
MARCELLO LA ROSA
Queensland University of Technology, Australia
MARLON DUMAS
University of Tartu, Estonia
REINA UBA
University of Tartu, Estonia
REMCO DIJKMAN
Eindhoven University of Technology, The Netherlands

This paper addresses the problem of constructing consolidated business process models out of collections of process models that share common fragments. The paper considers the construction of unions of multiple models (called merged models) as well as intersections (called digests). Merged models are intended for analysts who wish to create a model that subsumes a collection of process models – typically representing variants of the same underlying process – with the aim of replacing the variants with the merged model. Digests, on the other hand, are intended for analysts who wish to identify the most recurring fragments across a collection of process models, so that they can focus their efforts on optimizing these fragments. The paper presents an algorithm for computing merged models and an algorithm for extracting digests from a merged model. The merging and digest extraction algorithms have been implemented and tested against collections of process models taken from multiple application domains. The tests show that the merging algorithm produces compact models and scales up to process models containing hundreds of nodes. Furthermore, a case study conducted in a large insurance company has demonstrated the usefulness of the merging and digest extraction operators in a practical setting.

Categories and Subject Descriptors: H.4.1 [Office Automation]: Workflow management; D.2.7 [Distribution, Maintenance, and Enhancement]: Version control
General Terms: Design, Management
Additional Key Words and Phrases: Business process model, variability, model merging, graph matching

1. INTRODUCTION

In the context of company mergers and restructurings, it often occurs that multiple alternative business 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 comparison and integration task 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 setting, this paper proposes and evaluates algorithms for semi-automatically constructing a consolidated model out of a collection of process mod-
els. Specifically, the paper considers the problem of constructing a “union” of a set of process models (herewith called a “merged model”) as well as that of constructing an “intersection” of a collection of process models (herewith called a “digest”).

Merged models are consolidated views on multiple variants of a business process. For example, given two claim handling processes for the same type of incident across two different business units (e.g. two previously separate business units), the merger of these two models is a single claim handling process model that captures the behavior of both variants, and that, after fine-tuning, can be used as a replacement for the variants. In order to fine-tune the merged model, it is important that analysts are able to trace back each element of the process model to one or multiple original variants. Indeed, if part of the merged model is perceived as problematic (e.g. too inefficient from the perspective of cost or resource consumption), analysts need to quickly understand the origin of this part of the model in order to devise ways of fine-tuning the problematic part. This discussion leads us the following requirements for the process model merging:

1. **Behavior-preservation.** The behavior of the merged model should subsume that of the input models. In other words, every behavior captured in the input models should also be captured by the merged model.

2. **Traceability.** 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.

The merge algorithm takes as input a collection of process models and generates a *configurable process model* [Rosemann and van der Aalst 2007]. 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 traceability requirement.

Digests on the other hand, are intended to support the opposite situation: there are multiple process models that share some commonalities, and analysts wish to see which common fragments appear more frequently and how these fragments are related to one another. For example, insurance claim handling processes for different types of incidents (motor claim versus personal injury claim) share common fragments related to verifying policy details, verifying the validity of documents, and verifying invoices. Analysts may wish to understand which fragments occur most frequently across all claim handling process in order to focus their effort on consolidating those parts of the process, for example by factoring them out into shared services within the company in order to benefit from larger resource pools. Digests address this problem by providing a view on the most recurring fragments across a collection of process models. This paper shows how digests at different levels of abstraction can be extracted from a merged model by reusing the same annotations that are used to fulfill the traceability requirement. The input of this digest extraction algorithm is a configurable model (e.g. produced by the merging algorithm) while the output is a regular (non-configurable) process model. The
algorithm also takes an additional parameter that allows analysts to control the level of detail of the digest by stipulating how many times should a fragment recur for it to appear in the digest.

The merging algorithm has been evaluated on process models sourced from different domains. These tests show that the algorithm produces compact models and scales up to process models containing hundreds of nodes. In addition to this quantitative evaluation, we have conducted a case study in which the process model merging tool has been used to aid analysts at a large insurance company to build integrated models of their claim handling processes.

The merging algorithm requires as input a mapping that defines which elements of a process model correspond to which elements of another process model. The construction of this mapping is introduced in Section 2, which also introduces the notion of configurable process model. Section 3 then presents the algorithm for model merging, while Section 4 reports on the implementation and evaluation of this algorithm. Next, Section 5 describes the how the output of the merging algorithm can be used to produce digests at different levels of details. Finally, Section 6 discusses related work and Section 7 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.

**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 $id_G$, $\lambda_G$ and $\tau_G$.

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)onnect 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 Figure 1 are EPCs.

A Configurable EPC (C-EPC) [Rosemann and van der Aalst 2007] 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 is individualized by removing those branches that have been excluded during the configuration of each configurable connector. Model $CG$ in Figure 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 [Rosemann and van der Aalst 2007].

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 Figure 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.

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 Figure 1), 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
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 process 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 \((\text{pid}, l)\) where \( \text{pid} \) is a process graph identifier and \( l \) is the label of node \( n \) in process graph \( \text{pid} \), and \( \eta_G : N_G \rightarrow \{\text{true}, \text{false}\} \) 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 exis\[ \text{[Rosemann and van der Aalst 2007]} \]. However, the differences are purely syntactic: whereas in \[ \text{[Rosemann and van der Aalst 2007]} \], program graph identifiers are only attached to arcs emanating from a configurable split, in this paper we attach process graph identifiers to every arc in a configurable split.\(^1\)

\(^1\)\(\wp\) indicates the powerset.
This syntactic choice makes the definition of the algorithms simpler.

Below, we define some basic notations which we will use in the definitions and algorithms in the rest of the paper.

**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

$$\text{pren}(n) = \{ m \mid (m,n) \in G \}$$

and the postset as

$$\text{post}(n) = \{ m \mid (n,m) \in G \}.$$  

We call an element of the preset predecessor and an element of the postset successor. There is a path $p$ between two nodes $n \in N_G$ and $m \in N_G$, denoted $p = n \leftarrow m$, 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 $(n_i, n_{i+1}) \in G$. We use the notation \{p\} to retrieve the set of nodes in path $p$. If $n \neq m$ and for all $i \in 2, \ldots, k-1$ holds $\tau(n_i) = "c"$, the path $p = n \leftarrow^c m$ 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

$$\text{c\_pren}(n) = \{ m \in N_G \mid m \leftarrow^c n \}$$

and is called the transitive preset of $n$ via connector chains. Similarly, $\text{c\_post}(n) = \{ m \in N_G \mid n \leftarrow^c m \}$ 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 [Bunke 1997]. We use this matching score as it performed well in several empirical studies [van Dongen et al. 2008; Dijkman et al. 2009; Dijkman et al. 2009]. 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 [Levenshtein 1966], and a linguistic similarity measure, based on the Wordnet::Similarity package [Pedersen et al. 2004] (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 Figure 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 “transport” or between “process” 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 [Rahm and Bernstein 2001].

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 $M : N_{G_1} \rightarrow N_{G_2}$ 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:

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

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 Figure 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 $\text{dom}(M)$ denotes the domain of $M$ and $\text{cod}(M)$ denotes the codomain of $M$. Let also $0 \leq w_{\text{subn}} \leq 1$, $0 \leq w_{\text{skipn}} \leq 1$ and $0 \leq w_{\text{skipe}} \leq 1$ be the weights that we assign to substituted nodes, inserted or deleted nodes and inserted or deleted edges, respectively, and let $\text{Sim}(n,m)$ 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 $\text{subn}$, inserted or deleted nodes, denoted $\text{skipn}$, substituted edges, denoted $\text{sube}$, and inserted or deleted edges, denoted $\text{skipe}$, are defined as follows:

$$\text{subn} = \text{dom}(M) \cup \text{cod}(M)$$
$$\text{sube} = \{(a,b) \in E_1 | (M(a), M(b)) \in E_2\} \cup \{(a',b') \in E_2 | (M^{-1}(a'), M^{-1}(b')) \in E_1\}$$

The fraction of inserted or deleted nodes, denoted $f_{\text{skipn}}$, the fraction of inserted or deleted edges, denoted $f_{\text{skipe}}$, and the average distance of substituted nodes, denoted $f_{\text{subn}}$, are defined as follows.

$$f_{\text{skipn}} = \frac{|\text{skipn}|}{|N_1| + |N_2|}$$
$$f_{\text{skipe}} = \frac{|\text{skipe}|}{|E_1| + |E_2|}$$

$$f_{\text{subn}} = \frac{2 \cdot \sum_{(n,m) \in M} 1.0 - \text{Sim}(n,m)}{|\text{subn}|}$$

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

$$1.0 - \frac{w_{\text{skipn}} \cdot f_{\text{skipn}} + w_{\text{skipe}} \cdot f_{\text{skipe}} + w_{\text{subn}} \cdot f_{\text{subn}}}{w_{\text{skipn}} + w_{\text{skipe}} + w_{\text{subn}}}$$

For example, in Figure 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 $1.0 - \frac{\frac{7}{12} + \frac{1}{12} + \frac{2.05 + 2.05}{12}}{3} = 0.64$.

Definition 5 gives the matching score of a given mapping. To determine the matching score of two business process graphs, we must construct all possible mappings and find the one with the highest matching score. Various heuristics exist to efficiently find the mapping with the highest matching score [Dijkman et al. 2009].

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. After converting each input process
Business Process Model Merging

Graph into a configurable process graph, we can proceed to merge the configurable process graphs. We first present the basic merge algorithm. Then we show that the algorithm satisfies its requirements. Next, we show how to improve the mapping in order to avoid entangled nodes in the merged process graph. Finally, we discuss a set of reduction rules to simplify the merged process graph.

3.1 Basic Merging Algorithm

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 Figure 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 Figure 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 Figure 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 original model they come from. For example, we do not merge event “Deliveries need to be planned” in Figure 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.

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 $G_1$, as shown in the two graphs of Figure 2, where node $B$ is a sink for $G_2$ but not for $G_1$, and node $D$ is a source 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...
Algorithm 1: Merge

1 function Merge(Graph G1, Graph G2, Mapping M)
2 init
3 Mapping mcr, Graph CG
4 begin
5 CG ← G1 ∪ G2 \ (G2 ∩ sube)
6 foreach (x, y) in CG ∩ sube do
7 αCG(x, y) ← αG1(x, y) ∪ αG2(M(x), M(y))
8 end
9 foreach n in NCG ∩ subn do
10 γCG(n) ← γG1(n) ∪ γG2(M(n))
11 end
12 foreach mcr in MaximumCommonRegions(G1, G2, M) do
13 FG1 ← {x ∈ dom(mcr) | • x ∩ dom(mcr) = ∅ ∨ • M(x) ∩ cod(mcr) = ∅}
14 foreach fG1 in FG1 such that | • fG1| = 1 and | • M(fG1)| = 1 do
15 pfG1 ← Any(• fG1), pfG2 ← Any(• M(fG1))
16 xj ← new Node(“c”, “xor”, true)
17 γ(xj) = {(Pid(G1), “xor”), (Pid(G2), “xor”)}
18 CG ← (CG \ {(pfG1, xj), (pfG2, xj)}) ∪ {(pfG1, xj), (pfG2, xj), (xj, fG1)}
19 αCG(pfG1, xj) ← αG1(pfG1, fG1)
20 αCG(pfG2, xj) ← αG2(pfG2, fG2)
21 αCG(xj, fG1) ← αG1(pfG1, fG1) ∪ αG2(pfG2, fG2)
22 end
23 LG1 ← {x ∈ dom(mcr) | • x ∩ dom(mcr) = ∅ ∨ • M(x) ∩ cod(mcr) = ∅}
24 foreach lG1 in LG1 such that | lG1 • | = 1 and | M(lG1) • | = 1 do
25 slG1 ← Any(lG1 •), slG2 ← Any(M(lG1) •)
26 xs ← new Node(“c”, “xor”, true)
27 γ(xs) = {(Pid(G1), “xor”), (Pid(G2), “xor”)}
28 CG ← (CG \ {(lG1, slG1), (lG2, slG2)}) ∪ {(xs, slG1), (xs, slG2), (lG1, xs)}
29 αCG(xs, slG1) ← αG1(lG1, slG1)
30 αCG(xs, slG2) ← αG2(lG2, slG2)
31 αCG(lG1, xs) ← αG1(lG1, slG1) ∪ αG2(lG2, slG2)
32 end
33 end
34 CG ← MergeConnectors(M, CG)
35 return CG
36 end
Algorithm 2: Maximum Common Regions

1 function MaximumCommonRegions(Graph $G_1$, Graph $G_2$, Mapping $M$)
2     {Node} visited $\leftarrow \emptyset$, {Mapping} MCRs $\leftarrow \emptyset$
3 begin
4     while exists $c \in \text{dom}(M)$ such that $c \notin \text{visited}$ do
5         {Node} mcr $\leftarrow \emptyset$
6         {Node} tovisit $\leftarrow \{c\}$
7         while tovisit $\neq \emptyset$ do
8             $c \leftarrow \text{dequeue}(\text{tovisit})$
9             mcr $\leftarrow mcr \cup \{c\}$
10            visited $\leftarrow \text{visited} \cup \{c\}$
11            foreach $n$ in $\text{dom}(M)$ such that $((c, n) \in G_1 \text{ and } (M(c), M(n)) \in G_2)$ or
12              $((n, c) \in G_1 \text{ and } (M(n), M(c)) \in G_2)$ and $n \notin \text{visited}$ do
13                  enqueue(\text{tovisit}, n)
14         end
15     end
16     MCRs $\leftarrow \text{MCRs} \cup \{mcr\}$
17 end
18 return MCRs
19 end

Fig. 2. An example where a node is a source (sink) in one graph but not in the other.
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 $M(lG_1)$ 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 Figure 3 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 connectors that we need to add, function $Pid$ to retrieve the identifier of a graph when building the annotations for these new connectors, and function $Any$ to extract the element of a singleton set.

In Figure 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. For example, in the merged graph of Figure 2 node $B$ has been reconnected to its successors in $G_1$ and $G_2$ via a configurable XOR-join, even if the successor of $B$ in $G_1$ is inside the region.

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 Figure 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 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 Figure 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”, thus obtaining the merged graph \( CG \) in Figure 1.

With reference to Algorithm 1, we observe that if nothing is done in both the foreach clauses, \( G_1 \) and \( G_2 \) are equal except at most for intermediate connectors which are aligned by function MergeConnectors (Algorithm 3).

### Algorithm 3: Merge Connectors

```plaintext
function MergeConnectors(Mapping M, {Edge} CG)
init
   {Node} S \leftarrow \emptyset, {Node} J \leftarrow \emptyset
begin
   foreach \( c \) in \( \text{dom}(M) \) such that \( \tau(c) = "c" \) do
      \( S \leftarrow \{ x \in M(c) \cup \mid x \notin \text{cod}(M) \} \)
      \( J \leftarrow \{ x \in M(c) \cup \mid x \notin \text{cod}(M) \} \)
      \( CG \leftarrow (CG \cup \bigcup_{x \in S} \{(M(c), x)\} \cup \bigcup_{x \in J} \{(x, M(c))\}) \cup \bigcup_{x \in S} \{(c, x)\} \cup \bigcup_{x \in J} \{(x, c)\} \)
      \( \alpha_{CG}(c, x) \leftarrow \alpha_{G_1}(M(c), x) \)
   end
   foreach \( x \in J \) do
      \( \alpha_{CG}(x, c) \leftarrow \alpha_{G_2}(x, M(c)) \)
   end
   if |S| > 0 or |J| > 0 then
      \( \eta_{CG}(c) \leftarrow \text{true} \)
   end
   if \( \lambda_{G_1}(c) \neq \lambda_{G_2}(M(c)) \) then
      \( \lambda_{CG}(c) \leftarrow "or" \)
   end
end
return CG
```

The complexity of the algorithm for merging connectors is linear on the num-
ber of connectors, which is bounded by the number of edges. The algorithm for 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. On top of this, we need to consider the complexity of computing the matching score of the two input graphs. For example, if we use a greedy algorithm [Dijkman et al. 2009], the complexity is cubic on the number of nodes of the largest graph, which dominates the complexity of the merging algorithm.

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.

3.2 Properties of the Algorithm

In the introduction we stated that the algorithm should satisfy 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.

It is easy to see that the second requirement is met since a configurable business process graph relates each of its elements to the element from which it was derived by means of functions $\gamma$ and $\alpha$ in Definition 2. Below, we sketch a proof that the algorithm fulfills the first requirement.

**Proposition 1.** Let $CG$ be the configurable process graph produced by Algorithm 1 when given process graphs $G_1$ and $G_2$ as input. Any execution trace of $G_1$ or $G_2$ is also an execution trace of $CG$.

**Proof.** We sketch the proof for graph $G_1$, since the proof for $G_2$ is identical. Let $e_1e_2\ldots e_n$ be an execution trace of $G_1$ represented as a sequence of edges. First, we make the following observations:

(1) According to line 5 of the algorithm, every edge in $G_1$ is also an edge of $CG$, since the set of edges of $CG$ is initialized to be equal to the union of the set of edges of the input graphs, and subsequently, the algorithm only adds edges to $CG$.

(2) According to lines 18 and 28 of the algorithm, the merged graph $CG$ may also contain edges of the form $(n, c)$ and $(c, n)$ where $c$ is a new configurable XOR connector added during the merge and $n$ is a node of $CG$.

Thus, every edge $e_i$ that appears in this execution trace of $G_1$ is also an edge of $CG$, except edges that connect a node from a maximum common region to a node outside
of that maximum common region (lines 18 and 28). Let \((n, m)\) be such an edge at the boundary of a common region. According to lines 18 and 28 this is replaced by two edges: \((n, c)\) and \((c, m)\), where \(c\) is an XOR connector. Consequently, for each edge \(e = (k, l)\) in the execution trace \(e_1e_2\ldots e_n\) there are two possible cases, either:

1. it appears as edge \((k', l')\) in the merged graph \(CG\), connecting nodes \(k'\) and \(l'\) that are derived from nodes \(k\) and \(l\) in \(G_1\), in which case it can be traversed from \(k'\) to \(l'\) in an execution trace of \(CG\) as it could be in the original execution trace; or

2. it appears in the merged graph \(CG\), as a pair of edges \((k', c)\) and \((c, l')\), also connecting nodes \(k'\) and \(l'\) that are derived from nodes \(k\) and \(l\) in \(G_1\), in which case it can also be traversed from \(k'\) to \(l'\) in an execution trace of \(CG\) as it could be in the original execution trace, because \(c\) is an XOR connector which is a silent step (i.e. does not perform any visible action).

It remains to be shown that nodes \(k'\) and \(l'\) subsume the behavior of nodes \(k\) and \(l\) from which they are derived. A node \(k'\) or \(l'\) has either unchanged behavior with respect to the node \(k\) or \(l\) from which it was derived, or it is a merged connector according to Algorithm 3. In the latter case, the node type, and therefore its behavior, is either the same as the type of the node from which it was derived, or it is an OR connector that was derived from merging an XOR connector with an AND connector. However, the behavior of an OR connector subsumes that of an XOR and an AND connectors, therewith preserving the subsumption relation. Consequently, each edge in an execution trace of \(G_1\) can also be traversed in an execution trace of \(CG\).

3.3 Entanglement in Merged Models

The algorithm that we developed chooses to always merge the identified common regions. This, however, does not necessarily lead to an optimal solution in terms of the readability of the merged graph. Figure 4 illustrates this point. Here two models with common regions \(X-X', A-A', B-B'\) and \(Z-Z'\) are merged, but the resulting configurable graph \(CG\) contains an “entanglement” that gives the impression that \(A\) and \(B\) are in a cycle. This cycle only exists in the configurable graph and disappears in each individualization thereof because each of the two edges that introduces the cycle is annotated either with one of the originating process graphs or the other, but not both. During individualization, one of these two edges will be removed and the cycle will not appear in the individualized model. This feature makes the configurable graph confusing and affects its readability. Moreover, the resulting graph is not as compact as it could be.

An alternative merged model (namely \(CG'\)) is shown in Figure 4. This alternative is obtained if we choose not to merge nodes \(A-A'\) and \(B-B'\). This alternative is arguably easier to read and has less nodes than \(CG\).

If we analyze this entanglement pattern further, we observe that the underlying cause is not in the merge algorithm, but rather in the matching of nodes. Specifically, node \(A\) in the first graph is mapped to a node in the second graph that comes “after” the node to which node \(B\) is mapped, yet \(A\) comes “before” \(B\) in the first graph. In other words, the mapping is not consistent with the order of the nodes in the input graphs. Therefore, we want to avoid mappings that include two pairs.
of nodes \((A, A')\) and \((B, B')\), such that if \(A\) is merged with \(A'\) and \(B\) with \(B'\), the merged graph will contain a cyclic path that did not exist in any of the input graphs.

To avoid this situation, we discard from the mapping any two pairs of nodes \((A, A')\) and \((B, B')\) such that the following conditions are fulfilled:

— The graph obtained by computing the union of \(G_1\) and \(G_2\) and adding an undirected edge (i.e. a two-way arc) from \(A\) to \(A'\) and another from \(B\) to \(B'\), contains a cyclic path that traverses \(A, A', B', B\) (in this order) or \(A', A, B, B'\).

— There is no cyclic path traversing \(A\) and \(B\) in \(G_1\).

— There is no cyclic path traversing \(A'\) and \(B'\) in \(G_2\).

Whenever we find two pairs of nodes fulfilling the above conditions, these pairs are removed from the mapping. For example, by computing the mapping between \(G_1\) and \(G_2\) in Figure 4 we obtain the set of mapped pairs \{(X, X'), (A, A'), (B, B'), (Z, Z')\}. Then, we compare the mapped pairs of nodes and discard \((A, A')\) and \((B, B')\) since they fulfill the above conditions. So the final mapping will only contain \((X, X')\) and \((Z, Z')\). The merged model obtained from this latter mapping is \(CG_2\).

The entanglement problem can also occur between connectors. Figure 5 shows such an example.

---

**Fig. 4.** An entanglement in a merged graph.

**Fig. 5.** An example of entanglement with connectors.
Here the best matching score is yielded by mapping the OR-split $a$ in $G_1$ with the AND-split $d$ in $G_2$, and the XOR-join $b$ in $G_1$ with the XOR-join $c$ in $G_2$. The resulting graph $CG_A$ suffers from the same entanglement problem: there is a cycle between nodes $ad$ and $bc$. Again, the entanglement can be avoided simply by removing the pairs $(a, d)$ and $(b, c)$ from the mapping. The resulting merged graph is $CG_B$. Although $CG_B$ does not have cycles, it is less compact than $CG_A$. In the next subsection we will show how to simplify process graphs like $CG_B$ by applying reduction rules.

3.4 Reduction Rules

After merging two process graphs, we can simplify the resulting graph by applying a set of reduction rules. These rules are designed to eliminate “unnecessary” connectors or edges introduced by the merging algorithm. The 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, that may have been generated after applying the first two rules. The rules are applied until a process graph cannot be further reduced.

3.4.1 Merge consecutive splits/joins. Function $\text{MergeConsecutiveConnectors}$ (Algorithm 4) merges any two consecutive splits (joins) into a single split (join) connector. Since the idea is to eliminate unnecessary connectors introduced by the merge algorithm (and not to eliminate connectors already present in the input process graphs), we only apply this rule when one of the two connectors is a configurable XOR that has been added by Algorithm 1. The other connector will necessarily be an original connector, i.e. a connector that existed in one of the input graphs. This condition is checked using function $\text{IsAdded}$, which takes an edge as input and returns true if and only if the edge’s source or target is a configurable XOR added by the merge algorithm.

In order to merge two consecutive splits $m$ and $n$, we first remove all edges connected to $n$ and reconnect each successor $x$ of $n$ to $m$ via an edge $(m, x)$. Naturally, this edge is not added if $x$ was already a successor of $m$. Next, for all $x$, we set the label of the edge $(m, x)$ to be the union of its label and that of the edge being removed $(n, x)$, so that no information about the original variants is lost in the reduction. Then we update the annotation of $m$ via the “⊕” operator. This operator assigns to $m$ the annotation of the original connector between $m$ and $n$, to which it adds a pair (pid,“xor”) for all process identifiers that do not appear in that annotation. Pairs (pid,“xor”) are added because the label of the other connector being merged is always XOR. Finally, we make $m$ configurable and if there is a mismatch between its label and that of $n$, we change its label to “or”. The case of two consecutive joins is symmetric.

Figure 6 shows the application of this rule to graph $CG_B$ of Figure 5. This graph has two consecutive splits, $p$ and $a$, and two consecutive joins, $b$ and $s$, where $p$ and $s$ are two configurable XOR connectors that were added during the merge. By merging these two pairs of connectors we obtain graph $CG'_B$, shown in the right-hand side of Figure 6. For example, we can observe that connector $pa$ bears annotation $(1,“or”), (2,“xor”) as a result of adding $(2,“xor”) to the annotation of the original OR connector $a$ which did not contain process identifier 2.
Algorithm 4: Merge Consecutive Connectors

function MergeConsecutiveConnectors(\{\text{Edge} \} CG)
begin
foreach (m, n) in CG such that $\tau(m) = \tau(n) = \text{“c”}$ and IsAdded((m, n)) = true do
  if $|\bullet m| > 1$ and $|\bullet n| > 1$ then
    CG = (CG \ {((m, n))} \cup \bigcup_{x \in \bullet n} \{(n, x)\}) \cup \bigcup_{x \in \bullet m} \{(m, x)\}
    foreach x in $\bullet n$ do
      $\alpha(m, x) \leftarrow \alpha(m, x) \cup \alpha(n, x)$
    endforeach
    $\gamma(m) \leftarrow \gamma(m) \oplus \gamma(n)$
    $\eta(m) \leftarrow \text{true}$
    if $\lambda(m) \neq \lambda(n)$ then
      $\lambda(m) \leftarrow \text{“or”}$
    end
  end
  else if $|\bullet m| > 1$ and $|\bullet n| > 1$ then
    CG = (CG \ {((m, n))} \cup \bigcup_{x \in \bullet m} \{(x, m)\}) \cup \bigcup_{x \in \bullet m} \{(x, n)\}
    foreach x in $\bullet m$ do
      $\alpha(x, n) \leftarrow \alpha(x, n) \cup \alpha(x, m)$
    endforeach
    $\gamma(n) \leftarrow \gamma(m) \oplus \gamma(n)$
    $\eta(n) \leftarrow \text{true}$
    if $\lambda(m) \neq \lambda(n)$ then
      $\lambda(n) \leftarrow \text{“or”}$
    end
  end
end
return CG
end

We observe that the process graph prior to applying this reduction rule subsumes the process graph after the reduction because:

—Any path from a predecessor of $m$ to a successor of $n$ is still present in the reduced graph – it just contains one connector less.
—The behavior of the merged connector always subsumes that of the two connectors being merged (cf. lines 11, 12 and 22, 23 in Algorithm 6).

3.4.2 Remove redundant transitive edges. A redundant transitive edge is an edge whose source node and target node are also connected via an alternative path made of a chain of consecutive connectors. Thus, the source of a redundant edge is a split and its target is a join. Moreover, all edges emanating from an intermediate split in the connector chain that lead to nodes outside the connector chain, and all edges incoming to an intermediate join from a node outside the connector chain, must not bear any process identifier of the redundant edge.
Function \textit{RemoveRedundantTransitiveEdges} (Algorithm 5) removes all redundant transitive edges from a process graph. For all pairs of nodes \( m \) and \( n \) where \( m \) is a split, \( n \) is a join and \((m,n)\) is a redundant transitive edge, this algorithm first removes \((m,n)\). Next, it sets the annotation of each edge \((x,y)\) in the connector chain to be the union of the edge’s annotation and that of \((m,n)\). It then makes each intermediate connector in the connector chain configurable and merges its annotation with an “xor” for all process identifiers in the annotation of \((m,n)\). We observe that either the annotation of an intermediate connector does not have any process identifier in the annotation of \((m,n)\) (and so adding the process identifiers of \((m,n)\) to that annotation is safe), or the intermediate connector is an XOR. In fact, if it were a AND or OR connector, there would exist at least one edge linking that connector to a node not in the connector chain and containing of the process identifiers of the redundant edge \((m,n)\) – thereby violating the precondition for removing redundant transitive edges. After this step, if the connector’s label is not “xor”, the algorithm changes it to “or” in order to ensure that the reduced process graph subsumes the original one.

Figure 7 shows the application of this rule to graph \( CG_B' \) obtained after merging the consecutive connectors in Figure 6. In this graph there are three redundant transitive edges: \((pa,bs)\), \((pa,z)\) and \((y,bs)\), highlighted with a ticker line in the picture. Assume we start by removing edge \((pa,bs)\). This entails adding process identifier “1” to the annotations of the three edges in the connector chain between \( pa \) and \( bs \). We also need to make the intermediate connectors \( c \) and \( d \) configurable and we need to add \((1,\text{"xor"})\) to their annotations. Since the label of \( d \) is “and”, we also need to change it to “or”. After this step, edge \((pa,z)\) is no longer redundant, since now edge \((d,bs)\) contains “1” in its annotation and is not part of the connector chain. On the other hand, \((y,bs)\) is still a redundant edge. We remove it and obtain graph \( CG_B'' \), shown in the middle of Figure 7.

If we first removed \((y,bs)\), we could then remove \((pa,bs)\) only, thus obtaining the same graph \( CG_B' \). However, if we reduced graph \( CG_B' \) by first removing \((pa,z)\), we

\footnote{trivial annotations are not depicted}


Algorithm 5: Remove Redundant Transitive Edges

1 function RemoveRedundantTransitiveEdges({Edge} CG)
2 begin
3 foreach (m, n) in CG such that |m •| > 1 and |• n| > 1 and exists a path
4 p ∈ CG such that p = m ↣ n, |{p}| > 2 and for all connectors
5 c ∈ {p} \ {m, n} there not exists a node x ∈ c • \ {p} such that
6 α(c, x) ∩ α(m, n) ≠ ∅ or x ∈ c \ {p} such that α(x, c) ∩ α(m, n) ≠ ∅ do
7 CG ⇐ CG \ {{m, n}}
8 foreach (x, y) in CG such that {x, y} ∈ {p} do
9 α(x, y) ≅ α(x, y) ∪ α(m, n)
10 end
11 foreach c in {p} \ {m, n} do
12 η(c) ← true
13 γ(c) ← γ(c) ∪ \{pid∈α(m,n)\} \{(pid, “xor”)\}
14 if λ(c) ≠ “xor” then
15 λ(c) ← “or”
16 end
17 end
18 return CG
19 end

Fig. 7. Removing redundant transitive edges from graph \(CG_B\) of Figure 6.

would obtain a different graph, \(CG''_{B2}\) (shown on the right-hand side of Figure 7),
where there is no further edge that can be removed. In fact in this graph \((pa, bs)\)
and \((y, bs)\) are no longer redundant since edge \((d, z)\) now contains identifier 1
in its annotation. Although \(CG''_{B2}\) is less compact than \(CG''_{B}\), both graphs yield the
same set of traces (i.e. they have equal behavior).

We observe that the reduced graph subsumes the behavior of the unreduced graph
because any redundant edge appearing in an execution trace of the unreduced graph
can be replaced by the edges in the alternative connector chain, which only traverse
connectors (i.e. silent steps). Moreover, any connector in the connector chain whose
label has been changed to “or”, subsumes the behavior of the original connector in the unreduced graph.

3.4.3 Remove trivial connectors. A trivial connector is one that only has one incoming and one outgoing edge. Such connectors are clearly spurious and can be removed without any impact on the behavior. Function RemoveTrivialConnectors (Algorithm 6) removes all trivial connectors in a process graph. Before removing a trivial connector, the algorithm checks that it is a configurable connector. This may be a configurable XOR introduced by the merge algorithm, or a trivial configurable connector generated by applying MergeConsecutiveConnectors or RemoveRedundantEdges.

The algorithm removes a trivial connector $m$ by deleting its incoming edge from the single predecessor $pm$ and its outgoing edge to the single successor $sm$. Next, it reconnects $pm$ with $sm$ via a new edge, and sets the annotation of this edge to the annotation of the incoming edge being removed. Here, we observe that the annotation of the incoming edge and that of the outgoing edge of a trivial connector always coincide.

Figure 8 shows the application of this reduction rule to graph $CG_B''$ obtained after removing the redundant edges in Figure 7. In this graph we have two trivial connectors: $y$ and $bs$. After removing them, we obtain graph $CG_B'''$ which cannot be further reduced. This graph has the same size as graph $CG_A$ of Figure 5 but does not suffer from the entanglement problem. The two initial graphs $G_1$ and $G_2$ of Figure 5 can be derived by configuring $CG_B'''$ for the process identifier “1”, respectively, “2”.

Algorithm 6: Remove Trivial Connectors

1 function RemoveTrivialConnectors({Edge} CG)
2 begin
3     foreach $m$ in $N_{CG}$ such that $\tau(m) = "c"$ and $\bullet m = \bullet m$ and $\eta(m) =$
4         true do
5         $pm = \text{Any}(\bullet m), sm = \text{Any}(m\bullet)$
6         $CG \leftarrow (CG \setminus \{(pm, m), (m, sm)\}) \cup \{(pm, sm)\}$
7         $\alpha(pm, sm) \leftarrow \alpha(pm, m)$
8     end
9 return CG
10 end

We observe that the behavior of the reduced graph is subsumed by that of the unreduced one because any trace of the unreduced graph that traversed the trivial connector has an equivalent trace in the reduced graph where the trivial connector is simply skipped. Moreover, since the removed connector did not have any splitting/joining behavior, it does not create additional traces not present in the unreduced graph. Hence, we can even conclude that the unreduced and the reduced graph have the same set of traces.
3.5 Merging non-control-flow elements

In this section we discuss how the merge algorithm can be extended to deal with process models that contain information about roles and objects. A role (e.g. Clerk or Manager) is a class of organizational resources that is able to perform certain types of activities. Objects are information artifacts (e.g. files) or physical artifacts (e.g. paper documents or production materials) of an enterprise that are used (input objects) or produced (output objects) by a process activity. Several process modeling languages such as BPMN, extended EPCs and UML Activity Diagrams support these concepts to a different extent. For a comprehensive meta-model of business processes incorporating roles and objects we refer to [La Rosa et al. 2010].

In order to deal with non control-flow elements during process merging, we extend the concept of process graph by defining a notion of *multi-perspective* process graph. A multi-perspective process graph is a process graph where each node can be assigned a set of non-control-flow elements via an edge, each element being a pair (type, label). For example, an element can have type ‘role’ and label “Supply officer” or type ‘output object’ and label “Bill of lading”, but also type ‘risk’ and label ‘System failure’. This simple extension allows us to capture non-control-flow information in a generic (language-independent) manner and to “carry on” this information during process merging.

In order to merge multi-perspective process graphs, we proceed as follows. First, we annotate each edge linking a node and a non-control-flow element with the process identifier of the input graph, in the same way as we do for control-flow edges. Then we apply the merge algorithm (and reduction rules) as defined above. Next, we associate each merged model with all its non-control-flow elements in \( G_1 \) and \( G_2 \) via their original edges. In doing so, we merge a non-control-flow element in \( G_1 \) with one in \( G_2 \) if they have the same type and if their label similarity is above a threshold. We connect a merged non-control-flow element to the merged node via an edge labeled with the union of the process identifiers in the two original edges in \( G_1 \) and \( G_2 \). Similar to control-flow nodes, if the labels of two elements being merged were different, we add an annotation to the merged element recording the original label for each process identifier.

For example, Figure 9 shows how the roles and objects associated with function...
"Transportation planning and processing" from graph $G_1$ of Figure 1, are merged with those associated with its matched function "Transporting" from $G_2$. Assuming a label threshold of 0.5, we merge role "Supply officer" from $G_1$ with role "Junior supply officer" from $G_2$, and we make the union of all other roles and of the input/output objects.

![Graphs $G_1$, $G_2$, and $G_G$ showing the merging process.]

**Fig. 9.** Merging process models with roles and objects.

4. EVALUATION

The process merging algorithm has been implemented as a tool, namely Process Merger, that is freely available as part of the Synergia toolset (see: http://www.processconfiguration.com). The tool accepts two (configurable) EPCs represented in the EPML format and suggests a mapping between the two models. Users can select different matching algorithms (see [Dijkman et al. 2009] for a list of matching algorithms) and they can configure the parameters of the selected matching algorithm. After the user has reviewed and validated the resulting mapping, the tool produces a configurable EPC (encoded in EPML format). This merged model is simplified by applying the reduction rules, and a digest can be generated based on a given frequency threshold.

The implementation of the algorithm has also been integrated into the AProMoRe [La Rosa et al. 2009] platform – a process model repository toolset (see: http://code.google.com/p/apromore/). AProMoRe allows users to store and edit process models in a variety of languages (EPCs, BPMN, YAWL and BPEL). This is made possible via an internal, canonical representation of process models that captures a range of modeling constructs found across multiple process modeling languages, including constructs to represent resource and object information. From the AProMoRe’s repository, users can choose a set of process models to be merged. The merged model can be stored in the repository or exported in any process modeling language supported by the AProMoRe platform. Digests can be subsequently extracted from the merged model.

Using the implementation of the algorithm, we conducted experiments in order to evaluate (i) the size of the models produced by the merge operator, and (ii) the scalability of the merge operator. Furthermore, we conducted a case study to evaluate the usefulness of the merge algorithm in practice.
4.1 Size of merged models

Size is a key factor affecting the understandability of process models [Mendling et al. 2010] 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 be 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. In these and in the following tests, we used a greedy algorithm from [Dijkman et al. 2009] to identify the best matching score between input models, since its computational complexity is much lower than that of an exhaustive algorithm, while having a high precision. 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

\[ CF(G_1, G_2) = \frac{|CG|}{|G_1| + |G_2|}, \]

where \( CG = \text{Merge}(G_1, G_2) \). 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 1.3

Table I summarizes the 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. The last three columns show the size and compression factor of the merged model after applying the reduction rules, and the compression factor after removing from the mapping those nodes that generate entanglement. The table shows that the reduction rules improve the compression factor (average of 69% vs. 76%), 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 10 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. From these experiments we also observed that the

---

3In file compression, the compression factor is defined as \( 1 - \frac{|CG|}{|G_1| + |G_2|} \), but here we use the reverse in order to compare this factor with the matching score.

ACM Journal Name, Vol. V, No. N, Month 20YY.
impact of not merging nodes that generate entanglement on the compression factor is negligible (the average compression factor increases from 68.76% to 69.43%).

<table>
<thead>
<tr>
<th></th>
<th>Size 1</th>
<th>Size 2</th>
<th>Size merged</th>
<th>Compression after reduction</th>
<th>Compression without entanglements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>Max</td>
<td>130</td>
<td>130</td>
<td>194</td>
<td>1.17</td>
<td>1.06</td>
</tr>
<tr>
<td>Average</td>
<td>22.07</td>
<td>24.31</td>
<td>33.9</td>
<td>0.76</td>
<td>0.69</td>
</tr>
<tr>
<td>Std dev</td>
<td>20.95</td>
<td>22.98</td>
<td>30.35</td>
<td>0.15</td>
<td>0.13</td>
</tr>
</tbody>
</table>

Table I. Size statistics of merged SAP reference models.

Fig. 10. Correlation between matching score of input models and compression factor.

4.2 Scalability

We also conducted tests with large process models in order to assess the scalability of the merging operator. We considered four model pairs. The first three pairs capture a process for handling motor incident and personal injury claims at Suncorp-Metway Ltd, 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 is for 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 (vehicle vs. personal injury).

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.

Table II shows the sizes of the input models, their matching score, the total execution times, 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 32 bit and Oracle Java Virtual Machine version
1.6 (with 512MB of allocated memory). The total execution times include the time taken to read the models from disk, to match them and to merge them. The merge time is also indicated separately between brackets.

<table>
<thead>
<tr>
<th>Pair #</th>
<th>Size 1</th>
<th>Size 2</th>
<th>Match score</th>
<th>Total time (merge time) in msec.</th>
<th>Size merged</th>
<th>Compression</th>
<th>Merged compression after reduction</th>
<th>Compression after reduction</th>
<th>Compression without entanglements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>339</td>
<td>357</td>
<td>0.84</td>
<td>7409 (79)</td>
<td>486</td>
<td>0.70</td>
<td>474</td>
<td>0.68</td>
<td>0.75</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>78</td>
<td>0.56</td>
<td>78 (0)</td>
<td>88</td>
<td>0.88</td>
<td>87</td>
<td>0.87</td>
<td>0.88</td>
</tr>
<tr>
<td>3</td>
<td>468</td>
<td>211</td>
<td>0.62</td>
<td>3693 (85)</td>
<td>641</td>
<td>0.94</td>
<td>624</td>
<td>0.92</td>
<td>0.93</td>
</tr>
<tr>
<td>4</td>
<td>198</td>
<td>191</td>
<td>0.82</td>
<td>853 (20)</td>
<td>290</td>
<td>0.75</td>
<td>279</td>
<td>0.72</td>
<td>0.78</td>
</tr>
</tbody>
</table>

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

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 II also shows the compression factors. Pairs 2 and 3 have a poor compression factor (lower is better). However, this can be explained by the fact that these pairs of models have a low matching score (around 0.56).

From Table II we can also observe an increase in the compression factor after removing nodes that generate entanglements. This increase is significant in pair 1 (compression factor increases from 68% to 75%) and in pair 4 (from 72% to 78%). Although there is an apparent correlation between the increase in compression factor and the matching score (pairs 1 and 2 have the highest matching scores out of the four pairs), this increase is due to the type of entanglement in these models. These models have entire regions, and not single nodes, entangled with each other. Figure 11 shows an extract of pair 4 (the land development models).

In the South Australia variant, common regions $a_1$, $b_1$ and $c_1$ are sequential, whereas in the Western Australia variant, region $a_2$ is in parallel with $c_2$, and both regions precede $b_2$. This situation generates an entanglement in the merged model (shown in the middle of Figure 11), which highly affects the readability of the model. We can see this by comparing this model with the merged model after removing entanglements (shown in the right-hand side of Figure 11), where regions $b_1$ and $b_2$ are not merged, and regions $c_1$ and $c_2$ are partly merged. Despite the first model being slightly more compact (279 nodes vs. 304 nodes), the second model is arguably more structured.

4.3 Case study

To evaluate the usefulness of the merge algorithm in an industrial setting, we conducted a case study with Suncorp-Metway Ltd (Suncorp for short). Suncorp is one of Australia’s top-25 listed companies, providing a range of banking and insurance products. Suncorp has an established in-house commitment to increasing efficiency and effectiveness of its business operations, particularly through continuous process improvement. Over the last years, the company has accumulated over 6000 business process variants after a series of mergers and acquisitions. Maintaining such a large amount of variants has proved to be costly, both due to the high costs...
of developing and maintaining supporting software systems, as well as the inherent costs of measuring, monitoring and optimizing the performance of all process variants. Thus, the company has embarked in an effort to consolidate their process variants for the insurance segment. As part of this effort, the authors of this paper were engaged to help in matching and merging some of the key insurance-related process models in the company’s model repository.

The engagement started with three pairs of process models for claims handling (the ones discussed in Section 4.2). When these process models were given to us for semi-automated merging, a team of three analysts at Suncorp had already tried to manually merge them. However, it had taken them 130 man-hours to merge about 25% of the process models. The most time-consuming part of the work was to identify common (or similar) regions manually.

To speed up the merging effort, we started by running the algorithm for identifying common regions on the three pairs of process models. We then 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 12 shows a typical case (full node names are not shown for confidentiality reasons). Function C appears in 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 insur-
ance 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. We ran a simple algorithm to identify cases that match this pattern and submitted them to the analysts. The analysts then identified which cases correspond to omissions and which ones did not. The mapping was refined accordingly prior to merging the models.

The analysts also validated the mapping between pairs of process models that were produced automatically, and made a number of corrections amounting to around a third of the matched pairs of elements. This result is consistent with previous validations of the mapping algorithm [Dijkman et al. 2009]. The manual validation of the mapping made the analysts aware of the lack of strict modeling conventions (particularly naming conventions) across different teams of modelers. Indeed, closer analysis showed that most misalignments arose from fragments that had been modeled by two different teams, which did not share the same naming conventions and terminology.

After these pilot experiments with three pairs of models, Suncorp decided to employ the Process Merger tool to support the consolidation of their insurance processes. The algorithm is expected to be integrated in their development environment to produce batch reports showing the degree of consolidation of their models on a regular basis. Moreover, a team of analysts will build a Suncorp-specific ontology to ensure that modelers employ the same terminology, in order to obtain more accurate merged models. In parallel, a governance initiative will be started to implement standardized modeling conventions across all of Suncorp’s process models.

5. DIGEST EXTRACTION

The merge operator starts from a union of the input models. In some scenarios, especially when merging a large number of complex process models, we may not seek a union of the input models, but rather a “digest” showing the most recurrent fragments in the input models. In order to address this requirement, in this section we outline an algorithm to extract a digest from a merged process graph.

The merged graph gives valuable information to derive digests as each edge refers to the set of variants in which the edge is observed. This information, encoded in the edge’s annotation (function α), can be exploited to produce digests of the merged graph at different levels of detail. Specifically, we define the frequency of an edge as the number of variants in which the edge in question appears. The digest of a merged graph is a non-configurable process graph that comprises all edges of
the merged graph that have a frequency above a given frequency threshold. For example, the digest of a process graph with frequency threshold of 2, is the non-configurable process graph obtained by removing all edges in the merged graph that do not appear in at least two of the original variants.

When removing edges from a merged process graph, we may create a disconnected graph. Specifically, a disconnection can only occur between a split and a join that were configurable in the merged graph, such that the region between the split and the join (but excluding these nodes) is a single-entry single-exit region. Here, we observe that if a node is not a connector, the annotation of its incoming edge coincides with the annotation of its outgoing edge. Furthermore, for any split, the annotation of its incoming edge is equal to the union of the annotations of its outgoing edges. So each of the outgoing edges of a split has at most the same number of process identifiers (probably less if the split is configurable) than the incoming edge of the split. Conversely, for any join, the annotation of its outgoing edge is equal to the union of the annotations of its incoming edges. Thus, each of the incoming edges of a join has at most the same process identifiers (less if the join is configurable) than the outgoing edge of the join.

Therefore, if from a start node we walk through the merged graph forward, we observe that traversing a configurable split typically reduces the size of $\alpha$ while a configurable join typically increases it, and all other nodes leave $\alpha$ unchanged. When we create the digest graph, we remove those paths from a configurable split to a configurable join that do not satisfy the given frequency. Thus, in order to avoid disconnections, we just need to reconnect each split in the digest that has lost some outgoing edge, with all its subsequent joins that have lost some incoming edge. If such a path contains at least a node (i.e. if the size of the path is greater than 2), we
Algorithm 7: Digest

1 function Digest(Graph CG, Integer freq)
2   init
3     Graph D
4 begin
5     D ← \{ e ∈ CG | |α_{CG}(e)| ≥ freq \}
6     foreach s in D such that |s •|_D < |s •|CG do
7     foreach j in D such that |• j|_D < |• j|CG do
8       if exists a path p = s ↪ j in CG such that p is not in D and \{|p|\} > 2
9           then
10          z ← new Node("h", "]")
11          D ← D ∪ \{(s, z), (z, j)\}
12       end
13     else if exists a path p = s ↪ j in CG such that p is not in D and
14           |\{|p|\}| = 2 then
15           D ← D ∪ \{(s, j)\}
16       end
17     end
18   end
19 return RemoveTrivialConnectors(D)
20 end

reconnect the split with the join through a placeholder node, otherwise we reconnect
them via a simple edge. The placeholder node, labeled "]" by convention, indicates
that there was a path containing at least a node in the merged graph that does not
meet the frequency threshold. Moreover, we reduce the digest by removing trivial
connectors that may be generated during the derivation of the digest, or may result
from fixing the disconnections (e.g. if a split in the digest had an empty postset will
now have one outgoing edge). The computation of the digest graph is described in
Algorithm 7. Figure 13 shows the construction of the digest with a frequency of 2
for the merged graph in Figure 1.

6. RELATED WORK

The problem of merging process models has been posed in [Sun et al. 2006], [Küster
et al. 2008b], [Gottschalk et al. 2008] and [Li et al. 2009]. Sun et al. [Sun et al. 2006]
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. [Küster et al. 2008b] 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 [Küster et al. 2008a].

Gottschalk et al. [Gottschalk et al. 2008] 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. [Li et al. 2009] 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 [Mendling and Simon 2006; Küster et al. 2007]. 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 [Mendling and Simon 2006] 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. Similar to our approach, the authors define reduction rules to simplify the resulting models, although these rules do not guarantee behavior preservation since the type of connectors being affected by the rule is not changed.

Ryndina et al. [Küster et al. 2007] 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.

The problem of maintaining merged process models has been explored in [Reijers et al. 2009]. Here the authors propose an alternative (mostly manual) method.
which is applicable if the need for maintaining merged models is identified before the actual process modeling effort is started. In contrast, we seek to semi-automatically merge existing process models. Also, the solution proposed in [Reijers et al. 2009] is specifically designed for one modeling notation (EPCs) while our solution can be applied to other modeling notations (e.g., BPMN) thanks to the process graph abstraction.

Software merging [Mens 2002] deals with the problem of reconciling the work of multiple developers working on the same code base concurrently. Software merging techniques primarily deal with reconciling conflicts in text files. In this sense, these techniques tend to differ from those for model merging in general, and process model merging in particular.

Research on model merging has addressed the problem of merging static models (e.g., class diagrams) [Ohst et al. 2003] and dynamic models (e.g., statecharts) [Nejati et al. 2007]. Ohst et al. [Ohst et al. 2003] present an approach to merge two versions of a UML Class or Object Diagram by overlapping common parts, and highlighting specific parts via colors. The purpose is to visualize structural changes between diagram versions (e.g., an attribute being removed or an operation being shifted from one class to another) rather than resolving conflicts. Moreover, as stated by the authors themselves, the use of colors limits this approach to two-way merging, since multi-way merging requires the use of numerous colors which may confuse the reader. On the contrary, our approach relies on configurable connectors and annotations which are suitable for multi-way merging. Nejati et al. [Nejati et al. 2007] propose a technique for merging pairs of statecharts in such a way that the resulting statechart subsumes (in the behavioral sense) the input statecharts. However, their technique only takes into account sequential behavior (no parallelism). In contrast, we deal with different types of branching and merging connectors.

Model merging is also related to database schema integration [Rahm and Bernstein 2001]. In this latter domain, numerous techniques for merging heterogeneous database schemata into a unified schema have been developed. This problem arises for example in the context of federated databases or when a global application needs to access data from multiple databases. In this context, automatic conflict resolution among schemata is generally not feasible and thus, user intervention is required. Once conflicts have been resolved, the schemata are merged by superimposing common entities and relationships and applying restructuring operations.

This paper is an extended and revised version of our previous work [La Rosa et al. 2010]. The main extensions with respect to this previous publication include the reduction rules, the entanglement elimination rules, the extraction of digests, the proof that a merged model subsumes the input models, the extension of the merging algorithm to deal with process graphs containing data and resource attributes, and a more detailed case study.

7. CONCLUSION

The main contribution of this paper is an algorithm that takes as input a pair of process models and produces a merged (configurable) process model. The algorithm ensures that the merged model subsumes the original model and that the elements in the merged model can be traced back to the original models. 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 algorithm 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. A case study has also been conducted in order to validate the usefulness of the merging algorithm in a practical setting.

We have also showed that the output of the merging algorithm can be used to compute digests at different levels of details by exploiting the same annotations that are placed in the merged model in order to ensure traceability. In other words, digest extraction can be seen as a by-product of merging. During the case study, digests were used to shed insights into the commonalities between claim handling processes for different types of claims. It appears that several sub-processes could be shared across these processes, leading to higher standardization and its ensuing economies of scale. However, the exploitation of these opportunities is hindered by the fact that common fragments often differ in subtle ways. For example, the business rules for checking invoices related to personal claims differ from those for motor claims. An avenue for future work is to take into account these differences in business rules in order to determine if a recurrent fragment is suitable for standardization, and to provide methods and tool support for such standardization.

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 [Weidlich et al. 2010]. Integrating the output of such matching techniques into the merging operator is another avenue for future work.

ACKNOWLEDGMENTS
The authors wish to thank Jamie Cornes and his team at Suncorp-Metway for their valuable cooperation leading to the case study reported in this paper. This research is partly funded by the Estonian Science Foundation and the European Regional Development Fund through the Estonian Centre of Excellence in Computer Science.

REFERENCES

ACM Journal Name, Vol. V, No. N, Month 20YY.


