Merging Business Process Models

Marcello La Rosa[1], Marlon Dumas[2], Reina Uba[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 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 a pair 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 which model(s) a given element in the merged model originates 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 create synergies. To this end, teams of business analysts need to 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 three analysts 130 man-hours to merge 25% of two variants of an end-to-end process model.

In this paper, we consider the problem of (semi-)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* [15]. 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 their differences are, and why and how these differences occur. Given a configurable process model, analysts can derive individual members of the underlying process family by means of a procedure known as *individualization*. We contend that configurable process models are a suitable output for a process merging algorithm, because they provide a mechanism to fulfill the second and third requirements outlined above. Moreover, they can be used to derive new process models that were not available in the originating process family, e.g. when the need to capture new business procedures arises. In this respect, the merged model can be seen as a reference model [4] for the given process family.

The algorithm requires as input a mapping that defines which elements from one process model correspond to which elements from another process model. To assist in the construction of this mapping, a mapping is suggested to the user who can then adapt the mapping if necessary.The algorithm has been tested on process models sourced from different domains. The tests show that the process merging algorithm produces compact models and 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 technique for proposing an initial mapping between similar process model elements. Section 3 presents the process merging algorithm. Section 4 reports on the implementation and evaluation of the algorithm. Finally, Section 5 discusses related work and Section 6 draws conclusions.

2 Background

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. This latter technique is used to assist users in determining which pairs of process model elements should be considered as equivalent when merging.

2.1 Configurable Business Processes

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 as per the following definition. This process abstraction allows us to merge process models defined in different notations.

Definition 1 (Business Process Graph). *A* business process graph G *is a set of pairs of process model nodes—each pair denoting a directed edge. A node* n *of* G *is a tuple* $(id_G(n), \lambda_G(n), \tau_G(n))$ *consisting of a unique identifier* $id_G(n)$

(of type string), a label $\lambda_G(n)$ (of type string), and a type $\tau_G(n)$. In situations where there is no ambiguity, we will drop the subscript G from id_G, λ_G and τ_G.

For a business process graph G, its set of nodes, denoted N_G, is $\bigcup\{\{n_1, n_2\}|(n_1, n_2) \in G\}$. Each node has a type. The available types of nodes depend on the language that is used. For example, BPMN 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, $\tau_G \in \{$"f", "e", "c"$\}$ where the letters represent the (f)unction, (e)vent and (c)onnector type. 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) [15] is an EPC where some connectors are marked as configurable. A configurable connector can be configured by removing one or more of its incoming branches (in the case of a join) or one or more of its outgoing branches (in the case of a split). The result is a regular connector with a possibly reduced number of incoming or outgoing branches. In addition, a configurable OR connector can be mutated into a regular XOR or a regular AND. After all nodes in a C-EPC are configured, a C-EPC needs to be individualized by removing those branches that have been excluded during the configuration of 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 connectors are removed during individualization. For more details on the individualization algorithm, we refer to [15].

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

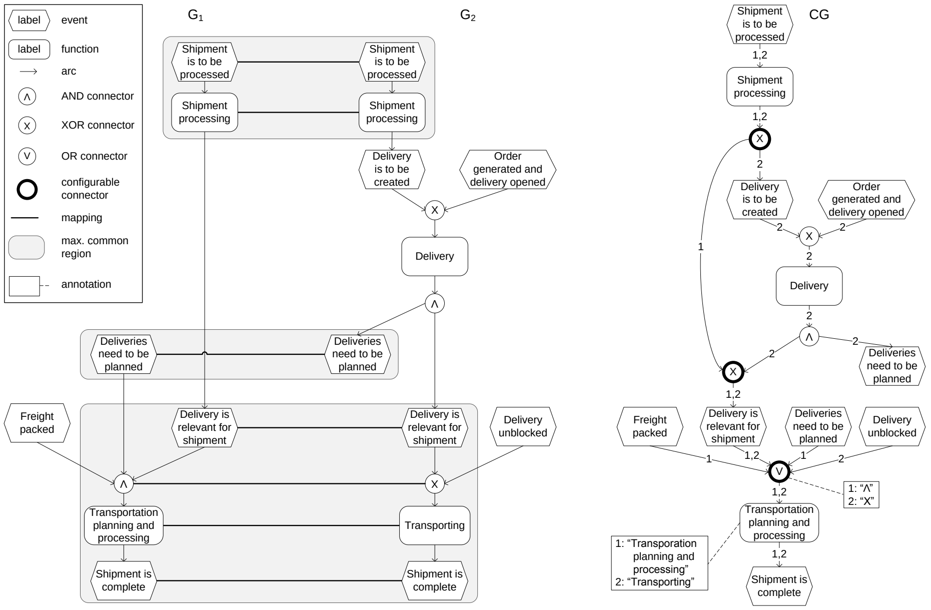

Fig. 1. Two business process models with a mapping, and their merged model.

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.

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 the concept of *Configurable Process Graph*, which attaches additional configuration metadata to each edge and node in a business process graph.

Definition 2 (Configurable Business Process Graph). *Let $\mathcal{I}$ be a set of identifiers of business process models, and $\mathcal{L}$ the set of all labels that pro-*

cess model nodes can take. A Configurable Business Process graph is a tuple $(G, \alpha_G, \gamma_G, \eta_G)$ *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,* $\gamma_G : N_G \rightarrow \wp(\mathcal{I} \times \mathcal{L})$ *is a function that maps each node* $n \in N_G$ *to a set of pairs* (pid, l) *where* pid *is a process graph identifier and* l *is the label of node* n *in process graph* pid*, and* $\eta_G : N_G \rightarrow \{true,false\}$ *is a boolean indicating whether a node is configurable or not.*

Because we attach annotations to graph elements, our concept of configurable process graph slightly differs from the one defined in [15].

Below, we define some auxiliary notations which we will use when matching pairs of process graphs.

Definition 3 (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. Similarly,* $n \overset{c}{\bullet}= \{m \in N_G | n \overset{c}{\hookrightarrow} m\}$ *is the* transitive postset *of* n *via connector chains.*

For example, the transitive preset of event "Delivery is relevant for shipment" in Figure 1, includes functions "Delivery" and "Shipment Processing", since these two latter functions can be reached from the event by traversing backward edges and skipping any connectors encountered in the backward path.

2.2 Matching Business Processes

The aim of matching two process models is to establish the best mapping between their nodes. Here, a mapping is a function from the nodes in the first graph to those in the second graph. 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 as it performed well in several empirical studies [17, 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 between each pair of nodes as follows. Nodes of different types must not be mapped, and splits must not 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 is measured by the *similarity* of their labels. To determine this similarity, we use a combination of a syntactic similarity measure, based on *string edit distance* [10], and a linguistic similarity measure, based on the Wordnet::Similarity package [13] (if specific ontologies for a domain are available, such ontologies can be used instead of Wordnet). We apply these measures on pairs of words from the two labels, after removing stop-words (e.g. articles and conjunctions) and stemming the remaining words (to remove word endings such as ”-ing”). The similarity between two words is the maximum between their syntactic similarity and their linguistic similarity. The total similarity between two labels is the average of the similarities between each pair of words (w_1, w_2) such that w_1 belongs to the first label and w_2 belongs to the second label. With reference to the example in Fig. 1, the similarity score between nodes ‘Transportation planning and processing’ in G_1 and node ‘Transporting’ in G_2 is around 0.35. After removing the stop-word “and”, we have three pairs of terms. The similarity between ‘Transportation” and “‘Transporting” after stemming is 1.0, while the similarity between “plan” and “process” or between “plan” and “transport” is close to 0. The average similarity between these three pairs is thus around 0.35. This approach is directly inspired from established techniques for matching pairs of elements in the context of schema matching [14].

The above approach to compute similarities between functions/events cannot be used to compute the similarity between pairs of splits or pairs of joins, as connectors' labels are restricted to a small set (e.g. ‘OR’, ‘XOR’ and ’AND’) and they each have a specific semantics. Instead, we use a notion of *context similarity.* Given two mapped nodes, context similarity is the fraction of nodes in their transitive presets and their 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 4 (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 $m \in N_{G_2}$ is:*

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

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

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 function succeeding the AND-join (via a connector chain of size 0) in G_1 is mapped to the function succeeding the XOR-join in G_2. Therefore, the context similarity of the two joins is: $\frac{1+1}{3+1} = 0.5$.

Second, we derive from the mapping the number of: *Node substitutions* (a node in one graph is substituted for a node in the other graph iff they appear in the mapping); *Node insertions/deletions* (a node is inserted into or deleted from one graph iff it does not appear in the mapping); *Edge substitutions* (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 use the matching scores from step one and the information about substituted, inserted and deleted nodes and edges from step two, to compute the matching score for the mapping as a whole. We define the matching score of a mapping as the weighted average of the fraction of inserted/deleted nodes, the fraction of inserted/deleted edges and the average score for node substitutions. Specifically, the matching score of a pair of process graphs and a mapping between them is defined 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, and let $Sim(n, m)$ be the function that returns the similarity score for a pair of mapped nodes, as computed in step one.*
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:

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

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.*

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

Finally, the matching score of a mapping is defined as:

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

For example, in Fig. 1 the node ‘Freight 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 0.35 as discussed above. 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 $1.0 - \frac{\frac{7}{21} + \frac{11}{19} + \frac{2 \cdot 0.5 + 2 \cdot 0.65}{14}}{3} = 0.64$.

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

3 Merging Algorithm

The merging algorithm is defined over pairs of configurable process graphs. In order to merge two or more (non-configurable) process graphs, we first need to convert each process graph into a configurable process graph. This is trivially achieved by annotating every edge of a process graph with the identifier of the process graph, and every node in the process graph with a pair indicating the process graph identifier and the label for that node. We then obtain a configurable process graph representing only one possible variant.

Given two configurable process graphs G_1 and G_2 and their mapping M, the merging algorithm (Algorithm 1) starts by creating an initial version of the merged graph CG by doing the 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. 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 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 cannot be merged, otherwise it would not be possible to trace back which 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. In this case, since the mcr contains this node only, we remove the mcr altogether.

Algorithm 1: Merge

```
function Merge(Graph $\mathsf{G}_1$, Graph $\mathsf{G}_2$, Mapping $\mathsf{M}$)
init
  Mapping $\mathsf{mcr}$, Graph $\mathsf{CG}$
begin
  $\mathsf{CG} \Leftarrow \mathsf{G}_1 \cup \mathsf{G}_2 \setminus (\mathsf{G}_2 \cap \mathit{sube})$
  foreach $(\mathsf{x}, \mathsf{y}) \in \mathsf{CG} \cap \mathit{sube}$ do
    $\alpha_{\mathsf{CG}}(\mathsf{x}, \mathsf{y}) \Leftarrow \alpha_{\mathsf{G}_1}(\mathsf{x}, \mathsf{y}) \cup \alpha_{\mathsf{G}_2}(\mathsf{M}(\mathsf{x}), \mathsf{M}(\mathsf{y}))$
  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
    $\mathsf{FG}_1 \Leftarrow \{\mathsf{x} \in \mathrm{dom}(\mathsf{mcr}) \mid \bullet\mathsf{x} \cap \mathrm{dom}(\mathsf{mcr}) = \varnothing \vee \bullet\mathsf{M}(\mathsf{x}) \cap \mathrm{cod}(\mathsf{mcr}) = \varnothing\}$
    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
    $\mathsf{LG}_1 \Leftarrow \{\mathsf{x} \in \mathrm{dom}(\mathsf{mcr}) \mid \mathsf{x}\bullet \cap \mathrm{dom}(\mathsf{mcr}) = \varnothing \vee \mathsf{M}(\mathsf{x})\bullet \cap \mathrm{cod}(\mathsf{mcr}) = \varnothing\}$
    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$(\mathsf{M}, \mathsf{CG})$
  return $\mathsf{CG}$
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 in G_1 and G_2. A region's source is a node whose preset is empty (the source is a start node) or at least one of its predecessors is not in the region; a region's sink is a node whose postset is empty (the sink is an end node) or at least one of its successors is not in the region. We observe that this condition may be satisfied by a node in one graph but not by its matched node in the other graph. For example, a node may be a source of a region for G_2 but not for G_1.

If a node fG_1 is a source in G_1 or its matched node $M(fG_1)$ is a source in G_2 and both fG_1 and $M(fG_1)$ have exactly one predecessor each, we insert a configurable XOR-join xj in CG to reconnect the two predecessors to the copy of fG_1 in CG. Similarly, if a node lG_1 is a sink in G_1 or its matched node $M(lG_1)$

Algorithm 2: Maximum Common Regions

```
function MaximumCommonRegions(Graph $\mathsf{G}_1$, Graph $\mathsf{G}_2$, Mapping $M$)
init
  {Node} visited $\Leftarrow \emptyset$, {Mapping} MCRs $\Leftarrow \emptyset$
begin
  while exists $\mathsf{c} \in \mathrm{dom}(\mathsf{M})$ such that $\mathsf{c} \notin$ visited do
    {Node} mcr $\Leftarrow \emptyset$
    {Node} tovisit $\Leftarrow \{\mathsf{c}\}$
    while tovisit $\neq \emptyset$ do
      c $\Leftarrow$ dequeue(tovisit)
      mcr $\Leftarrow$ mcr $\cup \{\mathsf{c}\}$
      visited $\Leftarrow$ visited $\cup \{\mathsf{c}\}$
      foreach $\mathsf{n} \in \mathrm{dom}(\mathsf{M})$ such that $((\mathsf{c},\mathsf{n}) \in \mathsf{G}_1$ and $(\mathsf{M}(\mathsf{c}),\mathsf{M}(\mathsf{n})) \in \mathsf{G}_2)$ or
      $((\mathsf{n},\mathsf{c}) \in \mathsf{G}_1$ and $(\mathsf{M}(\mathsf{n}),\mathsf{M}(\mathsf{c})) \in \mathsf{G}_2)$ and $\mathsf{n} \notin$ visited do
        enqueue(tovisit, n)
      end
    end
    MCRs $\Leftarrow$ MCRs $\cup \{\mathsf{mcr}\}$
  end
  return MCRs
end
```

is a sink in G_2 and both nodes have exactly one successor each, we insert a configurable XOR-split xs in CG to reconnect the two successors to the copy of lG_1 in CG. We also set the labels of the new edges in CG to track back the edges in the original models. This is illustrated in Fig. 2 where we use symbols pfG_1 to indicate the only predecessor of node fG_1 in G_1, slG_1 to indicate the only successor of node lG_1 in G_1 and so on. Moreover, in Algorithm 1 we use function *Node* to create the configurable XOR joins and splits that we need to add, and function *Any* to extract the element of a singleton set.

In Fig. 1, node "Shipment processing" in G_1 and its matched node in G_2 are both sink nodes and have exactly one successor each ("Delivery is relevant for shipment" in G_1 and "Delivery is to be created" in G_2). Thus, we reconnect this node in CG to the two successors via a configurable XOR-join and set the labels of the incoming and outgoing edges of this join accordingly. The same operation applies when a node is source (sink) in a graph but not in the other.

By removing from $MCRs$ all the nodes that are at the beginning or at the end of one model but not of the other, we guarantee 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. In Fig. 1, the region containing node "Deliveries need to be planned" is removed after postprocessing $MCRs$ since this node is a start node for G_1 and an end node for G_2.

If a source has multiple predecessors (i.e. it is a join) or a sink has multiple successors (i.e. it is a split), we do not need to add a configurable XOR-join before the source, or a configurable XOR-split after the sink. Instead, we can simply reconnect these nodes with the remaining nodes in their preset (if a

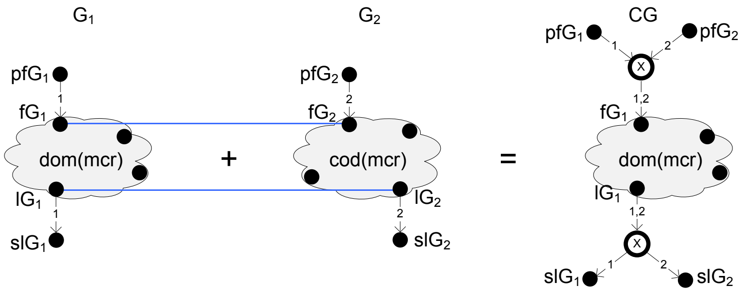

Fig. 2. Reconnecting a maximum common region to the nodes that are not matched.

join) or postset (if a split) which are not matched. This case is covered by function *MergeConnectors* (Algorithm 3). This function is invoked in the last step of Algorithm 1 to merge the preset and postset of all matched connectors, including those that are source or sink of a region, as well as any matched connector inside a region. In fact the operation that we need to perform is the same in both cases. 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 G_1 of Fig. 1 is matched with the XOR-join that precedes function "Transporting" in G_2. Since both nodes are source of the region in their respective graphs, we do not need to add a further configurable XOR-join. The only non-matched predecessor of the XOR-join in G_2 is node "Delivery unblocked". Thus, we reconnect the latter to the copy of the AND-join in CG via a new edge labeled "2". Also, we make this connector configurable and we change its label to "or", obtaining graph CG in Fig. 1.

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. This reduces the size of the merged process graph while preserving its behavior and its configuration options. The reduction rules are: 1) merge consecutive splits/joins, 2) remove redundant transitive edges between connectors, and 3) remove trivial connectors (i.e. those connectors with one input edge and one output edge), and are applied until a process graph cannot be further reduced. For space reasons, we cannot provide full details of the reduction rules. Detailed explanations and formal descriptions of the rules are given in a technical report [9].

The worst-case complexity of the process merging procedure is $O(|N_G|^3)$ where $|N_G|$ is the number of nodes of the largest graph. This is the complexity of the process mapping step when using a greedy algorithm [2], which dominates the complexity of the other steps of the procedure. The complexity of the algorithm for merging connectors is linear on the number of connectors. The algorithm for

Algorithm 3: Merge Connectors

```
function MergeConnectors(Mapping $\mathsf{M}$, {Edge} $\mathsf{CG}$)
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
      $\eta_{\mathsf{CG}}(\mathsf{c}) \Leftarrow$ 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 $\mathsf{CG}$
end
```

calculating the maximum common regions is a breadth-first search, thus linear on the number of edges. The algorithm for calculating the merged model calls the algorithm for calculating the maximum common regions, then visits at most all nodes of each maximum common region, and finally calls the algorithm for merging connectors. Since the number of nodes in a maximum common region and the number of maximum common regions are both bounded by the number of edges, and given that different regions do not share edges, the complexity of the merging algorithm is also linear on the number of edges.

The merged graph subsumes the input graphs in the sense that the set of traces induced by the merged graph includes the union of the traces of the two input graphs. The reason is that every node in an input graph has a corresponding node in the merged graph, and every edge in any of the original graphs has a corresponding edge (or pair of edges) in the merged graph. Hence, for any *run* of the input graph (represented as a sequence of traversed edges) there is a corresponding run in the merged graph. The run in the merged graph has additional edges which correspond to edges that have a configurable xor connector either as source or target. From a behavioral perspective, these configurable xor connectors are "silent" steps which do not alter the execution semantics. If we abstract from these connectors, the run in the input graph is equivalent to the corresponding run in the merged graph. Furthermore, each reduction rule is behavior-preserving. A detailed proof is outside the scope of this paper.

We observe that the merging algorithm accepts both configurable and non-configurable process graphs as input. Thus, the merging operator can be used for multi-way merging. Given a collection of process graphs to be merged, we can start by merging the first two graphs in the collection, then merge the resulting configurable process graph with the third graph in the collection and so on.

4 Evaluation

The algorithm for process merging has been implemented as a tool which is freely available as part of the Synergia toolset (see: `http://www.processconfiguration.com`). The tool takes as input two EPCs represented in the EPML format and suggests a mapping between the two models. Once this mapping has been validated by the user, the tool produces a configurable EPC in EPML by merging the two input models. Using this tool, we conducted tests in order to evaluate (i) the size of the models produced by the merging operator, and (ii) the scalability of the merging operator.

Size of merged models. Size is a key factor affecting the understandability of process models and it is thus desirable that merged models are as compact as possible. Of course, if we merge very different models, we can expect that the size of the merged model will almost equal to the sum of the sizes of the two input models, since we need to keep all the information in the original models. However, if we merge very similar models, we expect to obtain a model whose size is close to the size of the largest of the two models.

We conducted tests aimed at comparing the sizes of the models produced by the merging operator relative to the sizes of the input models. For these tests, 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 matching score of the models was less than 0.5. As a result of the filtering step, we were left with 489 pairs of similar but non-identical EPCs. 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 $CG = Merge(G_1, G_2)$. A compression factor of 1 means that the input models are totally different and thus the size of the merged model is equal to the sum of the sizes of the input models (the merging operator merely juxtaposes the two input models side-by-side). A compression factor close to 0.5 (but still greater than 0.5) means that the input models are very similar and thus the merged model is very close to one of the input models. Finally, if the matching score of the input models is very low (e.g. only a few isolated nodes are similar), the addition of configurable connectors may induce an overhead explaining a compression factor above 1.[1]

[1] In file compression, the compression factor is defined as $1 - |CG|/(|G_1| + |G_2|)$, but here we use the reverse in order to compare this factor with the matching score.

Table 1 summarizes the test results. The first two columns 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 rule, while the last 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 the compression factor (average of 68% vs. 75%), but the merging algorithm itself yields the bulk of the compression. This can be explained by the fact that the merging algorithm factors out common regions when merging. In light of this, we can expect that the more similar two process models are, the more they share common regions and thus the smaller the compression factor is. This hypothesis is confirmed by the scatter plot in Figure 3 which shows the compression factors (X axis) obtained for different matching scores of the input models (Y axis). The solid line is the linear regression of the points.

	Size 1	Size 2	Size merged	Compression	Merged after reduction	Compression after reduction
Min	3	3	3	0.5	3	0.5
Max	130	130	194	1.17	186	1.05
Average	22.07	24.31	33.90	0.75	31.52	0.68
Std dev	20.95	22.98	30.35	0.15	28.96	0.13

Table 1. Size statistics of merged SAP reference models.

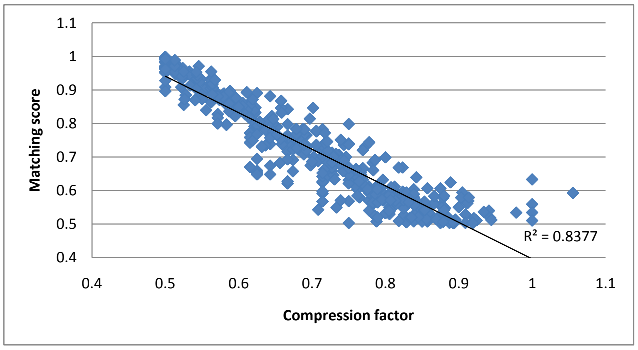

Fig. 3. Correlation between matching score of input models and compression factor.

Scalability. We also conducted tests with large process models in order to assess the scalability of the proposed merging operator. We considered four model pairs. The first three pairs capture a process for handling motor incident and personal injury claims at an Australian insurer. The first pair corresponds to the claim initiation phase (one model for motor incident and one for personal injury), the second pair corresponds to claim processing and the third pair corresponds to payment of invoices associated to a claim. Each pair of models has a high similarity, but they diverge due to differences in the object of the claim.

A fourth pair of models was obtained from an agency specialized in handling applications for developing parcels of land. 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 models was high since they cover the same process and were designed by the same analysts. However, due to regulatory differences, the models diverge in certain points.

Pair #	Size 1	Size 2	Merge time (msec.)	Size merged	Compression	Merged after reduction	Compression after reduction
1	339	357	79	486	0.7	474	0.68
2	22	78	0	88	0.88	87	0.87
3	469	213	85	641	0.95	624	0.92
4	200	191	20	290	0.75	279	0.72

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

Table 2 shows the sizes of the input models, the execution time of the merging operator and statistics related to the size of the merged models. The tests were conducted on a laptop with a dual core Intel processor, 2.53 GHz, 3 GB memory, running Microsoft Vista and SUN Java Virtual Machine version 1.6 (with 512MB of allocated memory). The execution times refer to the merging step only, excluding the time taken to read the models from disk and to match them.

The results show that the merging operator can handle pairs of models with around 350 nodes each in a matter of milliseconds—an observation supported by the execution times we observed when merging the pairs from the SAP reference model. Table 2 also shows the compression factors. Pairs 2 and 3 have a poor compression factor (lower is better). This is in great part due to difference in the size of these two models, which yields a low matching score. For example, in the case of pair 2 (matching score of 0.56) it can be seen that the merged model is only slightly larger than the larger of the two input models.

When the insurance process models were given to us, a team of three analysts at the insurance company had tried to manually merge these models. It took them 130 man-hours to merge about 25% of the end-to-end process models. The most time-consuming part of the work was to identify common regions manually. Later, we compared the common regions identified by our algorithm and those found manually. Often, the regions identified automatically were smaller than those identified manually. Closer inspection showed that during the manual merge, analysts had determined that some minor differences between the models being merged were due to omissions. Figure 4 shows a typical case (full node names are not shown for confidentiality reasons). Function C appears in

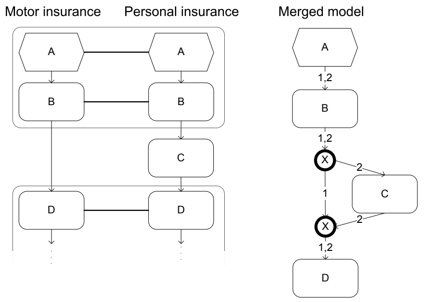

Fig. 4. Fragment of insurance models.

one model but not in the other, and so the algorithm identifies two separate common regions. However, the analysts determined that the absence of C in the motor insurance model was an omission and created a common region with all four nodes. This scenario suggests that when two regions are separated only by one or few elements, this may be due to omissions or minor differences in modeling granularity. Such patterns could be useful in pinpointing opportunities for process model homogenization.

5 Related Work

The problem of merging process models has been posed in [16], [7], [5] and [11]. Sun et al. [16] address the problem of merging block-structured Workflow nets. Their approach starts from a mapping between tasks of the input process models. Mapped tasks are copied into the merged model and regions where the two process models differ, are merged by applying a set of "merge patterns" (sequential, parallel, conditional and iterative). Their proposal does not fulfill the criteria in Section 1: the merged model does not subsume the initial variants and does not provide traceability. Also, their method is not fully automated.

Küster et al. [7] outline requirements for a process merging tool targeted towards version conflict resolution. Their envisaged merge procedure is not automated. Instead the aim is to assist modelers in resolving differences manually, by pinpointing and classifying changes using a technique outlined in [6].

Gottschalk et al. [5] merge pairs of EPCs by constructing an abstraction of each EPC, namely a *function graph*, in which connectors are replaced with edge annotations. Function graphs are merged using set union. Connectors are then restituted by inspecting the annotations in the merged function graph. This approach does not address criteria 2 and 3 in Section 1: the origin of each element cannot be traced, nor can the original models be derived 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. [11] 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 criteria in Section 1. The generic model does not subsume the initial variants and no traceability is provided. Moreover, the approach only works for block-structured process models with AND and XOR blocks.

The problem of process model merging is related to that of integrating multiple views of a process model [12, 8]. A process model view is the instantiation of a process model for a specific stakeholder or business object involved in the process. Mendling and Simon [12] propose, but do not implement, a merging operator that taken to different EPCs each representing a process view, and a mapping of their correspondences, produces a merged EPC. Correspondences can only be defined in terms of events, functions or sequences thereof (connectors and more complex graph topologies are not taken into account). Moreover, a method for identifying such correspondences is not provided. Since the models to be merged represent partial views of a same process, the resulting merged model allows the various views to be executed in parallel. In other words, common elements are taken only once and reconnected to view-specific elements by a preceding AND-join and a subsequent AND-split. However, the use of AND connectors may introduce deadlocks in the merged model. In addition, the origin of the various elements in the merged model cannot be traced.

Ryndina et al. [8] propose a method for merging state machines describing the lifecycle of independent objects involved in a business process, into a single UML AD capturing the overall process. Since the aim is to integrate partial views of a process model, their technique significantly differs from ours. Moreover, the problem of merging tasks that are similar but not identical is not posed. Similarly, the lifecycles to be merged are assumed to be disjoint and consistent, which eases the merge procedure.

For a comparison of our algorithm with work outside the business process management discipline, e.g. software merging and database schema integration, we refer to the technical report [9].

6 Conclusion

The main contribution of this paper is a merging 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. Since the merging algorithm accepts both configurable and non-configurable process models as input, it can be used for multi-way merging. In the case of more than two input process models, we can start by merging two process models, then merge the resulting model with a third model and so on.

We extensively tested the merging operator using process models from practice. The tests showed that the operator can deal with models with hundreds of nodes and that the size of the merged model is, in general, significantly smaller than the sum of the sizes of the original models.

The merging operator essentially performs a union of the input models. In some scenarios, we do not seek the union of the input models, but rather a “digest” showing the most frequently observed behavior in the input models. In future, we plan to define a variant of the merging operator addressing this requirement. We also plan to extend the merging operator in order to deal with process models containing modeling constructs not considered in this paper. For example, BPMN offers constructs such as error handlers and non-interrupting events that are not taken into account by the current merging operator and that would require non-trivial extensions.

Finally, the merging operator relies on a mapping between the nodes of the input models. In this paper we focused on 1:1 mappings. Recent work has addressed the problem of automatically identifying complex 1:n or n:m mappings between process models [18]. Integrating the output of such matching techniques into the merging operator is another avenue 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-Banuelos. Graph matching algorithms for business process model similarity search. In *Proc. of BPM*, volume 5701 of *LNCS*. Springer, 2009.
3. R.M. Dijkman, M. Dumas, L. García-Banuelos, and R. Käärik. Aligning business process models. In *Proc. of EDOC*. IEEE, 2009.
4. P. Fettke and P. Loos. Classification of Reference Models – A Methodology and its Application. In *Information Systems and e-Business Management*, volume 1, pages 35–53, 2003.
5. F. Gottschalk, W. M. P. van der Aalst, and M. H. Jansen-Vullers. Merging event-driven process chains. In *Proc. of CoopIS*, volume 5331 of *LNCS*, pages 418–426. Springer, 2008.
6. J.M. Küster, C. Gerth, A. Förster, and G. Engels. Detecting and resolving process model differences in the absence of a change log. In *Proc. of BPM*, volume 5240 of *LNCS*, pages 244–260. Springer, 2008.
7. J.M. Küster, C. Gerth, A. Förster, and G. Engels. A tool for process merging in business-driven development. volume 344 of *CEUR Workshop Proceedings*, pages 89–92. CEUR, 2008.
8. J.M. Küster, K. Ryndina, and H. Gall. Generation of business process models for object life cycle compliance. In *Proc. of BPM*, volume 4714 of *LNCS*, pages 165–181. Springer, 2007.
9. M. La Rosa, M. Dumas, R. Käärik, and R. Dijkman. Merging business process models (extended version). Technical report, Queensland University of Technology, 2009. `http://eprints.qut.edu.au/29120`.
10. I Levenshtein. Binary code capable of correcting deletions, insertions and reversals. *Cybernetics and Control Theory*, 10(8):707–710, 1966.
11. C. Li, M. Reichert, and A. Wombacher. Discovering reference models by mining process variants using a heuristic approach. In *Proc. of BPM*, volume 5701 of *LNCS*, pages 344–362. Springer, 2009.
12. J. Mendling and C. Simon. Business process design by view integration. In *Proc. of BPM Workshops*, pages 55–64, 2006.
13. T. Pedersen, S. Patwardhan, and J. Michelizzi. WordNet: : Similarity - Measuring the Relatedness of Concepts. In *Proc. of AAAI*, pages 1024–1025. AAAI, 2004.
14. E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
15. M. Rosemann and W. M. P. van der Aalst. A configurable reference modelling language. *Information Systems*, 32(1):1–23, 2007.
16. S. Sun, A. Kumar, and J. Yen. Merging workflows: A new perspective on connecting business processes. *Decision Support Systems*, 42(2):844–858, 2006.
17. B.F. van Dongen, R.M. Dijkman, and J. Mendling. Measuring similarity between business process models. In *Proc. of CAiSE*, volume 5074 of *LNCS*, pages 450–464. Springer, 2008.
18. M. Weidlich, R.M. Dijkman, and J. Mendling. The icop framework: Identification of correspondences between process models. In *Proc. of CAiSE*, 2010.