control Theory*, 10(8):707–710, 1966.
9. C. Li, M. Reichert, and A. Wombacher. Discovering reference models by mining process variants using a heuristic approach. In *Proceedings of the 7th International Conference on Business Process Management*, volume 5701 of *LNCS*, pages 344–362, Ulm, Germany, September 2009. Springer.
10. M. Rosemann and W. M. P. van der Aalst. A configurable reference modelling language. *Information Systems*, 32(1):1–23, 2007.
11. S. Sun, A. Kumar, and J. Yen. Merging workflows: A new perspective on connecting business processes. *Decision Support Systems*, 42(2):844–858, 2006.
12. B. F. van Dongen, R. M. Dijkman, and J. Mendling. Measuring similarity between business process models. In *Proc. of CAiSE 2008*, volume 5074 of *LNCS*, pages 450–464. Springer, 2008.