Business Process Model Merging: An Approach to Business Process Consolidation

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

This article addresses the problem of constructing consolidated business process models out of collections of process models that share common fragments. The article 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 article 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: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*; H.4.1 [**Information Systems Applications**]: Office Automation—*Workflow management*

General Terms: Design, Management

Additional Key Words and Phrases: Business process model, variability, model merging, graph matching

ACM Reference Format:
La Rosa, M., Dumas, M., Uba, R., and Dijkman, R. 2013. Business process model merging: An approach to business process consolidation. *ACM Trans. Softw. Eng. Methodol.* 22, 2, Article 11 (March 2013), 42 pages.
DOI:http://dx.doi.org/10.1145/2430545.2430547

1. INTRODUCTION

In the context of company mergers and restructurings, it often occurs that multiple variants of a business process – usually originating from different companies or

This is an extended and revised version of an article presented at CoopIS 2010 in Crete, Greece.
This research was partly funded by the ARC Linkage Project LP110100252 "Facilitating Business Process Standardization and Reuse", the Estonian Science Foundation, and ERDF via the Estonian Centre of Excellence in Computer Science. NICTA is funded by the Australian Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.
Authors' addresses: M. La Rosa, Faculty of Science and Engineering, Queensland University of Technology, 2 George Street, Brisbane QLD 4000, Australia; email: m.larosa@qut.edu.au; M. Dumas and R. Uba, Institute of Computer Science, University of Tartu, J Liivi 2, 50409 Tartu, Estonia; emails: {reinak, marlon.dumas}@ut.ee; R. Dijkman, School of Industrial Engineering, Eindhoven University of Technology, Den Dolech 2, 5612 AZ Eindhoven, The Netherlands; email: R.M.Dijkman@tue.nl.

DOI:http://dx.doi.org/10.1145/2430545.2430547

units – need to coevolve and to eventually converge into a single process in order to eliminate redundancies and create synergies. To this end, teams of business analysts need to compare process models so as to identify commonalities and differences, and create integrated process models that can be used to drive the consolidation effort. This model comparison and merging task is tedious, time-consuming and error-prone. In one instance reported in this article, it took a team of three analysts 130 man-hours to merge 25% of two variants of an end-to-end process model.

The consolidation of multiple "as is" processes into a single "to be" process generally requires nontrivial process changes to be implemented. Accordingly, such consolidation is more appropriately done in a staged manner. Initially, the variants are treated as a collection of processes that need to coevolve. Little by little, differences between variants are analyzed and reconciled, until the variants blend and their differences fade away, giving place to a single process.

With the aim of supporting staged process consolidation, this article proposes an approach to semi-automatically aggregate a collection of process models into a single one. Specifically, the article considers the problem of constructing a "union" of a collection of process models (herewith called a *merged model*) as well as that of constructing an "intersection" of a collection of process models (a *digest*).

The purpose of merged models is to allow analysts to view the commonalities and differences between multiple variants of a business process, and to manage their coevolution and convergence. Instead of making changes to each individual variant separately, analysts can make changes to the merged model. Changes to the merged model are propagated to each affected variant. 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 models leads to a single model that captures the behavior of both original claim handling processes. Whenever an analyst makes a change to the merged model, they may regenerate each of the variants. If a change is made to a region of the merged model that was originally common to both variants, the change will be reflected in both regenerated variants. This should be contrasted with the baseline approach consisting in keeping the variants as separate models. In this baseline approach, an analyst may modify one claim handling process and not the other, and in doing so contribute to making the models diverge from one another, rather than converge.

This discussion leads us to the following requirements.

(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. This requirement ensures that the execution semantics of the variants is not lost in the merged model.
(2) *Traceability*. Given an element in the merged model, analysts should be able to trace back from which process model(s) the element in question originates. In this way, analysts can identify the parts of the merged model that are shared by multiple (or all) variants. These clues are essential for the analyst to understand the impact of a change to the merged model.
(3) *Reversibility*. Analysts should be able to derive the input process models from the merged model. Note that traceability is a necessary (but not sufficient) condition for reversibility.

In this setting, we propose a merging algorithm that 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 (variants) 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 variants 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, allow analysts to identify and manage common fragments while keeping the variants separate. 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 variants in order to focus their effort on consolidating those parts, 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 article 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 (nonconfigurable) process model. The algorithm also takes an additional parameter that allows analysts to stipulate how many times should a fragment recur for it to appear in the digest.

The merging and digest extraction algorithms have been evaluated on process models sourced from different domains. These tests show that the algorithms produce compact and readable models and scale up to process models with 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 an insurance company to build consolidated models of their claim handling processes.

One of the key use cases of merged models is to enable the synchronized coevolution of multiple process variants. In this way, analysts can prevent variants from evolving along diverging paths, thereby reducing the risk of redundancy and inconsistency across variants. While the focus is on the construction of merged models and not on their evolution, the article also outlines a set of change primitives for merged models and a cleaning operator that ensures that the variants that can be derived from the merged model (via individualization) are syntactically correct, in the sense that every edge is on a path from a start to an end node.

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. Section 3 then presents the algorithm for model merging, while Section 4 describes how the output of the merging algorithm can be used to produce digests. Next, Section 5 reports on the implementation and evaluation of this algorithm. Section 6 outlines a framework to support the evolution of merged models. Finally, Section 7 discusses related work and Section 8 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.

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 article, 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 $id_G(n)$ within G, a label $\lambda_G(n)$, 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 article 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 preconditions 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 continues along multiple parallel paths (AND-split), or that it has to wait for multiple parallel paths before continuing (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 before continuing (XOR-join). OR connectors start or wait for multiple paths. The 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 identified as configurable. In order to configure a configurable connector, a user needs to mark for removal one or more of the connector's incoming branches (in the case of a join) or one or more of its outgoing branches (in the case of a split). In addition, a user may "restrict" a configurable OR connector into a regular XOR or a regular AND. We call this operation "restricting" because it reduces the number of possible traces induced by the connector. For example, an XOR-split is more restrictive than an OR-split, because it only allows execution to continue along one of its outgoing paths, while an OR-split allows execution to continue along one or more of its outgoing paths. After all nodes in a C-EPC are configured, a C-EPC is individualized by (i) removing those branches that have been marked for removal during configuration, and (ii) replacing the configurable connectors with regular connectors (taking into account the restrictions made by the user). The result of individualizing a configured connector is a regular connector with a possibly reduced number of incoming or outgoing branches. The individualization may also perform some clean-up operations, such as removing those connectors that are left with only one incoming and one outgoing branch. For more details on the individualization algorithm, we refer to Rosemann and van der Aalst [2007].

The 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 G_1 and G_2 are two possible individualized models of CG. G_1 can be obtained by configuring the three configurable connectors in order to keep all edges 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 edges labeled "2" and restricting the OR-join to an XOR-join. Since in both cases only one edge is kept for the two configurable XOR connectors (either

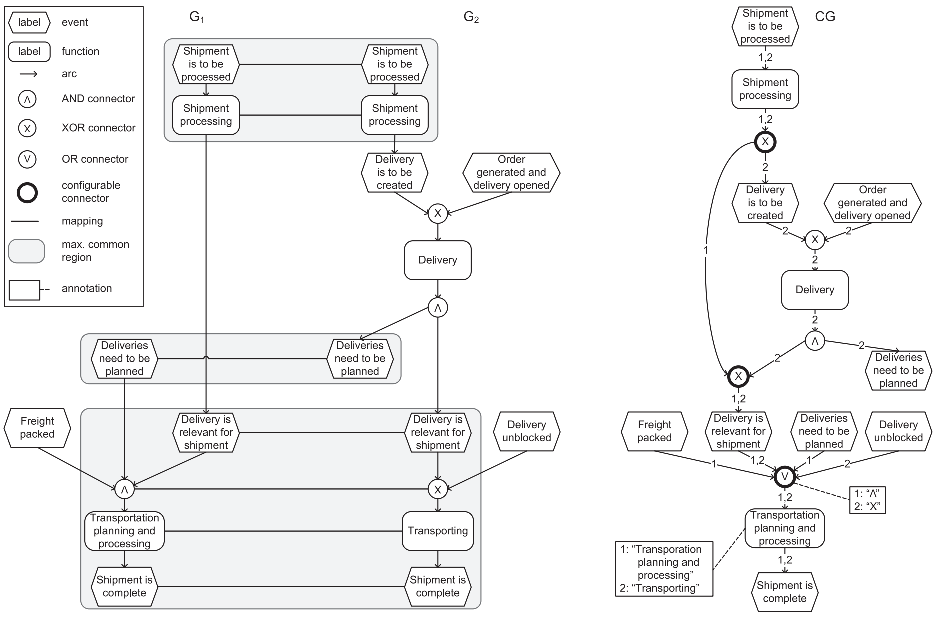

Fig. 1. Two business process models with a mapping, and their merged model.

the one labeled “1” or the one labeled “2”), these connectors are also removed during individualization.

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 output of merging the two EPCs (G_1 and G_2). The configurable XOR-split immediately below the 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 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 Figure 1) to record the fact that the label of this connector is “and” in variant 1, and “xor” in variant 2. These annotations enable us to achieve both traceability and reversibility (cf. requirements (2) and (3) in Section 1). Formally, we have the following.

Definition 2 (*Configurable Business Process Graph*). Let $\mathcal{I}$ be a set of identifiers of business process graphs, and $\mathcal{L}$ the set of all possible node labels. A Configurable

Business Process graph is a tuple $(G, \alpha_G, \gamma_G, \eta_G)$ where G is a business process graph, $\alpha_G : G \rightarrow \wp(\mathcal{I})$[1] 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 presented in Rosemann and van der Aalst [2007]. However, the differences are purely syntactic: while in Rosemann and van der Aalst [2007] process graph identifiers are only attached to edges emanating from a configurable split, in this article we attach process graph identifiers to every edge in a configurable process graph. This syntactic choice makes the definition of the algorithms simpler. Also, by building on top of the concept of process graph, this definition abstracts away from the EPC notation and can be used to capture (configurable) BPMN models as well. Specifically, the above definition covers BPMN models composed of tasks, events and gateways. In Section 3.6, we extend this definition to cover data objects and resource classes (i.e., lanes and pools). However, we do not cover BPMN subprocesses and boundary events nor message flows and choreography-related constructs of BPMN 2.0. The process merge algorithm proposed in this article merges one pair of processes or subprocesses at a time.

Here, we define some basic notations that we will use in the definitions and algorithms in the rest of the article.

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* p between two nodes $n \in N_G$ and $m \in N_G$, denoted $p = n \hookrightarrow m$, 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 $(n_i, n_{i+1}) \in G$. 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 $p = n \overset{c}{\hookrightarrow} m$ 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

Before merging business processes, we need to establish where these processes are the same, that is, we need to find which nodes in the first process graph *match* which nodes in the second process graph. Since there can be different candidate nodes in the second graph that may be matched to a given node in the first graph, and vice versa, the aim of matching two process graphs is to find 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. Figure 1 shows an example in which the mapping is represented by straight lines that

[1] $\wp$ indicates the powerset.

connect mapped nodes. In the figure, the best mapping is easy to establish, because the nodes have very similar labels. However, in practice this cannot be expected. Therefore, a (semi-)automatic technique to determine the best mapping would be of great help.

What is considered to be the best mapping depends on a scoring function, called the *matching score*.

Definition 4 (*Business Process Matching, Mapping*). Let G_1 and G_2 be two business process graphs. Business process matching is the procedure of finding a partial injective mapping $M \subseteq N_{G_1} \nrightarrow N_{G_2}$, for which some function $\mathsf{score} : M \rightarrow [0 \ldots 1]$ is maximal (i.e., there exists no M', such that $\mathsf{score}(M') > \mathsf{score}(M)$).

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 [Dijkman et al. 2009, 2011; van Dongen et al. 2008]. 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 by computing their similarity. The similarity, and thus the matching score, of nodes of different types or between a split and a join is 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 inspired from established techniques for matching pairs of elements for schema matching [Rahm and Bernstein 2001].

This approach to compute similarities between functions/events cannot be used to compute the similarity between pairs of splits or pairs of joins, as connector 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, provided at least one mapping of transitive preset nodes and one mapping of transitive postset nodes exists.

Definition 5 (*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 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). For example, in Figure 1, 14 nodes are involved in the mapping and consequently count as substituted nodes. 7 nodes are not involved in the mapping and consequently count as inserted/deleted nodes. 8 edges are matched. For example, the edge from “shipment is to be processed” to “shipment processing” in graph G_1 is matched to the edge from “shipment is to be processed” to ‘shipment processing’ in graph G_2 (and vice-versa). Consequently, there are 8 substituted edges and 11 inserted/deleted edges.

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 6 (*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 $0 \leq \mathsf{wsubn} \leq 1$, $0 \leq \mathsf{wskipn} \leq 1$ and $0 \leq \mathsf{wskipe} \leq 1$ be weights assigned 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:

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

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.

$$\mathsf{fskipn} = \frac{|\mathsf{skipn}|}{|N_{G_1}|+|N_{G_2}|} \quad \mathsf{fskipe} = \frac{|\mathsf{skipe}|}{|G_1|+|G_2|} \quad \mathsf{fsubn} = \frac{2.0 \cdot \Sigma_{(n,m)\in M} 1.0 - \mathrm{Sim}(n,m)}{|\mathsf{subn}|}$$

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

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

The 1.0 factor in this formulae is used to compute the opposite of the similarity (i.e., the distance) between two nodes or graphs. The 2.0 factor in the fsubsn formula is there for normalization purposes. For example, if a node labeled l_1 is substituted by a node labeled l_2, such that $\mathrm{Sim}(l_1, l_2) = 0$ (and thus the distance $1.0 - \mathrm{Sim}(l_1, l_2) = 1$), this substitution should be treated as equivalent to a deletion followed by an insertion, and thus it should have a magnitude of 2.

The weights wskipn, wskipe and wsubn are included in the matching score formula in order to allow users to adapt the matching technique to a particular setting. For example, in Dijkman et al. [2011], we conducted experiments to calibrate these weights in order to maximize the precision of the technique with respect to a sample set of models taken from the SAP reference model. In those experiments, it was shown that the precision of the matching technique remains high, so long as the ratio $(\mathsf{wskipn} + \mathsf{wskipe})/\mathsf{wsubn}$ is between 0 and 2. In particular, the matching technique works well if all three weights are set to 1.0, while moderate improvements are observed when $\mathsf{wskipn} + \mathsf{wskipe} = \mathsf{wsubn}$.

For example, in Figure 1, 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. 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 $1.0 - \frac{\frac{7}{21} + \frac{11}{19} + \frac{2 \cdot 0.5 + 2 \cdot 0.65}{14}}{3} = 0.64$. However, other values can be assigned for the weights depending on the properties of the dataset and the preferred result. For instance if the structure of the model carries more information than the node labels, then the skipping edges should be more penalized.

Definition 6 gives the matching score of a given mapping. To determine the matching score of two process graphs, we need to find the mapping that yields the highest matching score. Heuristics for this task are given in Dijkman et al. [2011].

3. MERGING ALGORITHM

The merging algorithm is defined over pairs of configurable process graphs. In order to merge two or more (nonconfigurable) 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 graph into a configurable process graph, we can proceed to merge the configurable process graphs. We first present the basic merging algorithm and 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. The notation used in the algorithms of this article is summarized in Appendix A.

3.1. Basic Merging Algorithm

Given two configurable process graphs G_1 and G_2 and their mapping M with the highest matching score, the merging algorithm (Algorithm 1) starts by creating an initial version of the merged graph CG by computing 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

Algorithm 1: Merge

```
1  function Merge(Graph $G_1$, Graph $G_2$, Mapping M)
2  init
3    Mapping mcr, Graph CG
4  begin
5    $CG \Leftarrow G_1 \cup G_2 \setminus (G_2 \cap sube)$
6    foreach $(x, y)$ in $CG \cap sube$ do
7      $\alpha_{CG}(x, y) \Leftarrow \alpha_{G_1}(x, y) \cup \alpha_{G_2}(M(x), M(y))$
8    end
9    foreach n in $N_{CG} \cap subn$ do
10     $\gamma_{CG}(n) \Leftarrow \gamma_{G_1}(n) \cup \gamma_{G_2}(M(n))$
11   end
12   foreach mcr in MaximumCommonRegions($G_1, G_2$, M) do
13     foreach $fG_1$ in dom(mcr) such that $|\bullet fG_1| = 1$ and $|\bullet M(fG_1)| = 1$ and
       (Any($\bullet fG_1$) $\notin$ dom(mcr) or Any($\bullet M(fG_1)$) $\notin$ cod(mcr)) do
14       $pfG_1 \Leftarrow$ Any($\bullet fG_1$), $pfG_2 \Leftarrow$ Any($\bullet M(fG_1)$)
15       xj $\Leftarrow$ new Node("c","xor",true)
16       $\gamma(xj) = \{(Pid(G_1),\text{"xor"}), (Pid(G_2),\text{"xor"})\}$
17       $CG \Leftarrow (CG \setminus (\{(pfG_1, fG_1), (pfG_2, fG_2)\})) \cup \{(pfG_1, xj), (pfG_2, xj), (xj, fG_1)\}$
18       $\alpha_{CG}(pfG_1, xj) \Leftarrow \alpha_{G_1}(pfG_1, fG_1)$
19       $\alpha_{CG}(pfG_2, xj) \Leftarrow \alpha_{G_2}(pfG_2, fG_2)$
20       $\alpha_{CG}(xj, fG_1) \Leftarrow \alpha_{G_1}(pfG_1, fG_1) \cup \alpha_{G_2}(pfG_2, fG_2)$
21     end
22     foreach $lG_1$ in dom(mcr) such that $|lG_1 \bullet| = 1$ and $|M(lG_1) \bullet| = 1$ and
       (Any($lG_1 \bullet$) $\notin$ dom(mcr) or Any($M(lG_1)\bullet$) $\notin$ cod(mcr)) do
23       $slG_1 \Leftarrow$ Any($lG_1\bullet$), $slG_2 \Leftarrow$ Any($M(lG_1)\bullet$)
24       xs $\Leftarrow$ new Node("c","xor",true)
25       $\gamma(xs) = \{(Pid(G_1),\text{"xor"}), (Pid(G_2),\text{"xor"})\}$
26       $CG \Leftarrow (CG \setminus (\{(lG_1, slG_1), (lG_2, slG_2)\})) \cup \{(xs, slG_1), (xs, slG_2), (lG_1, xs)\}$
27       $\alpha_{CG}(xs, slG_1) \Leftarrow \alpha_{G_1}(lG_1, slG_1)$
28       $\alpha_{CG}(xs, slG_2) \Leftarrow \alpha_{G_2}(lG_2, slG_2)$
29       $\alpha_{CG}(lG_1, xs) \Leftarrow \alpha_{G_1}(lG_1, slG_1) \cup \alpha_{G_2}(lG_2, slG_2)$
30     end
31   end
32   CG $\Leftarrow$ MergeConnectors(M, CG)
33   return CG
34 end
```

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*.

Algorithm 2: Maximum Common Regions

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

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 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.

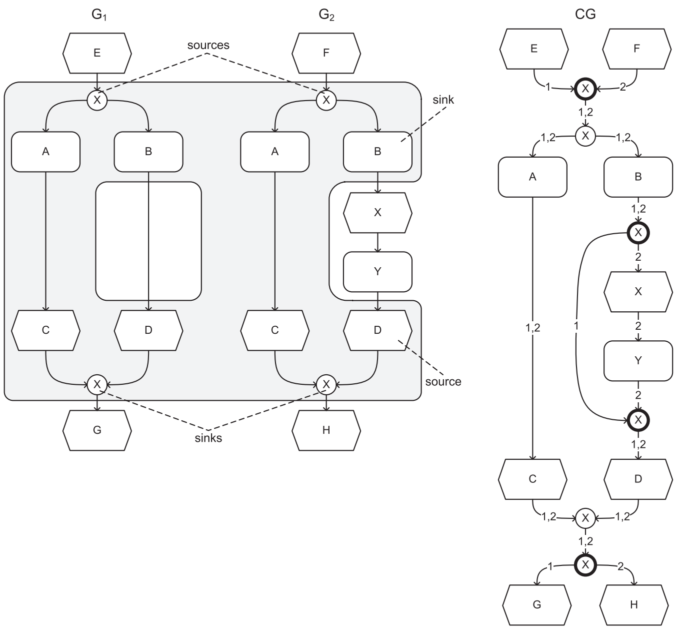

Fig. 2. An example where a node is a source (sink) in one graph but not in the other.

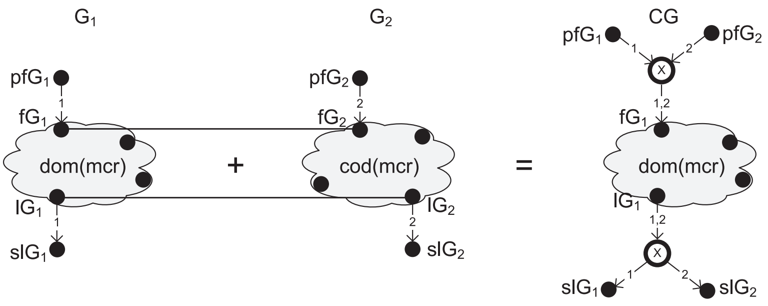

Fig. 3. Reconnecting a maximum common region to the nodes that are not matched.

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 nodes that are connectors ("matched connectors" for short), 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). We also observe that the merging algorithm accepts both configurable and non-configurable process graphs as input. Thus, the merging operator can be used for multiway 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. Complexity Analysis

The complexity of the algorithm for merging connectors is linear on the maximum number of connectors, which is bounded by the number of edges ϵ of the largest graph to be merged. Thus, the complexity is $O(\epsilon)$. The algorithm for calculating the maximum common regions is a breadth-first search which explores all nodes in a mapping, and for each of them, it cycles over the neighbor nodes. The number of nodes in a mapping is bounded by ϵ, while the number of neighbors of a node is bounded by the maximum degree δ among all nodes in the two input graphs. Thus, the complexity of this algorithm is $O(\epsilon\ \delta)$.

The algorithm for calculating the merged model calls the algorithm for calculating the maximum common regions (line 12). Then, it visits at most all nodes of each maximum common region (lines 13 and 22) and, for each of them, it inserts and deletes some edges on the merged graph CG (lines 17 and 26). Finally, it calls the algorithm for merging connectors (line 32). Edge insertions and deletions are bounded by $\log(\epsilon)$, which is the time that is needed to traverse the graph CG with a Binary tree if CG is implemented as a set of edges. 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 different regions do not share edges. Checking that the only predecessor node (line 13) or successor node (line 22) of an element in a maximum common region is not in the same region can be done in constant time using a Hash table. Therefore, the complexity of visiting all nodes of all maximum common regions (lines 13–30) is $O(\epsilon)$. Hence, the complexity of the merging algorithm is $O(\epsilon\ \log(\epsilon)\ \delta)$.

Algorithm 3: Merge Connectors

```
function MergeConstructors(Mapping M, {Edge} CG)
init
  {Node} $S_c \Leftarrow \emptyset$, {Node} $P_c \Leftarrow \emptyset$
begin
  foreach c in dom(M) such that $\tau(c) =$"c" do
    $S_c \Leftarrow \{x \in M(c)\bullet \mid x \notin cod(M)\}$
    $P_c \Leftarrow \{x \in \bullet M(c) \mid x \notin cod(M)\}$
    $CG \Leftarrow (CG \setminus \bigcup_{x \in S_c}\{(M(c), x)\} \cup \bigcup_{x \in P_c}\{(x, M(c))\}) \cup \bigcup_{x \in S_c}\{(c, x)\} \cup \cup \bigcup_{x \in P_c}\{(x, c)\}$
    foreach x in $S_c$ do
      $\alpha_{CG}(c, x) \Leftarrow \alpha_{G_2}(M(c), x)$
    end
    foreach x in $P_c$ do
      $\alpha_{CG}(x, c) \Leftarrow \alpha_{G_2}(x, M(c))$
    end
    if $|S_c| > 0$ or $|P_c| > 0$ then
      $\eta_{CG}(c) \Leftarrow$ true
    end
    if $\lambda_{G_1}(c) \neq \lambda_{G_2}(M(c))$ then
      $\lambda_{CG}(c) \Leftarrow$"or"
    end
  end
  return CG
end
```

On top of this, we need to consider the complexity of computing the best matching score of the two input graphs. For example, if we use a greedy algorithm [Dijkman et al. 2011], the complexity is cubic on the number of edges ϵ of the largest graph.

3.3. Properties of the Algorithm

In Section 1, we stated that the algorithm should satisfy three requirements: behavior-preservation, traceability and reversibility. The traceability requirement is met simply because a configurable business process graph relates each of its elements to the element from which it was derived by means of functions γ and α in Definition 2. Here, we sketch the proofs of two propositions showing that the algorithm fulfills the first and third requirements.

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 $e_1 e_2 \cdots e_n$ 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 the union of the set of edges of the input graphs, and subsequently, the algorithm only adds edges to CG.
(2) According to lines 17 and 26 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 17 and 26). Let (n, m) be such an edge at the boundary of a common region. According to lines 17 and 26, 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 $e_1 e_2 \cdots e_n$ 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. □

PROPOSITION 2. *Let CG be the configurable process graph produced by Algorithm 1 when given process graphs G_1 and G2 as input. CG has two possible individualizations: G_1 and G_2.*

PROOF. We first sketch the proof for graph G_1. We will show that the individualization of the merged graph CG for process identifier 1 is G_1. To this end, we make three observations.

(1) According to line 5 of the algorithm, for each edge e in CG originally appearing in G_1, CG will contain edge e since the set of edges of CG is equal to the union of the set of edges of the input graphs. Moreover, according to line 7 of the algorithm, the process identifier attached to each edge e contains process identifier 1. Thus, the individualization of CG for process identifier 1 contains all edges in G_1.
(2) According to line 10, for each node n in CG originally appearing in G_1, $\gamma_{CG}(n)$ maps identifier 1 to the label of node n in G_1. When CG is individualized for identifier 1, each node in the individualized graph is then given the same label that the node had in G_1.
(3) According to lines 17 and 26 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. By construction, if c is a join it only has two predecessors while if it is a split it only has two successors and the annotation of the edges to its predecessors/successors contain identifier 1 (lines 19–21 and 29–31). Hence, when the graph is individualized for process identifier 1, the incoming or outgoing edge of c that does not contain identifier 1 in its annotation is removed. Since c is left with one incoming and one outgoing edge, it is replaced by an edge between its unique predecessor and its unique successor. As a result, c does not appear in the individualized graph.

From these observations, we conclude that the individualization of CG for identifier 1 contains the same edges as G_1, the same nodes as G_1, and that configurable connectors in CG that do not exist in G_1 are deleted during the individualization. Thus, the individualization of CG for identifier 1 is exactly G_1.

The second and the third observations also hold for G_2. The first observation differs with respect to:

(1) the set of edges $G_2 \cap sube$ (i.e., the edges in G_2 that are mapped to edges in G_1). These edges are initially excluded from CG (line 5) because for each mapped edge only the copy in G_1 is kept; and
(2) the set of edges connecting a mapped node in G_2 to a node in G_2 that is not mapped. As a consequence of excluding edges $G_2 \cap sube$, these edges are also initially excluded from CG. This causes CG to be disconnected in two separate graphs: the graph of the mapped nodes plus the nodes of G_1 which are not mapped, and the graph of the nodes of G_2 which are not mapped.
(3) the set of edges that connect a node from G_2 that is mapped to a node in G_1 and another node from G_2 that is *not* mapped to a node in G_1. These edges cause the merged graph to be disconnected, consisting of a separate graph for G_1 and a separate graph for G_2 and need to be reconnected.

The set of edges $G_2 \cap sube$ is added to CG in line 7, by adding the annotations that these edges had in G_2 to edges that originate from G_1. In line 10, the same operation is done for the annotations of the nodes connected by these edges. Each node x in G_2 that is not mapped, is reconnected to a node c in CG which originates from a node in G_1 that is mapped, by Algorithm 3 (line 8). This is done via new edges of the type (c, x) or (x, c) which take the annotations of the original edges between x and the mapped node for c in G_2 (lines 10 and 13). Thus, the individualization of CG for process identifier 2 contains all edges in G_2. Hence, we conclude that the individualization of CG for identifier 2 is exactly G_2. It does so in such a way that they connect a node that originates from G_1 (this is the node to which the node from G_2 was mapped) to a node that originates from G_2. Aside from these differences, the first observation made here for G_1 also holds for G_2. □

3.4. 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 G_1 and G_2 with common regions X-X', A-A', B-B' and Z-Z' are merged, but the resulting configurable graph CG_A 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 individualized model thereof because each of the two edges that introduces the cycle is annotated with the identifier of either of the originating process graphs, 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, this graph is not as compact as it could be.

An alternative merged model (namely CG_B) 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_A.

If we analyze this entanglement pattern further, we observe that the underlying cause is not in the merging algorithm, but rather in the mapping. 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

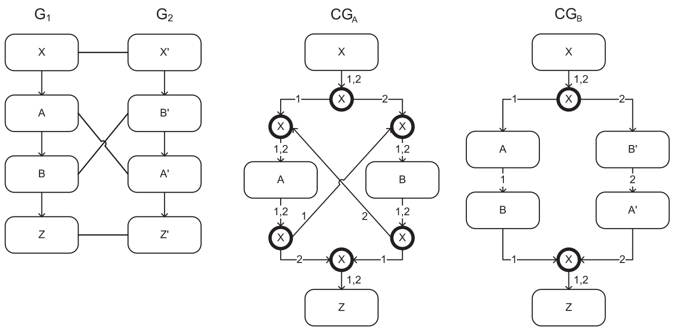

Fig. 4. An entanglement in a merged graph.

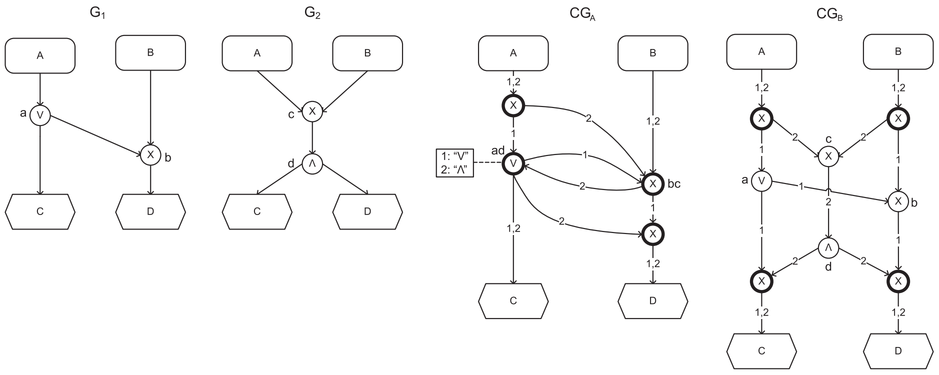

Fig. 5. An example of entanglement with connectors.

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.

(1) 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 traversing A, A', B', B (in this order) or A', A, B, B'.
(2) There is no cyclic path traversing A and B in G_1.
(3) There is no cyclic path traversing A' and B' in G_2.

Whenever we find two pairs of nodes fulfilling these 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 these conditions. So the final mapping will only contain (X,X') and (Z,Z'). The merged model obtained from this latter mapping is CG_B in Figure 4.

The entanglement problem can also occur between connectors. Figure 5 shows such an example. 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 entanglement (cf. cycle between nodes ad and bc). Again, the entanglement can be avoided by removing the pairs (a,d) and (b,c) from

Algorithm 4: Merge Consecutive Connectors

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

the mapping. The resulting merged graph is CG_B. Although CG_B has no cycles, it is less compact than CG_A. In the next section, we show how to simplify process graphs like CG_B by applying reduction rules.

3.5. 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, that is, 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.

3.5.1. Merge Consecutive Splits/Joins. Function *MergeConsecutiveConnectors* (Algorithm 4) merges two consecutive splits (joins) into a single split (join) connector. Since the idea is to eliminate unnecessary connectors introduced by the merging 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 added by Algorithm 1. The other connector will necessarily be an *original* connector, that is, a connector that existed in one of the input graphs. This condition is checked using function *IsAdded* that takes an edge as input and returns true if the edge’s source or target is a configurable XOR added during merging.

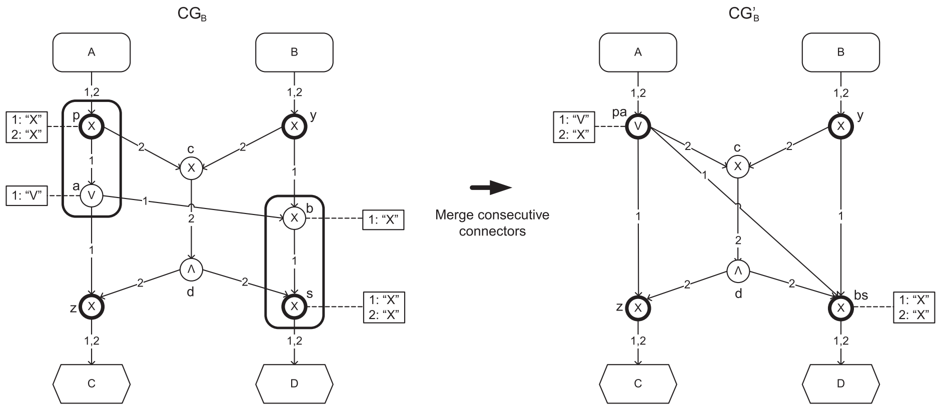

Fig. 6. Merging consecutive splits and joins in the graph CG_B of Figure 5.

In order to merge two consecutive splits m and n, we first remove all incoming edges of 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 edge (m, x) to be the union of its label and that of the removed edge (n, x), so that no information about the original variants is lost. Then we update the annotation of m via the "$\oplus$" 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. In other words, "$\oplus$" computes the union of the annotations of m and n except that if there are two different connector labels for the same process identifier, for example, (1, "xor") and (1, "and"), the pair (pid,"xor") is discarded. 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.

The reduction rule preserves the traceability between the elements in the merged graph and the elements in the input graphs through functions α and γ. The input graphs can be reconstructed from the merged graph via function *IsAdded*, which checks that the removed connector is either a connector added during the merge, or an original connector that succeeds or precedes an added connector (line 3). In either case, the annotation of the merged connector resulting from the reduction rule contains the annotation of the original connector (lines 9 and 20). Moreover, the edges linking the connectors with their surrounding nodes become edges of the merged connector (lines 5 and 16) and preserve their annotations (lines 7 and 18).

We also 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 fewer connector.

Algorithm 5: Remove Redundant Transitive Edges

```
function RemoveRedundantTransitiveEdges({Edge} CG)
begin
  foreach $(m, n)$ in CG such that $|m \bullet| > 1$ and $|\bullet n| > 1$ and exists a path $p = m \overset{c}{\hookrightarrow} n$ in
  $CG^*$ such that $|\{p\}| > 2$ and for all connectors $c \in \{p\} \setminus \{m, n\}$ there not exists a node
  $x \in c\bullet \setminus \{p\}$ such that $\alpha(c, x) \cap \alpha(m, n) \neq \varnothing$ or $x \in \bullet c \setminus \{p\}$ such that $\alpha(x, c) \cap \alpha(m, n) \neq \varnothing$ do
    $CG \Leftarrow CG \setminus \{(m, n)\}$
    foreach $(x, y)$ in CG such that $\{x, y\} \in \{p\}$ do
      $\alpha(x, y) \Leftarrow \alpha(x, y) \cup \alpha(m, n)$
    end
    foreach c in $\{p\} \setminus \{m, n\}$ do
      $\eta(c) \Leftarrow$ true
      $\gamma(c) \Leftarrow \gamma(c) \cup \bigcup_{pid \in \alpha(m,n)} \{(pid, \text{“xor”})\}$
      if $\lambda(c) \neq$ “xor” then
        $\lambda(c) \Leftarrow$“or”
      end
    end
  end
  return CG
end
```

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).

3.5.2. 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 *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 an AND or OR connector, there would exist at least one edge linking that connector to a node not in the connector chain and containing 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[2] there are three redundant transitive edges: (pa, bs), (pa, z) and (y, bs), highlighted with a thicker line in the picture.

[2]Trivial annotations are not depicted.

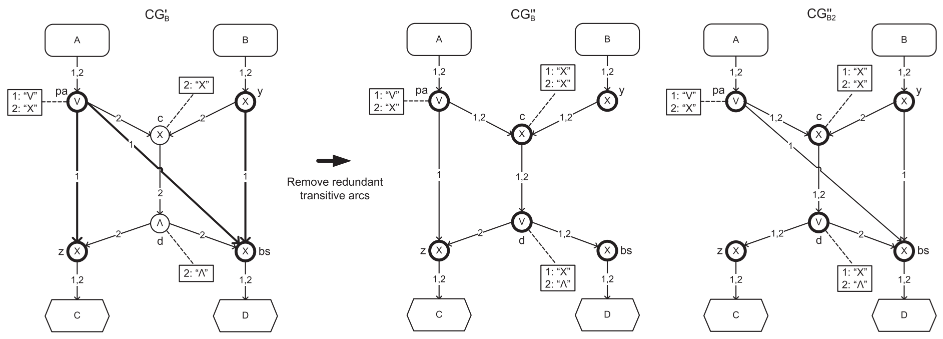

Fig. 7. Removing redundant transitive edges from graph CG'_B of Figure 6.

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,"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 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).

The reduction rule preserves the traceability of graph elements through functions γ and α. The rule also preserves the ability to reconstruct the input graphs from the reduced graph. The graphs that are affected by removing the redundant edge can be reconstructed because their process identifiers are added to each edge in the alternative connector chain (line 6). This chain does not have alternative paths for the identifiers of the graphs the redundant edge came from. Consequently, for these graphs the connector chain is only a single path that, if we abstract from connectors (silent steps), corresponds to the redundant edge. Each graph whose process identifier was already in the annotation of the edges in the connector chain, can be reconstructed because its own identifier and the original types of its connectors are preserved by α and γ. Finally, 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.

3.5.3. Remove Trivial Connectors. A *trivial* connector is one that only has one incoming and one outgoing edge. Such connectors 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 merging

Algorithm 6: Remove Trivial Connectors

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

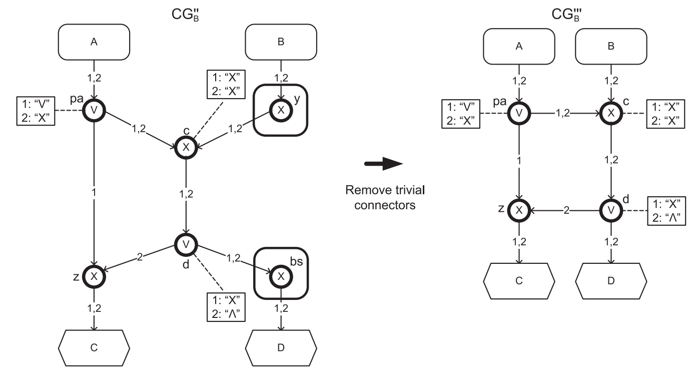

Fig. 8. Removing trivial connectors from graph CG_B'' of Figure 7.

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. Note 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 entanglement. The two initial graphs G_1 and G_2 of Figure 5 can be derived by configuring CG_B''' for the process identifier 1, 2, respectively.

This reduction rule preserves the traceability of graph elements through γ and α. The rule also preserves the ability to reconstruct the input graphs from the reduced graph, because it only removes trivial connectors that are configurable, i.e. that have been produced during the application of a reduction rule, and thus did not exist in any input graph. Moreover, the nodes that were linked via this connector are now directly linked by a new edge bearing the annotation of the removed edges (lines 5 and 6). We also 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 split/join behavior, it does not create additional traces not present in the unreduced graph.

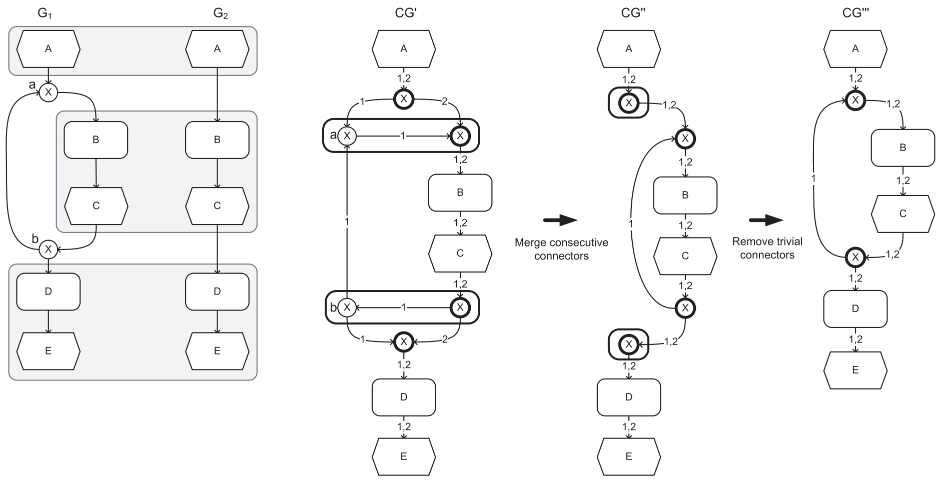

Fig. 9. An example of merging a model containing a loop.

Now that we have presented the reduction rules, we can illustrate how the merging algorithm deals with loops. The merging algorithm can deal with models that contain loops without having to treat them as special cases. For example, Figure 9 shows how a model containing a simple loop is merged with a model without a loop. The maximum common regions in this case are computed in the same way as if no loops were present. In this example, the merged model gives rise to two consecutive joins and two consecutive splits, which are merged and the resulting trivial connectors removed.

3.6. 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. [2011].

In view of handling non-control-flow elements during process merging, we extend the concept of process graph into that of *multiperspective* process graph. A multiperspective process graph is a process graph where each node may be associated with one or multiple non-control-flow elements, each such 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”, or type ‘lane’ and label “Production department”. Hence, this extension allows us to capture non-control-flow information in a language-independent manner and to “carry on” this information during process merging. In the case of EPCs, we can use this extension to capture objects, roles, but also nonfunctional elements representing risks or cost items. In the case of BPMN, this definition allows us to capture data objects attached as inputs or outputs to activities as well as lanes and pools. Note that lanes and pools may be seen as objects attached to each activity. Simply, we associate each activity in a multilane or multipool process model to its enclosing lane.

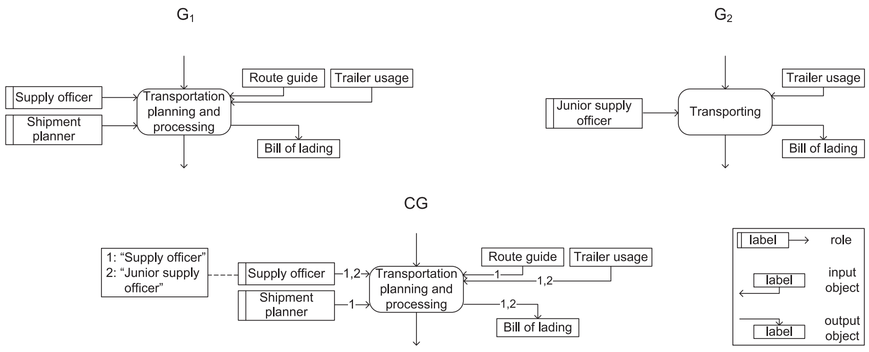

Fig. 10. Merging process models with roles and objects.

In order to merge multiperspective 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 previously. 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 10 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.

The complexity of the merging algorithm remains proportional to the maximum number of edges between the two input models. Naturally, this number increases when we have to deal with edges connecting a function to its roles and objects.

4. DIGEST EXTRACTION

The merge operator starts from the union of the input models. In some scenarios, especially when merging a large number of complex process models, we may not seek the 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 nonconfigurable 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 number of process identifiers (less if the join is configurable) as 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 α while a configurable join typically increases it, and all other nodes leave α 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 reconnect the split with the join through a placeholder node, otherwise we reconnect them via a simple edge. The placeholder node is a function labeled "#" by convention, indicating that there was a path containing at least a node in the merged graph below the frequency threshold. Moreover, we reduce the digest by removing trivial connectors that may be generated during the derivation of the digest (e.g., if a split in the digest had an empty postset, it will now have one outgoing edge). The computation of the digest graph is described in Algorithm 7. Here we invoke function RemoveTrivialConnectors*. This function is the same as the one defined in Algorithm 6, except that it removes any trivial connector, not only those that are configurable. Figure 11 shows the construction of the digest with a frequency of 2 for the merged graph in Figure 1.

5. EVALUATION

The merging algorithm has been implemented as a tool, namely *Process Merger*, that is freely available as part of the Synergia toolset.[3] 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. [2011] 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 fomat). 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 platform – a process model repository toolset.[4] 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

[3]See `http://www.processconfiguration.com`
[4]See `http://www.apromore.org`

Algorithm 7: Digest

```
function Digest(Graph $\mathsf{CG}$, Integer $\mathsf{freq}$)
init
    Graph $\mathsf{D}$
begin
    $\mathsf{D} \Leftarrow \{\mathsf{e} \in \mathsf{CG} \mid |\alpha_{\mathsf{CG}}(\mathsf{e})| \geq \mathsf{freq}\}$
    foreach $\mathsf{s}$ in $\mathsf{D}$ such that $|\mathsf{s}\bullet|_{\mathsf{D}} < |\mathsf{s}\bullet|_{\mathsf{CG}}$ do
        foreach $\mathsf{j}$ in $\mathsf{D}$ such that $|\bullet\mathsf{j}|_{\mathsf{D}} < |\bullet\mathsf{j}|_{\mathsf{CG}}$ do
            if exists a path $\mathsf{p} = \mathsf{s} \hookrightarrow \mathsf{j}$ in $\mathsf{CG}^*$ such that $\mathsf{p} \notin \mathsf{D}^*$ and $|\{\mathsf{p}\}| > 2$ then
                $\mathsf{z} \Leftarrow$ new Node("f","#",false)
                $\mathsf{D} \Leftarrow \mathsf{D} \cup \{(\mathsf{s},\mathsf{z}), (\mathsf{z},\mathsf{j})\}$
            end
            else if exists a path $\mathsf{p} = \mathsf{s} \hookrightarrow \mathsf{j}$ in $\mathsf{CG}^*$ such that $\mathsf{p} \notin \mathsf{D}^*$ and $|\{\mathsf{p}\}| = 2$ then
                $\mathsf{D} \Leftarrow \mathsf{D} \cup \{(\mathsf{s},\mathsf{j})\}$
            end
        end
    end
    return RemoveTrivialConnectors$^*$($\mathsf{D}$)
end
```

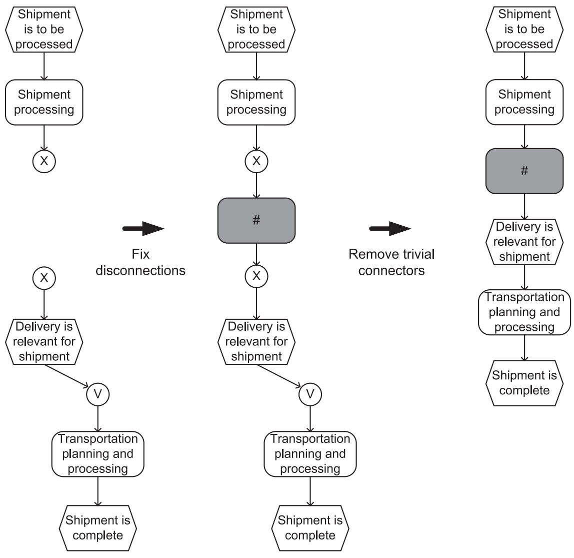
Shipment is to be processed
Shipment processing
X
X
Delivery is relevant for shipment
V
Transportation planning and processing
Shipment is complete
Fix disconnections
#
Remove trivial connectors

Fig. 11. The construction of the digest with frequency 2 for graph *CG* in Figure 1.

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 the size and complexity of the merged models, as well as the scalability of the merge algorithm. Furthermore, we conducted a case study to evaluate the potential usefulness of merged models and digests in practice. In these experiments, we set the weights wskipn, wskipe, and wsubn of the matching score formula to 1.0 (see Definition 6). In other words, we give equal priority to all graph-edit operations.

Table I. Size Statistics of Merged SAP Reference Models

	Size 1	Size 2	Size merged	Compression	Size after reduction	Compression after reduction	Compression without entanglements
Min	3	3	3	0.50	3	0.50	0.50
Max	130	130	194	1.17	186	1.06	1.06
Average	22.07	24.31	33.9	0.76	31.52	0.69	0.69
Std dev	20.95	22.98	30.35	0.15	28.96	0.13	0.13

All the process merging steps are done automatically—human intervention was only needed in order to validate and fine-tune the mapping, prior to execution of the merge algorithm.

5.1. Size of Merged Models

Size – defined as the number of edges – is a key factor affecting the understandability of process models [Mendling et al. 2010]. 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. However, if we merge very similar models, we expect to obtain a model of size close to that of the largest of the two models.

We conducted tests to compare 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. [2011] to search for 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 nonidentical EPCs. Next, we merged each of these model pairs and calculated the *compression ratio* [Salomon 2006], which in our context is the ratio between the size of the merged model and the size of the input models, that is, $CR(G_1, G_2) = |CG|/(|G_1| + |G_2|)$, where $CG = Merge(G_1, G_2)$. A compression ratio 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, that is, the merging operator merely juxtaposes the two input models side-by-side. A compression ratio close to 0.5 (but 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 introduction of configurable connectors during merging may induce an overhead leading to compression ratios above 1.

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 ratio before applying any reduction rule. The last three columns show the size and compression ratio of the merged model after applying the reduction rules, and the compression ratio after removing from the mapping those nodes that generate entanglement. The compression ratios shown in this table refer to the minimum, maximum, average and standard deviation of the compression ratios obtained for the entire SAP dataset, and not the compression ratios when merging the models in columns 2 and 3. The table shows that the reduction rules improve the compression ratio (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

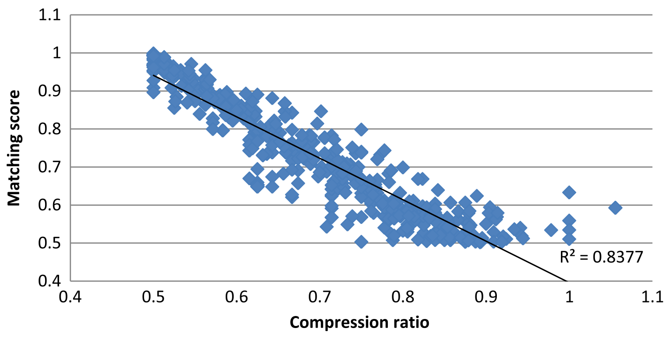

Fig. 12. Matching score of input models and compression ratio.

models are, the more they share common regions and thus the smaller the compression ratio is. This hypothesis is confirmed by the scatter plot in Figure 12 which shows the compression ratios (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 impact of not merging nodes that generate entanglement on the compression ratio is negligible (the average compression ratio increases from 68.76% to 69.43%). Previous research suggests that “model size” is the most prominent of the model characteristics that can explain the understandability of a process model: a decrease in model size leads to an increase in understanding of a model [Mendling et al. 2007]. Since a compression ratio lower than 1 implies a decrease in overall size of the model collection, there is evidence that the merge operator improves understandability of the model collection. This question is further discussed in Section 5.4.

5.2. Scalability of Merge Operator

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. This pair of models were originally captured in BPMN.

Table II shows the sizes of the input models, their matching score, the total execution times, and statistics related to the sizes of the merged models and digest. 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.

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

Table II. Results of Merging Insurance and Land Development Models

Pair #	Size 1	Size 2	Match score	Total time (merge time) in msec.	Size merged	Compression	Size merged after reduct.	Compr. after reduct.	Compr. without entang.	Digest size
1	339	357	0.84	7409 (79)	486	0.70	474	0.68	0.75	339
2	22	78	0.56	78 (0)	88	0.88	87	0.87	0.87	24
3	468	211	0.62	3693 (85)	641	0.94	624	0.92	0.93	374
4	198	191	0.82	853 (20)	290	0.75	279	0.72	0.78	190

times we observed when merging the pairs from the SAP reference model. Table II also shows the compression ratios. Pairs 2 and 3 have a poor compression ratio (lower is better). However, this can be explained by the fact that these pairs of models have a low matching score (0.56 and 0.62).

5.3. Effect of Entanglement Removal

From Table II, we can also observe an increase in the compression ratio after removing nodes that generate entanglements. This increase is significant in pair 1 (compression ratio increases from 68% to 75%) and in pair 4 (from 72% to 78%). Although there is an apparent correlation between the increase in compression ratio 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 5.3 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 5.3). We can see this by comparing this model with the merged model after removing entanglements (shown in the right-hand side of Figure 5.3), 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.

To validate this hypothesis, we computed three complexity metrics on the merged models from the insurance and land development pairs, before and after removing entanglements. Specifically, we extracted *density*, *structuredness* and *sequentiality* [Mendling 2009]. The density of a process graph refers to the number of edges divided by the maximum number of possible edges. Structuredness refers to the degree to which a process graph is composed of single-entry single-exit (SESE) regions (i.e., block-structuredness). This metric is defined as the ratio of matching connectors that are either the entry or the exit of a SESE region relative to the total number of connectors. Sequentiality measures the degree to which a graph is constructed of sequences, and is defined as the number of edges between non-connector nodes divided by the total number of edges. Previous studies [Mendling et al. 2007] have found that density is negatively correlated with understandability (a lower density means more understandable process models) while structuredness and sequentiality are positively correlated with understandability. Table III shows these complexity metrics before and after the removal of entanglements for the insurance and land development models of Table II. The second model pair is not shown since in that pair there are no entanglements. The first two columns of Table III give the size of the mapping (i.e., the number of pairs of nodes mapped) before and after removing entanglements. The remaining columns give the complexity metrics. We observe that the merged models without entanglements have always lower density and higher structuredness and sequentiality than the models with entanglements. These results confirm the hypothesis that by

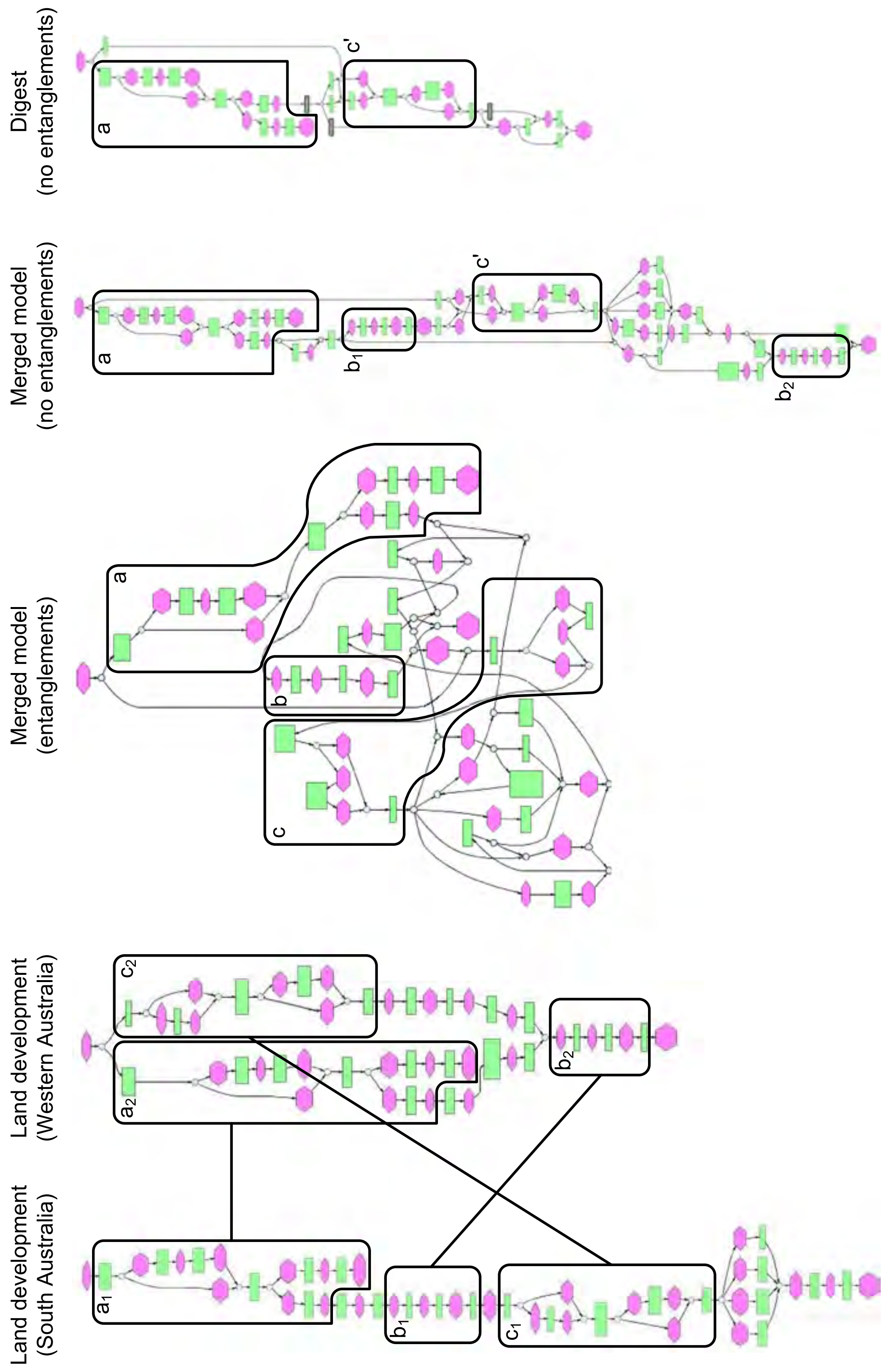
Land development (South Australia)
Land development (Western Australia)
Merged model (entanglements)
Merged model (no entanglements)
Digest (no entanglements)
a1
b1
c1
a2
b2
c2
a
b
c
c'

Fig. 13. Extract of land development models, their merged model with and without entanglements, and their digest.

Table III. Understandability Metrics for the Merged Models with and without Entanglements

Pair #	|Mapping| unentangled	|Mapping| entangled	With entanglement			W/out entanglement		
			Density	Structureness	Sequentiality	Density	Structureness	Sequentiality
1	305	40	0.51	0.15	0.26	0.31	0.59	0.27
3	198	10	0.91	0.13	0.14	0.82	0.13	0.15
4	160	24	0.45	0.18	0.29	0.25	0.32	0.34

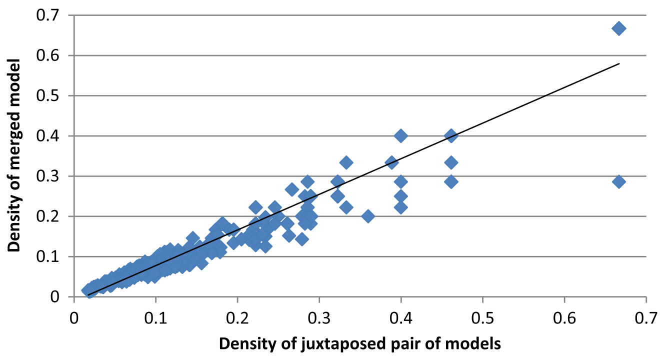

Fig. 14. Density of merged models vs. density of juxtaposed original models.

removing entanglements we obtain models that may be moderately larger but less complex.

5.4. Density of Merged Models

One could argue that while the size of the merged models is lower than the sum of the sizes of the input models (cf. Section 5.1), the merged models are not necessarily easier to understand because they contain additional (configurable) connectors. In other words, merging process variants might lead to less process model elements overall, but also to denser models that analysts might find harder to understand. An alternative to combining process variants using the proposed merge algorithm, would be to juxtapose the process variants. In other words, given two variants V1 and V2, one could trivially construct a configurable process model CM by putting a configurable XOR-split at the start of CM, a configurable XOR-join at the end of CM, and two branches between these two connectors: one containing V1 (unchanged) and another branch containing V2 (unchanged). This configurable model constructed by juxtaposition would satisfy the three requirements outlined in Section 1. The question then raises as to whether the models produced by our merge algorithm are easier or harder to understand than the configurable models obtained by juxtaposition. To address this question, we plotted the density of the merged models obtained from the SAP dataset (after entanglement removal) against the density of the "juxtaposed models". The results are shown in Figure 14 where each point represents a pair of models in the dataset. The x-coordinate of a point is the density of the juxtaposed pair of models, while the y-coordinate is the density of the merged model after entanglement removal. The figure shows that the density of the merged models is in general slightly lower than the density of the juxtaposed models. The average density of the merged models is 0.127 versus 0.158 for the juxtaposed models. As mentioned above, lower density is correlated with higher understandability [Mendling et al. 2007].

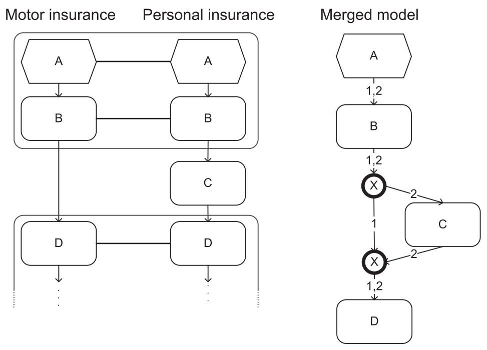

Fig. 15. Fragment of insurance models.

5.5. 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 6,000 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 article 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 5.2). 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 15 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 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. 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 mappings that were produced automatically, and made a number of corrections amounting to around a third of the matched pairs of nodes. 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, using different naming conventions and terminology.

More generally, it is important to distinguish between "homogeneous models" developed with modeling conventions in place and "heterogeneous models" developed without such conventions. Previous work shows that the mapping algorithm performs well for homogeneous models, but not well for heterogeneous ones. A particularly hard problem with heterogeneous models is that activities may be modeled at different levels of granularity, such that one activity in one process model may correspond to multiple activities in another process model. Existing matching algorithms have problems with detecting such complex correspondences [Dijkman et al. 2009, 2011; Weidlich et al. 2010]. The set of models used in this case study were to a large extent homogeneous (barring minor deviations from the established modeling conventions), such that the algorithm returned results that were considered a useful starting point by the analysts.

After this pilot study, 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.

The team at Suncorp also recognized the value of the digests of the merged models, as a means to prioritize candidate regions for merging. In fact, these digests represent those common regions that belong to the largest number of process variants, and as such, are likely to most benefit from consolidation.

6. OUTLOOK: EVOLUTION OF MERGED MODELS

As discussed in Section 1, one of the use cases of merged models is to support the synchronized coevolution of multiple process variants. While the focus of this article is on the construction of merged models and not on their evolution, we outline here a set of change primitives for merged models. We envision these change primitives as a foundation for building up tool support for coevolution of process variants.

— *Adding a Node*. This operator creates a new (nonconfigurable) node with a given label and type, and with an empty γ annotation.
— *Removing a Node*. This operator removes a given node from the model. The removed node must be "orphan" (no adjacent edges).
— *Adding a Node Annotation*. This operator takes a node x as input, a reference to a variant pid and a label l, and adds (pid, l) to function $\gamma(x)$. If a label for pid already exists in $\gamma(x)$, this is updated with the new label l. Moreover, if x is a connector and there already exists another pid in $\gamma(x)$, x becomes configurable, and if the labels associated with the different pid's are different, x takes type "or".
— *Removing a Node Annotation*. This operator takes a node x as input, a reference to a variant pid and removes this reference and the associated label l from $\gamma(x)$.

— *Adding an Edge*. This operator takes as input two nodes and creates an edge between them (assuming no such edge already exists). The new edge has an empty annotation α.
— *Removing an Edge*. The reverse of adding an edge.
— *Adding an Edge Annotation*. This operator takes as input an edge (x,y) and a reference to a variant *pid*, and adds *pid* to the annotation $\alpha(x,y)$.
— *Removing an Edge Annotation*. The reverse of adding an edge annotation.

Additionally, we propose a *cleaning* operator to be used after the application of the above primitives. This operator deletes dangling edges and orphan nodes in the merged model and ensures that the variants that can be derived from the merged model are syntactically correct in the sense that every edge is on a path from a start to an end node. The cleaning operator works as follows (cf. Algorithm 8).

(1) Given a node x, if an annotation *pid* appears in one of its incoming edges but not in any of its outgoing edges, delete annotation *pid* from the incoming edge it appears in. Vice-versa, if an annotation *pid* appears in one of the outgoing edges of x but not in any of its incoming edges, delete annotation *pid* from the outgoing edge it appears in (lines 3–6 of Algorithm 8).
(2) For each node x, update $\gamma(x)$ so that it only refers to annotations that appear in its incoming and outgoing edges. If x is a connector and $\gamma(x)$ contains one pair (pid, l) only, make x nonconfigurable and change its label to l (lines 7–12).
(3) Remove edges that have an empty annotation (lines 14–18).
(4) Remove edges that are not on a path from a start to an end node (lines 19–22).
(5) Remove trivial connectors, including those that are not configurable (line 23).
(6) Remove orphan nodes (this step is not reflected in Algorithm 8 because, for convenience, we adopted a representation of process graphs that does not explicitly capture the set of nodes).

Steps (1) and (4) ensure that in each variant *pid*, every node is on a path from a start node to an end node. Step (1) ensures that if an annotation appears in the source node of an edge it also appears in one of this node's incoming edges and similarly, if it appears in the target node of an edge, it also appears in one of this node's outgoing edges. This ensures that in each individualization, the paths emanating from a start node are not broken before reaching an end node. Meanwhile, step (4) cuts away edges that are not on a path from a start to an end node in the merged model itself. These latter edges may arise because of Step (3) and because of applying the primitive for removing edges.

Consider Figure 1 as an example, and imagine we delete annotation 2 from the edge between "Order generated and delivery opened" and the subsequent XOR-join. As a result of this deletion, the edge's annotation becomes empty and the edge is deleted. The XOR connector just above node "Delivery" then becomes a trivial connector and it is thus removed and replaced with a single edge from "Delivery is to be created" to "Delivery". This completes the cleaning. The updated version of variant G_2 can then be derived via individualization.

Let us consider another change scenario. Starting from the merged model in Figure 1, annotation 2 is deleted from the edge between node "Delivery is relevant for shipment" and its preceding XOR-join. The cleaning operator deletes annotation 2 from the outgoing edge of node "Delivery is relevant for shipment" and consequently also from the incoming edge of the XOR-join preceding this node (step (1)). This latter edge gets an empty annotation and is thus removed (step (3)). As a result, the connectors linked by this edge (the AND-split and the XOR-join) become trivial and are removed as well (step (5)). This leads to model CG' in Figure 16. If this model is

Algorithm 8: Cleaning

```
1  function Cleaning({Edge} CG)
2  begin
3    while exists a node x in $N_{CG}$ with $A_{\bullet x} \Leftarrow \bigcup_{y \in \bullet x} \alpha(y,x)$ and $A_{x\bullet} \Leftarrow \bigcup_{z \in x\bullet} \alpha(x,z)$, such that
     there exists a process identifier pid in $A_{\bullet x} \cup A_{x\bullet}$ such that pid $\notin A_{\bullet x} \cap A_{x\bullet}$ do
4      foreach a in $A_{\bullet x} \cup A_{x\bullet}$ do
5        $a \Leftarrow a \setminus \{pid\}$
6      end
7      $\gamma(x) \Leftarrow \gamma(x) \setminus \{(pid, l)\}$
8      if $\tau(x) =$"c" and $|\gamma(x)| = 1$ then
9        $\eta(x) \Leftarrow$ false
10       $(pid, l) \Leftarrow$ Any($\gamma(x)$)
11       $\lambda(x) \Leftarrow l$
12     end
13   end
14   foreach (x, y) in CG do
15     if $(\alpha(x,y) = \varnothing)$ then
16       $CG = CG \setminus \{(x,y)\}$
17     end
18   end
19   $N^S_{CG} \Leftarrow \{x \in N_{CG} : \bullet x = \varnothing\}$, $N^E_{CG} \Leftarrow \{x \in N_{CG} : x\bullet = \varnothing\}$
20   while exists an edge $(x,y)$ in CG such that there not exists a path $p = m \hookrightarrow n$ in $CG^*$ such
     that $m \in N^S_{CG}$, $n \in N^E_{CG}$, $x \in \{p\}$ and $y \in \{p\}$ do
21     $CG = CG \setminus \{(x,y)\}$
22   end
23   RemoveTrivialConnectors*(CG)
24   return CG
25 end
```

individualized for variant 2, we obtain model G'_2 in Figure 16. This model comprises two disconnected fragments. In such cases, we envisage that the user will be warned. The user may reconnect the fragments by adding edges according to the intended business logic, or split the disjoint fragments into separate models.

We observe that a supporting tool does not need to apply the cleaning operator after every application of a change primitive. Instead, the tool may allow a user to perform multiple changes before cleaning. One could envisage that the tool would allow the user to view what changes would the cleaning operator perform.

The change primitives and the cleaning operator preserve traceability since they maintain the annotations in the edges and nodes. Reversibility is preserved in the sense that changes in the merged model are reflected in the variants via individualization. Indeed, deleting an annotation *pid* from an edge *e* in the merged model leads to this edge not being part of the individualized model corresponding to *pid*. In other words, this deletion is implicitly propagated to the variant obtained via individualization. Similarly, adding an annotation *pid* to an edge in the merged model entails adding an edge in variant *pid*. Similar remarks can be made about the remaining change primitives. Behavior-preservation is also preserved in the sense that a configurable model subsumes the behavior of its variants since every path in a variant is obtained from a path in the configurable model.

In practice, users might not exclusively rely on the change primitives outlined above, but on other higher-level operations. In particular, we envision operations to delete an edge from all the variants where it appears in or to insert an edge in all variants at

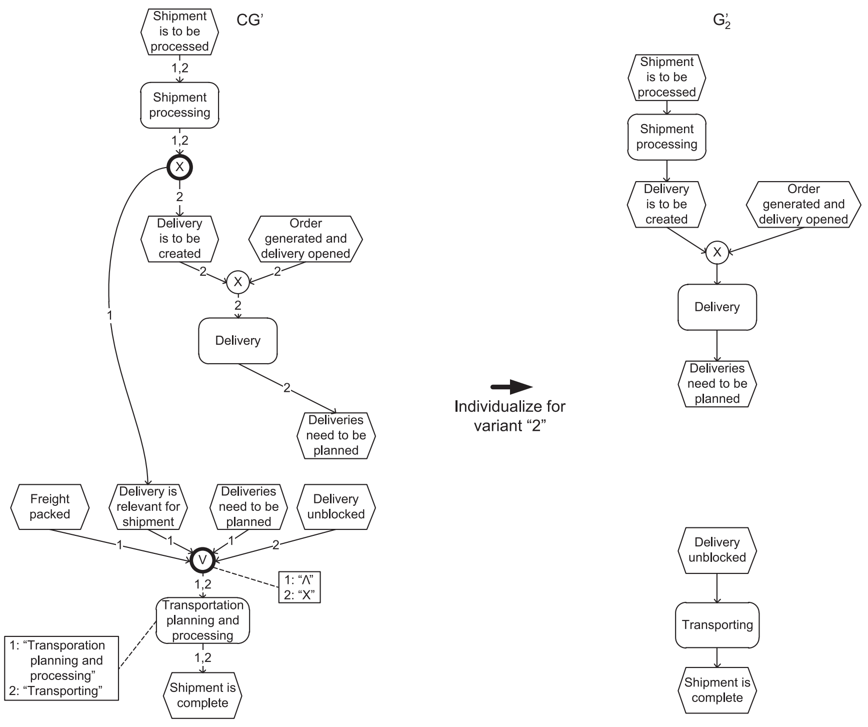
CG'
G'2
Shipment is to be processed
Shipment processing
Delivery is to be created
Order generated and delivery opened
Delivery
Deliveries need to be planned
Individualize for variant "2"
Freight packed
Delivery is relevant for shipment
Delivery unblocked
1: "∧"
2: "X"
Transportation planning and processing
1: "Transporation planning and processing"
2: "Transporting"
Transporting
Shipment is complete

Fig. 16. Modified version of the merged model of Figure 1 and individualization for variant 2.

once. Such operations can be defined as macros on top of these change primitives and on the cleaning operator.

7. RELATED WORK

The problem of merging process models has been previously posed by several authors. Sun et al. [2006] address the problem of merging block-structured Workflow nets. Their approach starts from a mapping between tasks of the input variants. Mapped tasks are copied into the merged model, and regions where the two variants 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. [2008b] outline requirements for a process merging tool for version conflict resolution. Their 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. [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. [2010] 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 [Küster et al. 2007; Mendling and Simon 2006]. 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 [2006] propose but do not implement a merging operator that produces a merged EPC from two different EPCs each representing a process view, based on a mapping of their correspondences. 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 a rule is not changed.

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 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 (process) model merging.

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. [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 that 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. [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 split and merge connectors.

Model merging is also related to database schema integration [Rahm and Bernstein 2001]. In this latter domain, numerous techniques for integrating heterogeneous database schemas 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. A key step in schema integration is to identify a suitable mapping between schema elements. This problem is related to the problem of computing a mapping between two process models. There are, however, important differences between process models and data schemas. First, data schemas generally have labeled edges (e.g., associations in an entity-relationship diagram) in addition to having labeled nodes. Meanwhile, edges in process models generally are unlabeled. This difference is relevant because schema matching techniques rely heavily on semantic information captured in the edge labels [Do and Rahm 2002; Melnik et al. 2002]. Second, the types of nodes and the attributes attached to nodes are different in process models when compared to data schemas (e.g., there are no control nodes in data schemas). Because of these differences, process model matching requires slightly different techniques as explained in Dijkman et al. [2011]. It is worth noting that automatic matching of schemas does not lead to 100% correct mappings [Mitra et al. 1999]. Instead, user intervention is required to fine-tune the mappings generated by schema matching algorithms. We encounter the same problem when merging process models.

This article is an extended and revised version of our previous work [La Rosa et al. 2010]. The 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, a more detailed case study and the framework for evolution of merged models.

8. CONCLUSION

The main contribution of this article 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 nonconfigurable process models as input, it can be used for multiway 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 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 article, we reused and adapted a previously proposed technique to automatically generate such mappings. This technique is not fail-proof. The generated mappings need to be verified and fine-tuned by an analyst. The manual effort required for such fine-tuning depends on whether or not the input models were developed with strict modeling conventions in place. Furthermore, the matching technique employed generates 1:1 mappings. When modeling conventions are not strictly followed, it may happen that the models to be merged are defined at different levels of granularity. In this case, one node in a model may have to be mapped to multiple nodes in the other model [Weidlich et al. 2010]. Extending the merging algorithm to deal with 1:N and N:M mappings is an avenue for future work.

In this article, we used a greedy algorithm for computing the initial mapping. In previous work, we have shown that this algorithm strikes a trade-off between computational complexity and quality of the computed mappings [Dijkman et al. 2011]. However, we observed that the greedy matching algorithm leads to entanglements in the merged models, because the greedy strategy makes local choices. We addressed this issue by removing pairs of nodes in the mapping so that the resulting mapping does not lead to entanglements. However, in doing so, we lower the matching score of the mapping. As an alternative, we could use a more exhaustive search algorithm—as outlined in Dijkman et al. [2011]—that explores a larger set of mappings and that would possibly lead to mappings with higher matching scores and no entanglement. However, such algorithm would be exponential. The experiments reported in this article show that the penalty of removing entanglements (in terms of lower compression ratios) is within acceptable ranges, so applying a more complex algorithm might not always be justified. Exploring the trade-off between the complexity of the mapping algorithm, the quality of the produced mappings and the desire to reduce entanglement is another direction for future work.

The empirical evaluation reported in this article focused on the case where only two models are merged at a time. While the proposed merging operator can be used for multi-way merging (i.e., merging more than two models at once) it is conceivable that multi-way merging may lead to merged models that are difficult to understand. Accordingly, another avenue for future work is to explore the understandability of models resulting from multiway merging. Specifically, it would be interesting to determine if understandability decreases beyond acceptable levels after a given number of input models are merged.

Finally, further work is required in order to support the coevolution of process variants based on a merged model. We outlined a set of change primitives and a cleaning operator over merged models. The proposed cleaning operator is designed to ensure a syntactic correctness criterion on the merged models and on the variants. More sophisticated versions of the cleaning operator could be defined to guarantee semantic correctness (e.g., deadlock-freeness) in addition to syntactic correctness. A possible theoretical foundation for a soundness-preserving cleaning operator is given in Aalst et al. [2010]. Further research is needed to define such a cleaning operator and to integrate it into a change management framework for merged models, and ultimately, to achieve the vision of consolidated management of process variants.

Appendix A. Notation

Notation	Meaning
$\gamma_G(x) \oplus \gamma_G(y)$	Assuming x and y are two connectors, $\oplus$ returns the union of their annotations γ_G, except that if there are two different connector labels for the same *pid*, e.g. (1, "xor") and (1, "and"), the pair (pid,"xor") is discarded
$\{p\}$	Set of nodes in path p
$\bullet x$, $x\bullet$	Preset of node x, postset of node x
$x \hookrightarrow y$	Path from node x to node y
$x \overset{c}{\hookrightarrow} y$	Connector chain, i.e., path of connectors from node x to node y
$\alpha_G(x,y)$	Returns the annotation of edge (x,y) in graph G, i.e. the set of process graph identifiers *pid* assigned to (x,y) in G. Subscript G can be omitted if clear from the context
$\gamma_G(x)$	Returns the annotation of node x in graph G, i.e. the set of pairs (pid,l) where *pid* is a process graph identifier and l is the label of x in graph *pid*. Subscript G can be omitted if clear from the context
$\eta_G(x)$	Returns true if connector x is configurable, false otherwise. Subscript G can be omitted if clear from the context
$\lambda_G(x)$	Returns the label of node x in graph G. Subscript G can be omitted if clear from the context
$\tau_G(x)$	Returns the type of node x in graph G. Subscript G can be omitted if clear from the context
$A_{\bullet x}$ $(A_{x\bullet})$	Union of all annotations α of all incoming (outgoing) edges of node x
$Any(X)$	Returns the only node of a singleton set X
CG	Configurable graph obtained by merging G_1 and G_2
fG_1	Source node in G_1 of a maximum common region between G_1 and G_2
G_1, G_2	Graphs to be merged
$IsAdded(x,y)$	Returns true if x or y is a configurable XOR added by the merging algorithm
lG_1	Sink node in G_1 of a maximum common region between G_1 and G_2
$M(x)$	Returns the node in G_2 that is matched with the node x in G_1
mcr	Element of *MCRs* (Algorithm 1)
$MCRs$	Set of maximum common regions between G_1 and G_2
N_G	Set of nodes of graph G
N_G^S (N_G^E)	Set of start nodes, i.e. nodes with an empty preset (end nodes, i.e. nodes with an empty postset) of graph G
$Node(t,l,b)$	Returns a new node of type t and label l, which is configurable if b is true
P_c	Set of predecessor nodes of $M(c)$ which are not matched
pfG_1 (pfG_2)	Predecessor node of fG_1 $(M(fG_1))$ in G_1 (G_2)
pm	Predecessor node of node m
$Pid(G)$	Returns the process graph identifier of graph G
S_c	Set of successor nodes of $M(c)$ which are not matched
slG_1 (slG_2)	Successor node of lG_1 $(M(lG_1))$ in G_1 (G_2)
sm	Successor node of node m
$skipe$	Set edges in G_1 not matched to an edge in G_2 or vice-versa
$skipn$	Set nodes in G_1 not matched to a node in G_2 or vice-versa
$sube$ $(subn)$	Set of matched edges (nodes) between G_1 and G_2
xs (xj)	Connector node of type XOR-split (XOR-join)
$wskipe$ $(wskipn)$	Weight attached to *skipe* (*skipn*)
$wsube$ $(wsubn)$	Weights attached to *sube* (*subn*)

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 article.

REFERENCES

Aalst, W., Dumas, M., Gottschalk, F., Hofstede, A., La Rosa, M., and Mendling, J. 2010. Preserving correctness during business process model configuration. *Form. Aspects Comput. 22,* 3, 459–482.

Bunke, H. 1997. On a relation between graph edit distance and maximum common subgraph. *Patt. Rec. Lett. 18*, 8, 689–694.

Dijkman, R., Dumas, M., García-Banuelos, L., and Käärik, R. 2009. Aligning business process models. In *Proceedings of EDOC*. IEEE Computer Society, 45–53.

Dijkman, R., Dumas, M., van Dongen, B., Uba, R., and Mendling, J. 2011. Similarity of business process models: Metrics and evaluation. *Inf. Syst. 36*, 2, 498–516.

Do, H. H. and Rahm, E. 2002. Coma - A system for flexible combination of schema matching approaches. In *Proceedings of VLDB*. Morgan Kaufmann, 610–621.

Gottschalk, F., van der Aalst, W. M. P., and Jansen-Vullers, M. H. 2008. Merging event-driven process chains. In *Proceedings of CoopIS*. Lecture Notes in Computer Science, vol. 5331, Springer, 418–426.

Küster, J., Ryndina, K., and Gall, H. 2007. Generation of business process models for object life cycle compliance. In *Proceedings of BPM*. Lecture Notes in Computer Science, vol. 4714, Springer, 165–181.

Küster, J., Gerth, C., Förster, A., and Engels, G. 2008a. Detecting and resolving process model differences in the absence of a change log. In *Proceedings of BPM*. Lecture Notes in Computer Science, vol. 5240. Springer, 244–260.

Küster, J., Gerth, C., Förster, A., and Engels, G. 2008b. A tool for process merging in business-driven development. In *Proceedings of the CAiSE'2008 Forum*. CEUR Workshop Proceedings, vol. 344, 89–92.

La Rosa, M., Dumas, M., ter Hofstede, A., and Mendling, J. 2011. Configurable multi-perspective business process models. *Inf. Syst. 36*, 2, 313–340.

La Rosa, M., Dumas, M., Uba, R., and Dijkman, R. 2010. Merging business process models. In *Proceedings of CoopIS*. Lecture Notes in Computer Science. Springer.

Levenshtein, I. 1966. Binary code capable of correcting deletions, insertions and reversals. *Cybernet. Control Theory 10*, 8, 707–710.

Li, C., Reichert, M., and Wombacher, A. 2010. The minadept clustering approach for discovering reference process models out of process variants. *Int. J. Coop. Inf. Syst. 19*, 3–4, 159–203.

Melnik, S., Garcia-Molina, H., and Rahm, E. 2002. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of ICDE*. IEEE Computer Society, 117–128.

Mendling, J. 2009. *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*. Lecture Notes in Business Information Processing, vol. 6, Springer.

Mendling, J. and Simon, C. 2006. Business process design by view integration. In *Proceedings of BPM Workshops*. Lecture Notes in Computer Science, vol. 4103, Springer, 55–64.

Mendling, J., Reijers, H. A., and Cardoso, J. 2007. What makes process models understandable? In *Proceedings of BPM*. 48–63.

Mendling, J., Reijers, H., and van der Aalst, W. 2010. Seven Process Modeling Guidelines (7PMG). *Inf. Softw. Tech. 52*, 2, 127–136.

Mens, T. 2002. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng. 28*, 5, 449–462.

Mitra, P., Wiederhold, G., and Jannink, J. 1999. Semi-automatic integration of knowledge sources. In *Proceedings of FUSION*.

Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., and Zave, P. 2007. Matching and merging of statecharts specifications. In *Proceedings of ICSE*. IEEE Computer Society, 54–63.

Ohst, D., Welle, M., and Kelter, U. 2003. Differences between versions of UML diagrams. In *Proceedings of ESEC/SIGSOFT FSE*. ACM, 227–236.

Pedersen, T., Patwardhan, S., and Michelizzi, J. 2004. WordNet: Similarity - Measuring the Relatedness of Concepts. In *Proceedings of AAAI*. AAAI, 1024–1025.

Rahm, E. and Bernstein, P. 2001. A survey of approaches to automatic schema matching. *VLDB J. 10*, 4, 334–350.

Reijers, H. A., Mans, R. S., and van der Toorn, R. A. 2009. Improved model management with aggregated business process models. *Data Knowl. Eng. 68*, 2, 221–243.

Rosemann, M. and van der Aalst, W. M. P. 2007. A configurable reference modelling language. *Inf. Syst. 32*, 1, 1–23.

Salomon, D. 2006. *Data Compression: The Complete Reference* 4th Ed. Springer.

Sun, S., Kumar, A., and Yen, J. 2006. Merging workflows: A new perspective on connecting business processes. *Decis. Supp. Syst. 42*, 2, 844–858.

van Dongen, B., Dijkman, R., and Mendling, J. 2008. Measuring similarity between business process models. In *Proceedings of CAiSE*. Lecture Notes in Computer Science, vol. 5074, Springer, 450–464.

Weidlich, M., Dijkman, R., and Mendling, J. 2010. The ICoP framework: Identification of correspondences between process models. In *Proceedings of CAiSE*. Lecture Notes in Computer Science, vol. 6051, Springer.

Received October 2010; revised May 2011, October 2011, December 2011; accepted January 2012