Merging Event-driven Process Chains

Florian Gottschalk, Wil M.P. van der Aalst, Monique H. Jansen-Vullers

Eindhoven University of Technology, The Netherlands.
{f.gottschalk,w.m.p.v.d.aalst,m.h.jansen-vullers}@tue.nl

Abstract. A main goal of corporate mergers is to gain synergy effects through the alignment, integration, or combination of business processes. While business processes are typically handled differently among companies, many of them are variations of a common process like, e.g., procurement or invoicing. The first goal for process analysts when aligning business processes is thus to identify and seamlessly integrate overlapping process parts. To support this we present in this paper an approach that merges two business process models which are depicted as Event-Driven Process Chains into a single process model without restricting the behavior that was possible in both the original models. The resulting model can serve as a starting point for further process optimization. The approach has been implemented in the ProM process mining framework. Therefore, it can be used together with a wide range of other process mining and analysis techniques.

Key words: EPC, process alignment, process model integration

1 Introduction

Mergers, take-overs, or acquisitions even of huge cooperations are common headlines in today's news. Across all businesses – from airlines to banks, from the computer to the food industry – cooperations join forces not only to gain market shares and thus to use economies of scale, but also to cut costs through synergy effects. To achieve this, the combined company aims at reducing duplicate departments or operations by merging the business processes of the companies.

Especially secondary and supporting processes like purchasing, invoicing, or HR processes are good candidates for this as they are executed similarly among companies. However, they are hardly identical. Thus, merging such similar business processes still requires a process analysis of the as-is processes, an identification and mapping of identical tasks among the processes, and finally based on this knowledge the creation of an integrated business process applicable for both of the merging corporations.

In this paper, we focus on this last aspect by providing an algorithm that integrates two process models into one simple process model which still allows for the behavior possible in any of the original models. Traditionally, such a merge of business process models is performed "in the mind of the process analyst", i.e. the process analysts builds a new process model after comparing the input models manually. While building such a new, integrated model from scratch allows for neglecting all aspects which are considered as irrelevant for the new process by the process analysts, it comes with the risk that some aspects of the one or the other process are accidentally neglected. This then might prevent the performance of some process parts which are essential for the products of one of the companies. By integrating the original process models automatically such that all the behavior depicted in the original models remains possible, this risk can be minimized. Process analysts are then not burdened with the integration of the control-flow of the models, but can rather use the integrated model as starting point for an optimization of the process.

To make the resulting process models easily understandable to business analysts, we based our work on Event-driven Process Chains (EPCs), a common, and easy-to-understand process modelling language [12], also supported by commercial process modelling tools like ARIS from IDS Scheer [13] or Microsoft Visio. We therefore start this paper in the next section with a formal definition of EPCs as a prerequisite for the merge algorithm. Afterwards Section 3 describes the algorithm to merge multiple EPCs through merging graphs depicting the relations between the EPCs' functions as well as analyzes the behavioral implications of the merge. The algorithm's implementation within the ProM process mining framework is then depicted in Section 4. The paper ends with a brief description of related work and with drawing some conclusions.

2 Prerequisites

Event-driven Process Chains (EPCs) basically consist of three node types, namely events, functions and logical connectors, which are connected by directed arcs as depicted in Figure 1. Events are passive elements depicting prerequisites for or results from the execution of functions which are the active elements.

The logical connectors determine the control-flow behavior in case the control-flow is split up into or joined from several parallel or alternative execution branches. If a connector of an EPC splits the control-flow up and is of type *XOR* and has n successors this means that at runtime one out of its n succeeding arcs is followed. If the connector is of type $\wedge$ and has n succeeding arcs, then all n of them are followed. If the split connector is of type $\vee$, any m out of the n succeeding arcs can be triggered while m can have any value between 1 and n. Thus, the $\wedge$ and the *XOR* connectors are specializations of the $\vee$ connector, i.e. an $\vee$ connector allows for the same triggering of succeeding paths and thus for the same behavior as an *XOR* connector, it allows for the same behavior as an $\wedge$ connector, but it also allows for additional behaviors. If a connector joins the control-flow, it behaves in a similar fashion. This means, an $\wedge$ connector requires input from all incoming arcs to forward the case, an *XOR* connector requires the input just from just one incoming arc to forward a case, and an $\vee$ connector requires the input from a number of incoming arcs which is determined at run-time. Thus an $\vee$ connector can join the control-flow behavior in the same way as an $\wedge$ connector and in the same way as an *XOR* connector, but also as a partial synchronizer.

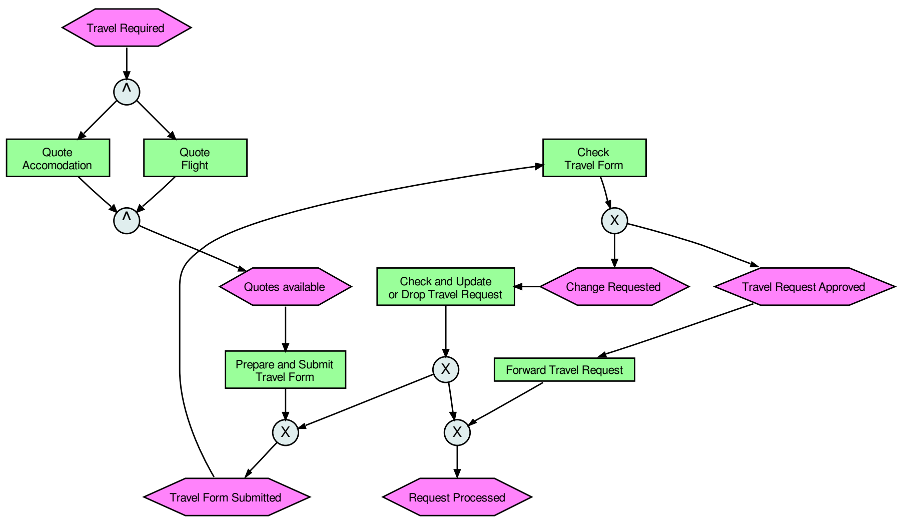


Fig. 1. An EPC – Hexagons depict events, rectangles depict functions, and circles depict connectors

After it has been triggered by a travel request, the process in Figure 1 thus requires both an accommodation quote and a flight quote. Both quotes need to be available before the process can pass the subsequent $\wedge$ connector and can continue with the preparation and submission of the travel form. After the form has been checked, it can either be accepted or a change can be requested – as indicated by the *XOR* connector subsequent to the "Check Travel Form" function. If a change is requested, a choice exists if the form is either updated or dropped. If it is updated, it re-joins the control-flow as if it would have been submitted as a new travel form. If it is dropped, there is no need for further processing. If the travel request is accepted, it is forwarded to the clearing center and the processing of the request finishes as well.

The following formal EPC definitions are in line with the definitions in [9]:

Definition 1 (EPC). *An Event-driven Process Chain is a five-tuple* (E, F, C, l, A)*:*

– E *is a finite (non-empty) set of events,*
– F *is a finite (non-empty) set of functions,*
– C *is a finite set of connectors,*
– $l \in C \rightarrow \{\wedge, XOR, \vee\}$ *is a function which maps each connector onto a connector type,*
– $A \subseteq (E \times F) \cup (F \times E) \cup (E \times C) \cup (C \times E) \cup (F \times C) \cup (C \times F) \cup (C \times C)$ *is a set of arcs.*

Although Definition 1 already shows that arcs of an EPC cannot connect two events or two functions directly, a well-formed EPC has to satisfy some further requirements. To formalize them, we need to define some additional notations.

Definition 2 (EPC notations). *Let $EPC = (E, F, C, l, A)$ be an Event-driven Process Chain.*

- *$N = E \cup F \cup C$ is the set of nodes of EPC,*
- *For $n \in N$:*
  - *$\bullet n = \{m | (m, n) \in A\}$ is the set of input nodes, and*
  - *$n\bullet = \{m | (n, m) \in A\}$ is the set of output nodes,*
- *$C_J = \{c \in C | \ |\bullet c| \geq 2\}$ is the set of join connectors,*
- *$C_S = \{c \in C | \ |c \bullet| \geq 2\}$ is the set of split connectors,*
- *A directed path p from a node n_1 to a node n_k is a sequence $\langle n_1, ..., n_k\rangle$ such that $(n_i, n_{i+1}) \in A$ for $1 \leq i \leq k - 1$.*
- *$C_{EF} \subseteq C$ such that $c \in C_{EF}$ if and only if there is a path $p = \langle n_1, n_2, ..., n_{k-1}, n_k\rangle$ such that $n_1 \in E, n_, ..., n_{k-1} \in C, n_k \in F$, and $c \in \{n_2, ..., n_{k-1}\}$,*
- *$C_{FE} \subseteq C$ such that $c \in C_{FE}$ if and only if there is a path $p = \langle n_1, n_2, ..., n_{k-1}, n_k\rangle$ such that $n_1 \in F, n_, ..., n_{k-1} \in C, n_k \in E$, and $c \in \{n_2, ..., n_{k-1}\}$,*
- *$C_{EE} \subseteq C$ such that $c \in C_{EE}$ if and only if there is a path $p = \langle n_1, n_2, ..., n_{k-1}, n_k\rangle$ such that $n_1 \in E, n_, ..., n_{k-1} \in C, n_k \in E$, and $c \in \{n_2, ..., n_{k-1}\}$,*
- *$C_{FF} \subseteq C$ such that $c \in C_{FF}$ if and only if there is a path $p = \langle n_1, n_2, ..., n_{k-1}, n_k\rangle$ such that $n_1 \in F, n_, ..., n_{k-1} \in C, n_k \in F$, and $c \in \{n_2, ..., n_{k-1}\}$.*

We use the set of input and output nodes to define that each event is at maximum preceded by one input node and at maximum succeeded by one output node while each function has exactly one input and one output node. Further on, each well-formed EPC needs at least one start event that is not preceded by any other node and one end event that is not succeeded by any other node. Connectors must have at least one input and one output node. They can also have several input nodes or several output nodes, but not both at the same time. To ensure this, we use the sets C_J and C_S and require that they partition the set of connectors C into on the one hand a set of connectors that split the control-flow up, and on the other hand a set of connectors that join the control-flow. Finally, we use directed paths to limit the set of routing constructs that can be constructed using connectors. In a well-formed EPC there should be no paths connecting two events or two functions only via connector nodes in between. This also means that each connector should be either on paths from events to functions only or on paths from functions to events only. Altogether, well-formed EPCs can be formalized as follows:

Definition 3 (Well-formed EPC). *A well-formed Event-driven Process Chain (E, F, C, l, A) satisfies the following requirements:*

1. *The sets E, F, and C are pairwise disjoint, i.e. $E \cap F = \emptyset$, $E \cap C = \emptyset$, and $F \cap C = \emptyset$,*
2. *for each $e \in E : |\bullet e| \leq 1$ and $|e \bullet| \leq 1$,*
3. *there is at least one event $e_{start} \in E$ such that $|\bullet e_{start}| = 0$,*
4. *there is at least one event $e_{end} \in E$ such that $|e_{end} \bullet| = 0$,*
5. *for each $f \in F : |\bullet f| = 1$ and $|f \bullet| = 1$,*
6. *for each $c \in C : |\bullet c| \geq 1$ and $|c \bullet| \geq 1$,*

7. C_J and C_S partition C, i.e. $C_J \cap C_S = \emptyset$ and $C_J \cup C_S = C$,
8. C_{EE} and C_{FF} are empty, i.e. $C_{EE} = \emptyset$ and $C_{FF} = \emptyset$,
9. C_{EF} and C_{FE} partition C, i.e. $C_{EF} \cap C_{FE} = \emptyset$ and $C_{EF} \cup C_{FE} = C$.

3 The Merge Algorithm

In the following we aim at merging multiple well-formed EPCs into a single well-formed EPC such that the behavior that is possible according to the new EPC, is at least the behavior that was possible in each of the original EPCs. The behavior of an EPC is the order in which the EPC's functions can be executed as the functions represent the "active" behavior. Two EPCs represent the same behavior if all the orders in which the functions of one of the EPCs can be executed are also possible in the second EPC and vice versa. For our purposes of providing a model which serves as the starting point for further process optimization we further on prefer a compact and clear model (i.e. less model elements and arcs) and therefore accept if this is on the expense of allowing some additional behavior that was not possible in any of the original EPCs. This means, the resulting model may allow for more behavior than the sum of the parts' behaviors. Hence, the merge algorithm generalizes.

To preserve the behavioral character of initial, i.e. the first executed, and final, i.e. last executed, functions we require in the following that the initial and final functions of the two EPCs are unique among the two models. Otherwise it would be possible that the merge algorithms incorporates functions which are first or last executed in one of the EPCs into the process flow of the other EPC because the same function is executed somewhen in the middle of this other EPC. If a pair of EPCs does not have such unique initial or final functions, a unique dummy function not representing any real behavior can simply be added in front of the initial functions and after the final functions of the original EPCs. Without any behavioral meaning, the dummy function simply guarantees the presence of a unique entry point of the process and thus preserves the behavioral requirements of initial or final functions.

The algorithm we suggest for merging two EPCs conducts the merge in three phases. At first we reduce the original EPCs to their active behavior, i.e. to their functions, and represent them in form of models which we call *function graphs*. Afterwards, the behavior represented by the two resulting function graphs is merged into a new function graph which represents the combined behavior. And last, the resulting function graph is then converted back into an EPC.

3.1 From EPCs to Function Graphs

A function graph is a reduction of an EPC to its functions as sole nodes. The functions can be connected through directed arcs, meaning that the source function of the arc can be executed before the target function. Both a *split type* and a *join type* is assigned to each arc. Both types can have either the value $\wedge$, *XOR*, or $\vee$. Some examples for function graphs are given in the second row of Figure 2.

Definition 4 (Function graph). *A Function graph is a four-tuple* (F, A, l_J, l_S)*:*

- F *is a finite (non-empty) set of functions,*
- $A \subseteq (F \times F)$ *is a set of arcs,*
- $l_J \in A \rightarrow \{\wedge, XOR, \vee\}$ *is a function which maps each arc onto a join relation type such that* $\forall_{f_1,f_2,f_3 \in F}((f_1, f_2) \in A \wedge (f_3, f_2) \in A \wedge l_J(f_1, f_2) = \wedge) \Rightarrow (l_J(f_3, f_2) \neq XOR)$*,*
- $l_S \in A \rightarrow \{\wedge, XOR, \vee\}$ *is a function which maps each arc onto a split relation type such that* $\forall_{f_1,f_2,f_3 \in F}((f_1, f_2) \in A \wedge (f_1, f_3) \in A \wedge l_J(f_1, f_2) = \wedge) \Rightarrow (l_J(f_1, f_3) \neq XOR)$*.*

The behavior of a function graph is as follows. After a completed execution, the functions in a function graph mark some outgoing arcs with tokens. Which of the arcs are marked depends on the split types. If an arc has an $\wedge$ split type, a token has to be added to this arc after the execution of its source function. In case of an *XOR* split type, a token can only be added to the arc if none of the other arcs succeeding the source function receives a token at the same time. The $\vee$ type depicts that the marking of the arc after the source function's execution is optional. However, if a function is the source of a set of arcs, at least one of these arcs must be marked with a new token after any of the function's executions.

We require for the split types that a function is never the source of an arc of type $\wedge$ and of an arc of type *XOR* at the same time. From a logical point of view, such type values would imply that the arc with the *XOR* type could never be marked as the arc of type $\wedge$ must always be marked with a token. That means, the arcs of type *XOR* would never be followed and should thus be omitted.

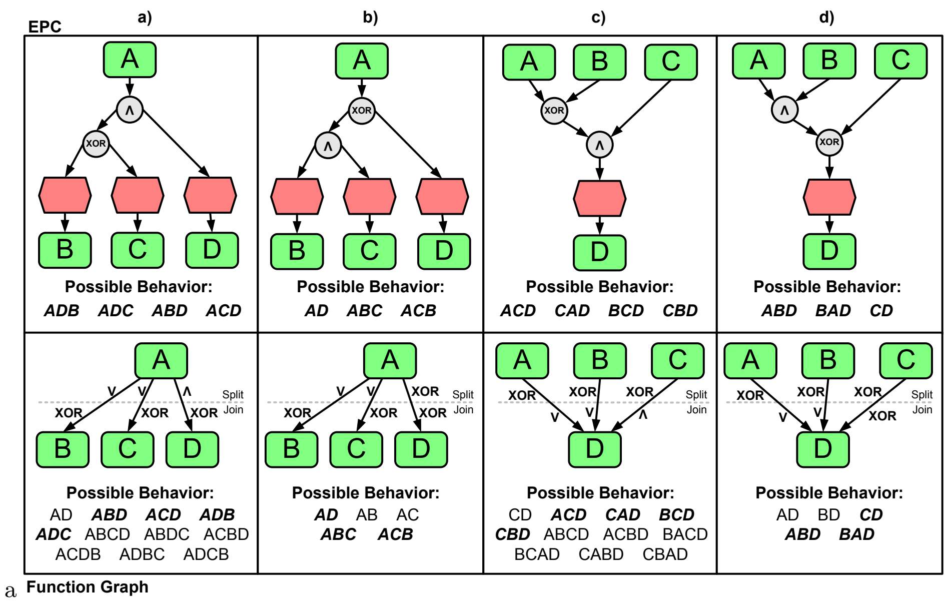


Fig. 2. The relation between EPCs and function graphs

The execution of a (non-initial) function then depends on the tokens on its incoming arcs. The join types of the arcs determine when their joined target function can be executed. In case an arc is assigned the join type $\wedge$, the target function needs to consume a token from that arc when it is executed, i.e. for the function's execution this arc must be marked with at least one token. An arc of type *XOR* means that the target function can be executed if this arc is marked with a token, independent of the markings of the other incoming arcs. In this case this token is then also the only token consumed during the function's execution. An $\vee$ join type means that the consumption of a token from that arc is optional for the execution of its target function. In any case, however, a function with at least one incoming arc can only be executed if it consumes at least one token from one of its incoming arcs.

For the join types we require that a function is never target of an arc of type $\wedge$ and of an arc of type *XOR* at the same time. In such a case, the type values would imply that the function must always consume a token from the arc of type $\wedge$ and thus can never consume a token from the arc of type *XOR* exclusively. As this means that these tokens would never be consumed, it is not necessary to produce them and the arc should be omitted.

When transforming an EPC into a function graph, the functions in the function graph are the same functions as in the EPC. Every function in the function graph is connected by arcs to exactly those functions which could be reached in the EPC through a path which does not pass any other function. The arcs' join and split types are calculated based on the values of the join/split connectors along the corresponding path in the EPC. If no connectors exist along the path or if all these connectors are of type *XOR*, the corresponding arc is assigned the type *XOR*. If all the connectors are of type $\wedge$, the corresponding arc is assigned the type $\wedge$. Otherwise the value $\vee$ is assigned to the arc.

Definition 5 (Function graph from EPC). *Let (E, F, C, l, A) be a well-formed EPC. Then (F, A^F, l_J^F, l_S^F) is a function graph with*

- $A^F = \{(x,y) \in (F \times F) | \exists_{z_1,...,z_n \in C \cup E} \langle x, z_1, ..., z_n, y\rangle \in A^*\}$[1]
- $l_J^F \in A^F \rightarrow \{\wedge, XOR, \vee\}$ *is a function which maps each arc onto a join relation type as follows:*

$$l_J^F((x,y)) = \begin{cases} XOR \ \textit{if} \ \forall_{n\geq 0} \forall_{z_1,...,z_n \in C\cup E} \langle x, z_1, ..., z_n, y\rangle \in A^* \Rightarrow \forall_{1\leq i \leq n, z_i \in C_J} \\ \qquad l(z_i) = XOR \\ \wedge \qquad \textit{if} \ \forall_{n\geq 1} \forall_{z_1,...,z_n \in C\cup E} \langle x, z_1, ..., z_n, y\rangle \in A^* \Rightarrow \forall_{1\leq i \leq n, z_i \in C_J} \\ \qquad l(z_i) = \wedge \\ \vee \qquad \textit{otherwise,} \end{cases}$$

[1] A^* is the set of sequences over A, i.e, $\langle z_1, ..., z_n\rangle \in A^*$ if and only if $\forall_{1\leq i<n}(z_i, z_{i+1}) \in A$.

– $l_S^F \in A^F \to \{\wedge, XOR, \vee\}$ *is a function which maps each arc onto a split relation type as follows:*

$$l_S^F((x,y)) = \begin{cases} XOR & \textit{if } \forall_{n\geq 0}\forall_{z_1,\dots,z_n \in C\cup E}\langle x, z_1, ..., z_n, y\rangle \in A^* \Rightarrow \forall_{1\leq i\leq n, z_i\in C_S} \\ & l(z_i) = XOR \\ \wedge & \textit{if } \forall_{n\geq 1}\forall_{z_1,\dots,z_n \in C\cup E}\langle x, z_1, ..., z_n, y\rangle \in A^* \Rightarrow \forall_{1\leq i\leq n, z_i\in C_S} \\ & l(z_i) = \wedge \\ \vee & \textit{otherwise.} \end{cases}$$

We use the two examples in figures 2a and 2b to show the relation between the splits of the control flow in an EPC and in the function graph as well as the possible behaviors of these control-flow splits. In both figures we analyze the behavior that can happen after a single execution of A. If all the split connectors between two functions of an EPC are of type $\wedge$, then the execution of the first function implies the execution of the second function as it is the case for functions A and D in Figure 2a. The $\wedge$ split type of the corresponding arc in the function graph therefore requires the marking of the arc after the execution of A. As this arc is the only incoming arc of D, D requires only this token for its execution. Hence, D can be executed after A in the function graph as in the EPC. If all the split connectors between two functions of an EPC are of type XOR, then the execution of the first function implies the exclusive execution of the second function among all its subsequent functions. This is the case for functions A and D in Figure 2b. The corresponding function graph thus requires that after A's execution the arc to D is marked exclusively as indicated by its XOR split type.

If the path between two functions of an EPC contains $\vee$ connectors, or if it contains connectors of different types, then the succeeding function might or might not be triggered. As an example, in the model in Figure 2a either B or C can follow A (but always in combination with D). Within the function graph, this behavior is reproduced by assigning the split value $\vee$ to the arc between A and B and also to the arc between A and C. In this way, it is optional for each of these arcs if it is marked after A's execution. The function graph thus allows for executing B or C after A as the EPC. But in addition to marking one of the two arcs, the function graph also allows marking both or none of them as the marking of an arc in a function graph is independent from the marking of other arcs. Thus in addition to the behavior of the EPC, also behaviors where both B and C or none of them follow A are possible in the function graph.

Figures 2c and 2d show the corresponding behavior relation for join connectors of an EPC. Here we analyze the behavior that leads to a single execution of D. If all the connectors join the control flow between two functions are of type $\wedge$, then the first function always has to precede the second one, as e.g. in the case of functions C and D in Figure 2c. The corresponding arc's $\wedge$ join type in the function graph also ensures that D can only be executed after C has been executed and produced a token on this arc. In case all join connectors between two functions in an EPC are of type XOR, the first function always precedes the second one exclusively. Functions C and D in Figure 2d provide an example for this. In the function graph an arc's XOR join type implies exactly this behavior. The target function requires and exclusively consumes a token from that arc. In case different join connectors exist between two functions in an EPC (or if there are only $\vee$ connectors), there are several combinations of preceding function executions possible before the execution of the succeeding function. The join type $\vee$ in the corresponding function graph thus specifies that the consumption of tokens from each corresponding arc is optional for the execution of the target function. However, although this covers the behavior of the EPC, it might also allow for executing the target function in cases for which this was not allowed in the EPC.

All in all this means that whenever the split or join type of an arc in the function graph is XOR or $\wedge$, the behavioral relation between those two functions corresponds to the one in the EPC. When the split/join type is set to $\vee$ the behavior of the EPC is also covered by the function graph. Then, however, also additional behavior might be possible.

3.2 Combining Function Graphs

The goal of this paper is to merge process models (e.g. EPCs) while preserving the original behavior. Function graphs over-approximate these behaviors and are used as a tool to merge process models.

Two function graphs can be combined by merging the sets of functions, merging the sets of arcs, and calculating the split and join types of the arcs based on the values in the two function graphs. When calculating the split and join types of an arc, it is needed to analyze if functions are succeeded or preceded by the same functions in both models. For this, let us first define shorthand notations for the preset and postset of a function in a function graph.

Definition 6 (Function graph notations). *Let (F, A, l_J, l_S) be a function graph. Then is*

- $pre_A \in F \to \mathbb{P}(F)$ *such that* $pre_A(n) = \{m|(m,n) \in A\}$ *the preset of a function* n *according to the set of arcs* A*, and*
- $post_A \in F \to \mathbb{P}(F)$ *such that* $post_A(n) = \{m|(n,m) \in A\}$ *the postset of a function* n *according to the set of arcs* A*.*

If an arc is of split type XOR in one of the two function graphs and does not exist in the other graph, or if it has type XOR in both graphs, then the arc to the target function is either marked exclusively or not marked (a non-existent arc cannot be marked). This corresponds to the behavior of an XOR split type which is thus assigned to the arc in the resulting function graph as well. The split type $\wedge$ is only assigned to an arc if it either is of split type $\wedge$ in both input function graphs or if it has the value $\wedge$ in one of the input function graphs and there is no outgoing arc at all from the arc's source function in the other function graph. In case there is an $\wedge$ split type assigned to the arc in one model and the arc does not exist in the other model, but the source function of the arc has other successor functions in the other model, the $\wedge$ value cannot be assigned to the resulting arc because it would imply that the arc must always be marked in the resulting model. This conflicts with the model without the arc where a non-existent arc can obviously not be marked. Thus, in such cases the arc gets the more general split type $\vee$ assigned as it gets in all other cases.

The join types of arcs are calculated in line with the split types. Thus, the corresponding line of argumentation also holds for the three join types of arcs. If an arc is of join type XOR in one function graphs and does not exist in the other, or if it has type XOR in both graphs, the arc is assigned the join type XOR. The join type $\wedge$ is assigned if the arc is either of split type $\wedge$ in both function graphs or if it is of type $\wedge$ in one of them and the arc's target function has no predecessors in the other function graph. In all other cases the arc is assigned join type $\vee$.

Definition 7 (Combining function graphs). *Two function graphs $g^1 = (F^1, A^1, l^1_J, l^1_S)$, $g^2 = (F^2, A^2, l^2_J, l^2_S)$ can be combined to a new function graph $g^3 = (F^1 \cup F^2, A^1 \cup A^2, l^3_J, l^3_S)$ where:*

- $l^3_S \in (A^1 \cup A^2) \to \{\wedge, XOR, \vee\}$ *is a function which maps each arc onto a split relation type as follows:*

$$l^3_S((x,y)) = \begin{cases} XOR \text{ } \mathit{if} \text{ } (l^1_S((x,y)) = XOR \text{ and } (x,y) \notin A^2) \text{ or} \\ \qquad (l^1_S((x,y)) = XOR \text{ and } l^2_S((x,y)) = XOR) \text{ or} \\ \qquad ((x,y) \notin A^1 \text{ and } l^2_S((x,y)) = XOR), \\ \wedge \qquad \mathit{if} \text{ } (l^1_S((x,y)) = \wedge \text{ and } post_{A^2}(x) = \emptyset) \text{ or} \\ \qquad (l^1_S((x,y)) = \wedge \text{ and } l^2_S((x,y)) = \wedge) \text{ or} \\ \qquad (post_{A^1}(x) = \emptyset \text{ and } l^2_S((x,y)) = \wedge), \\ \vee \qquad \mathit{otherwise}, \end{cases}$$

- $l^3_J \in (A^1 \cup A^2) \to \{\wedge, XOR, \vee\}$ *is a function which maps each arc onto a join relation type as follows:*

$$l^3_J((x,y)) = \begin{cases} XOR \text{ } \mathit{if} \text{ } (l^1_J((x,y)) = XOR \text{ and } (x,y) \notin A^2) \text{ or} \\ \qquad (l^1_J((x,y)) = XOR \text{ and } l^2_J((x,y)) = XOR) \text{ or} \\ \qquad ((x,y) \notin A^1 \text{ and } l^2_J((x,y)) = XOR), \\ \wedge \qquad \mathit{if} \text{ } (l^1_J((x,y)) = \wedge \text{ and } pre_{A^2}(y) = \emptyset) \text{ or} \\ \qquad (l^1_J((x,y)) = \wedge \text{ and } l^2_J((x,y)) = \wedge) \text{ or} \\ \qquad (pre_{A^1}(y) = \emptyset \text{ and } l^2_J((x,y)) = \wedge), \\ \vee \qquad \mathit{otherwise}. \end{cases}$$

Figure 3 shows an example for the arc values resulting from combining two function graphs. If an arc between two functions has the split type XOR assigned (as, e.g., the arc between A and C in Figure 3a), then this succeeding arc can be marked either exclusively or not at all after the execution of its source function. This behavior must be preserved in the resulting function graph. As there is no arc from A to C in Figure 3b, this function graph only requires that the arc between A and C cannot be marked during execution which is covered by the opportunity not to mark the arc of the XOR split type. Thus the corresponding arc of the resulting function graph gets the type XOR assigned. The same would hold if both input function graphs would require an XOR split type here as this would still cover all possibilities allowed by the two input graphs.

If, however, one model requires that the arc succeeding a function is exclusively marked after its execution (as the arc from A to B in Figure 3a) while the

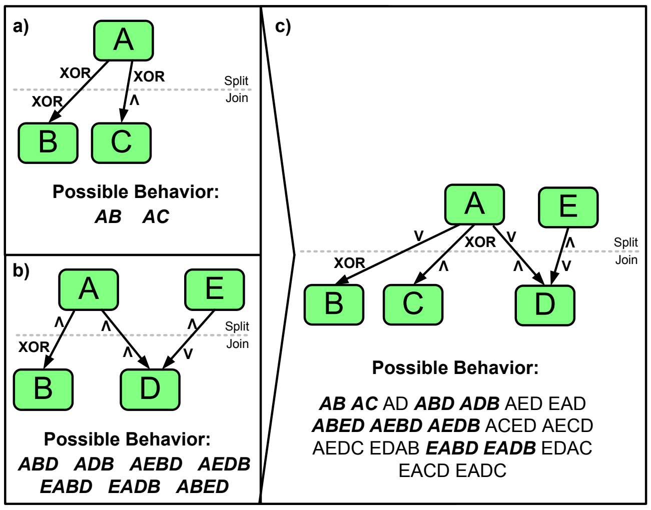


Fig. 3. Combining two function graphs and the behavior possible according to each of the function graphs such that no tokens remain on the arcs

other model allows for a combined marking of the arc with other arcs through an $\wedge$ or $\vee$ split type (as in Figure 3b), then the arc is assigned the split type $\vee$ in the resulting model (see the arc from A to B in Figure 3c). In this way, both combining the marking of this arc with other arcs (as required by Figure 3b) as well as its exclusive marking (as required by Figure 3a) remains possible as long as none of the other arcs leaving the first function is assigned the split type $\wedge$.

According to the definition of function graphs, a single function that is the the source of an arc with split type XOR cannot be source of an arc with type $\wedge$ in the same graph. However, it is well possible that a function like function A in Figure 3 is source of an arc of type XOR in one of the merged function graphs (e.g., the arc to C in Figure 3a) and in the other graph it is the source of an arc non-existent in the first graph which is of type $\wedge$ (the arc to D in Figure 3b). In this case the function is the source of the arc between functions A and D which must obligatory be marked after A's execution in the function graph of b) while this arc is not part of the graph in a). Thus, the marking of the arc cannot be made obligatory in the resulting model in c) because the obligatory marking of the arc would then conflict with the behavior of the function graph in a) which did not ask for the marking of an arc between A and D. Hence, in such cases the marking of the arc becomes optional by assigning it the $\vee$ split type. In this way also the condition that functions cannot be succeeded by arcs of split type $\wedge$ and of split type XOR at the same time is guaranteed.

In case of the join type of the arc between A and D, there is no such function D in the function graph of Figure 3a and thus also no incoming arc to D. Hence, the only way in which D can be executed is depicted in Figure 3b where it always requires a token on the arc from A as specified by the arc's $\wedge$ join type. The resulting, corresponding arc in Figure 3c can therefore preserve the $\wedge$ join type.

As the $\vee$ split and join types are less restrictive than the $\wedge$ and the XOR split/join types, using the $\vee$ for an arc which had type $\wedge$ or type XOR in one of the initial function graphs not only allows for the same behavior as in the original graphs, but also for a number of additional behaviors that were not possible in any of the original graphs. The behavior depicted in Figure 3 provides an example for this. The figure shows the behavior required and possible such that A is executed at least once and such that no tokens remain on the arcs of the function graph. The behavior possible in both initial graphs is shown in boldface while the new behavior is shown using non-boldface characters.

3.3 From function graphs to EPCs

A function graph can be transformed back into an EPC. For this, we first generate an EPC where each non-initial function is preceded by a dedicated join connector and each non-final function is succeeded by a dedicated split connector (see Figure 4b). Arcs connect these connectors in line with the arcs connecting the functions of the function graph via an event for each such arc. Start events are directly connected to initial functions while final functions are directly connected to end events. The connector types are calculated for each connector based on the values of all arcs leaving or ending in the corresponding function in the function graph. If all arcs leaving a function in the function graph are of split type XOR, the split connector of this function in the EPC becomes an XOR connector (as for function B in Figure 4). If all arcs are of split type $\wedge$, it becomes an $\wedge$ connector (as for A in Figure 4). Otherwise it becomes an $\vee$ connector. In the same way the join connector before each function in the EPC becomes an XOR connector if all arcs pointing at this function in the function graph are of join type XOR, it becomes an $\wedge$ connector if the arcs are of join type $\wedge$, and it becomes an $\vee$ connector otherwise (as for E in Figure 4).

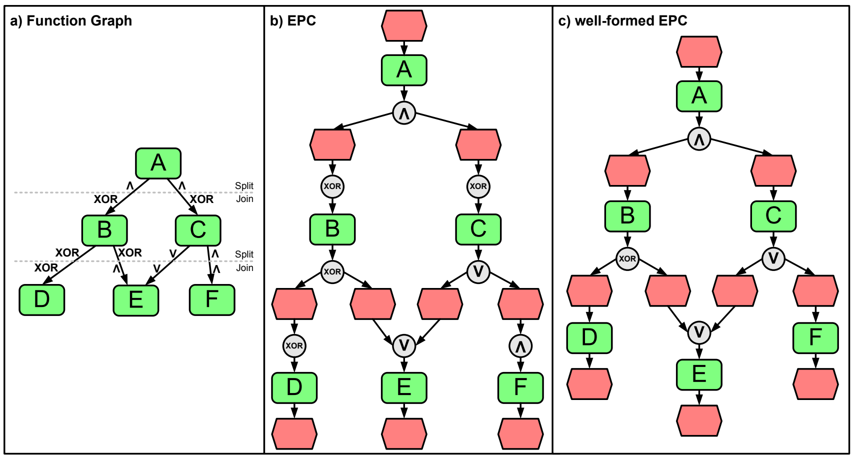


Fig. 4. Transforming a function graph back into an EPC

Definition 8 (Generating an EPC from a function graph). *A function graph* (F, A^F, l_J^F, l_S^F) *can be converted into an EPC* $(E, F, C_J^N \cup C_S^N, l, A)$ *as follows:*

- *For all* $f \in F$*:* $\bullet f = \{x|(x,f) \in A^F\}$*, and* $f\bullet = \{x|(f,x) \in A^F\}$*.*
- $C_J^N = \{c_J^f | f \in F \wedge |\bullet f| \geq 1\}$ *assigns a join connector to each function,*
- $C_S^N = \{c_S^f | f \in F \wedge |f\bullet| \geq 1\}\}$ *assigns a split connector to each function,*
- $E = \{e^{(f_1,f_2)} | (f_1,f_2) \in A^F\} \cup \{e_{start}^f | f \in F \wedge \bullet f = \emptyset\} \cup \{e_{end}^f | f \in F \wedge f\bullet = \emptyset\}$ *is the set of events,*
- $A = \{(f, c_S^f) | f \in F \wedge c_S^f \in C_S^N\} \cup \{(c_J^f, f) | f \in F \wedge c_J^f \in C_J^N\}$
  $\cup \{(c_S^{f_1}, e^{(f_1,f_2)}) | (f_1,f_2) \in A^F\} \cup \{(e^{(f_1,f_2)}, c_J^{f_2}) | (f_1,f_2) \in A^F\}$
  $\cup \{(e_{start}^f, f) | f \in F \wedge \bullet f = \emptyset\} \cup \{(f, e_{end}^f) | f \in F \wedge f\bullet = \emptyset\}$,
- *for all* $c_J^f \in C_J^N$*:* $l(c_J^f) = \begin{cases} XOR \text{ if } \forall_{x \in F}(x,f) \in A^F \Rightarrow (l_J^F((x,f)) = XOR) \\ \wedge \quad \text{ if } \forall_{x \in F}(x,f) \in A^F \Rightarrow (l_J^F((x,f)) = \wedge \\ \vee \quad \text{ otherwise,} \end{cases}$
- *for all* $c_S^f \in C_S^N$*:* $l(c_S^f) = \begin{cases} XOR \text{ if } \forall_{x \in F}(f,x) \in A^F \Rightarrow (l_S^F((f,x)) = XOR \\ \wedge \quad \text{ if } \forall_{x \in F}(f,x) \in A^F \Rightarrow (l_S^F((f,x)) = \wedge \\ \vee \quad \text{ otherwise.} \end{cases}$

The EPC generated in this way might not be well-formed because it can violate the Requirement 7 of Definition 3. It may contain connectors which have only one incoming and at the same time only one outgoing arc (e.g., see functions B, C, D, and F in Figure 4b). However, such connectors can simply be eliminated from the net by replacing each of these connectors with a direct arc from its predecessor node to its successor node (see Figure 4c).

Definition 9 (Generating a well-formed EPC from a function graph). *If an EPC* $(E^\diamond, F^\diamond, C^\diamond, l^\diamond, A^\diamond)$ *was derived from a function graph, it can be converted into a well-formed EPC* (E, F, C, l, A) *as follows:*

- $E = E^\diamond$,
- $F = F^\diamond$,
- *For all* $n \in (E^\diamond \cup F^\diamond \cup C^\diamond)$*:* $\bullet n = \{m | (m,n) \in A^\diamond\}$ *is the set of input nodes, and* $n\bullet = \{m | (n,m) \in A^\diamond\}$ *is the set of output nodes,*
- $C = \{c \in C^\diamond | \, |\bullet c| \geq 2 \vee |c\bullet| \geq 2\}$,
- $l \in C \rightarrow \{\wedge, \vee, XOR\}$ *such that* $l(c) = l^\diamond(c)$ *for all* $c \in C$,
- $N' = E^\diamond \cup F^\diamond \cup C$,
- $A = (A^\diamond \cap (N' \times N')) \cup \{(x,y) \in N' \times N' | \exists_{z_1,\ldots,z_n \in (C^\diamond \setminus C)} \langle x, z_1, \ldots, z_n, y\rangle \in (A^\diamond)^*\}$.

When transforming a function graph into an EPC, the connector values determining the possible behavior of the resulting EPC are calculated from the split and join type values of the function graph's arcs. Only if each arc leaving a function is assigned the split type *XOR*, then also the corresponding EPC connector determining the successors of the function in the EPC is of type *XOR* allowing for an exclusive choice of one of the succeeding functions. This corresponds exactly to the behavior possible in the function graph. If all these arcs are of type $\wedge$, also the corresponding connector in the EPC becomes an $\wedge$ connector. Again, the behavior of triggering all succeeding paths is in line with the behavior of the function graph. If the arcs are of type $\vee$ or if there is a mixture of different split types among the arcs leaving a function of a function graph, then the corresponding EPC connector is assigned type $\vee$. The resulting behavior then corresponds to exactly the behavior if all the corresponding arcs of the function graph are of type $\vee$. Each other combination of arc types in the function graph allows for a subset of this behavior. Thus, by assigning the EPC connector the type $\vee$, additional behavior might become possible in the EPC compared to the behavior allowed in the function graph.

The argumentation for the behavior allowed by join connectors in the resulting EPCs is in line with this behavior of the split connectors.

All in all, in each of the merge algorithm's three steps the possible behavior of the input model(s) is at least preserved in the resulting model, but usually even extended with additional behavior. As explained in the introduction, we aim at "simple" models that preserve at least the original behavior over complex models that match exactly the behavior of the input models. Thus, such an over-approximation is a desired outcome. For that reason, we also rather stick to the "simple" EPC derived in the last transformation here although introducing additional connectors to create an EPC that preserves the behavior of the function graph more exactly would well be possible.

4 Tool support

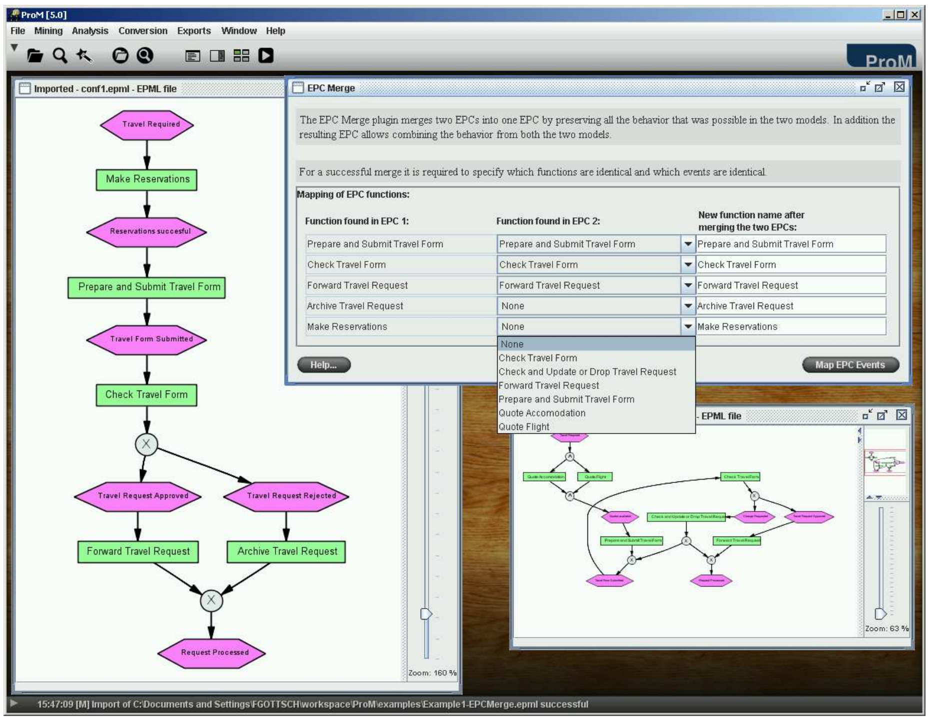
We have implemented the approach described in this paper as a plug-in of the ProM process mining framework[2] (see Figure 5 for a screenshot of ProM). The ProM framework integrates and enables the combined use of a wide variety of process mining techniques through more than 250 plug-ins [2]. By implementing the merge of EPCs as a ProM plug-in, it can be used and combined with existing mining plug-ins to create integrated process models from log files and other models.

Before performing the actual merge of two selected EPCs as described in this paper, the ProM plug-in allows users to create a mapping between the functions of the two input EPCs (see the top-right window in Figure 5) as well as between their events. In this way, it can be avoided that different names for the same functions or events (as e.g. defined through ontology classes or caused by typos etc.) cause superfluous additional elements in the resulting model. To help the user with this mapping, the plug-in automatically suggests possible corresponding elements using a library provided by ProM for matching identifiers.[3]

After mapping the functions and events, the algorithm merges the two EPCs in three phases as explained in this paper. As the behavior of a process model

[2] See `http://prom.sourceforge.net/`.

[3] An illustrated description of how to use the EPC merge plug-in can be found at `http://www.floriangottschalk.de/epcmerge`.



The EPC Merge plugin merges two EPCs into one EPC by preserving all the behavior that was possible in the two models. In addition the resulting EPC allows combining the behavior from both the two models.

For a successful merge it is required to specify which functions are identical and which events are identical.

Mapping of EPC functions:

Function found in EPC 1:	Function found in EPC 2:	New function name after merging the two EPCs:
Prepare and Submit Travel Form	Prepare and Submit Travel Form	Prepare and Submit Travel Form
Check Travel Form	Check Travel Form	Check Travel Form
Forward Travel Request	Forward Travel Request	Forward Travel Request
Archive Travel Request	None	Archive Travel Request
Make Reservations	None	Make Reservations

None
Check Travel Form
Check and Update or Drop Travel Request
Forward Travel Request
Prepare and Submit Travel Form
Quote Accomodation
Quote Flight

15:47:09 [M] Import of C:\Documents and Settings\FGOTTSCH\workspace\ProM\examples\Example1-EPCMerge.epml successful

Fig. 5. Merging the EPC from Figure 1 with the EPC depicted in the left using the EPC merge's implementation in ProM

is determined by the order of executing its functions, the names of the events between the functions are irrelevant from a behavioral point of view and thus ignored in the algorithm presented in this paper. The algorithm simply creates new events when transforming the function graph back into an EPC. However, for the depiction and understanding of the process, the event names are of course important. Thus, after merging the two models, the plug-in re-names each event of the resulting model based on the event names of the corresponding event(s) in the original models. Considering that the plug-in introduces unique initial functions before each start event in the original model, and unique final functions after each end event of the original models, an event is always located on a path between two functions. Thus, each event in the resulting model can be named according to the event that is on the same path in the original models. Whenever the events of the original EPCs differ, a choice between these differing events is added on the path instead of the single event.

To illustrate the functionality of the ProM plug-in, consider the left window in Figure 5 which shows another EPC for a travel approval process. This model is merged with the process from Figure 1 (which is also shown in small on the lower right of Figure 5) to illustrate our approach. In this process, reservations are made directly before the travel form is filled in and submitted. The travel is

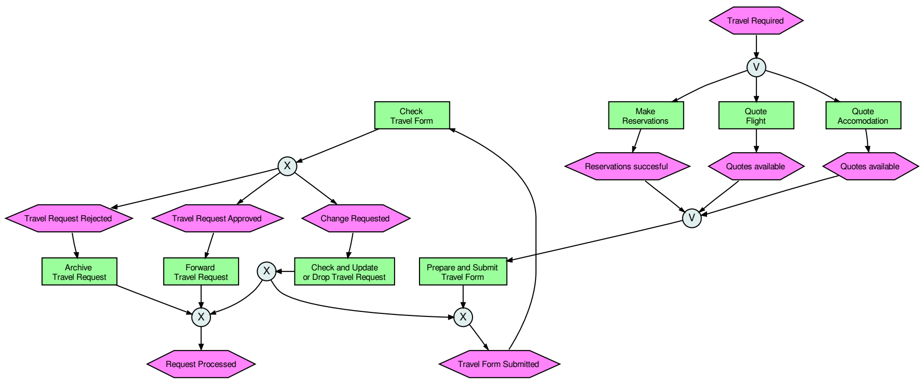


Fig. 6. The EPC resulting from merging the EPCs from figures 1 and 5

afterwards either approved and the form forwarded to the clearing center or rejected and the form is archived. While the functions "Prepare and Submit Travel Form", "Check Travel Form", and "Forward Travel Request" can be mapped to equivalent functions in the EPC from Figure 1, the functions "Archive Travel Request" and "Make reservations" cannot be mapped to any corresponding function (upper-right of Figure 5).

The EPC resulting from this merge is shown in Figure 6. In the integrated model, an $\vee$ connector allows in the beginning for both the request of quotes for accommodation and flights as in the model from Figure 1 as well as for making directly a reservation as in Figure 5. Due to the use of the $\vee$ connector even solely quoting a flight or combining such a quote with a reservation of an accommodation would in contrast to the original models be possible according to this new model. While in each of the original models the check of the travel form could result in an acceptance of the travel request, the alternative to this was a rejection in one of the models, while it was the request for a change in the other model. Thus, the new merged model in Figure 6 allows for a choice of one of these three options.

5 Related work

Process models like EPCs are used to provide abstract views on the process behavior of complex systems. As various models are usually inconsistent [10] even if they are intended to depict the same behavior, several frameworks to merge such models and align the depicted system descriptions are suggested in literature [4, 10, 11]. A concrete, but not implemented algorithm for merging EPCs is suggested in [8]. While our goal is to merge several EPCs depicting different processes that are executed similarly, the goal of [8] is to integrate different views on the same process. For that reason the algorithm of [8] tries to synchronize different behavior while our approach provides a choice between the varying behavior.

Besides as a model, behavior can also be represented as a state-transition system which explicitly shows each state the whole system can be in and how it can change between these states. The synthesis of such state-based models into Petri nets by using minimal regions is, e.g, presented in [6] and implemented in tools such as *Petrify* [5]. Also ProM provides several region-based approaches [2]. The merge of transition systems and properties of such a merge of behavior are discussed in [4, 14].

Furthermore, the behavior of a system can be represented as traces which consist of a log entry for every action that has happened. Various process mining approaches have been suggested to generate process models from such a set of behavioral traces (e.g. [1, 3, 7]). While originally intended to discover the behavior of one system, these techniques can also discover a process model that covers the behavior of several systems if they are provided with a combined set of traces from all the relevant systems.

6 Conclusions

In this paper we have shown an approach to merge two EPCs into a single EPC that preserves all the behavior possible in the original EPCs, but that may also allow for additional behavior. For this, we reduce EPCs to function graphs that represent an abstraction of the behavior of the EPCs and unify such function graphs. In this way the only expensive calculation during the merge is re-calculating the connector values between the functions which is only required locally. Thus, an efficient implementation is possible.

By implementing the algorithm within the ProM process mining framework, it can now be used in combination with a wide range of other techniques helping users in constructing process models for existing systems. While in some cases, the additional behavior compared to the original models which is made possible by this approach might be undesired, usually the goal of process analysts when merging process models is to quickly align various depictions of the same or similar processes. The resulting model is then just the starting point for further process optimizations. Thus, a simple model which guarantees the behavior of the original models, is usually more desired in such cases than a complex model which tries to match the behavior exactly. For this reason, we also refrained from encoding behavioral information available during the merge more exactly in longer chains of EPC connectors.

While we depicted this approach using EPCs, the same approach should be applicable to other business process modelling languages that support $\vee$ splits and joins of the control flow. In our approach we focused on preserving the behavior that was possible in the original EPCs, but neglected if this behavior or the behavior that we created additionally during the merge is desirable and sound behavior. In future research we thus have to show how the merged models should be post-processed by process analysts in order to derive optimal and executable business processes for merging corporations.

References

1. W.M.P. van der Aalst, A.K. Alves de Medeiros, and A.J.M.M. Weijters. Genetic Process Mining. In *26th International Conference on Applications and Theory of Petri Nets (ICATPN 2005)*, volume 3536 of *LNCS*, pages 48–69. Springer, 2005.
2. W.M.P. van der Aalst, B. van Dongen, C. Günther, R. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H.M.W. Verbeek, and A.J.M.M. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. In *Petri Nets and Other Models of Concurrency ICATPN 2007*, volume 4546 of *LNCS*, pages 484–494. Springer, 2007.
3. W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
4. G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A Manifesto for Model Merging. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 5–12, New York, NY, USA, 2006. ACM.
5. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, 1997.
6. J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing Petri Nets from State-Based Models. In *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 164–171, Washington, DC, USA, 1995. IEEE Computer Society.
7. B. van Dongen and W.M.P. van der Aalst. Multi-Phase Mining: Aggregating Instances Graphs into EPCs and Petri Nets. In *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 35–58. Florida International University, Miami, FL, USA, 2005.
8. J. Mendling and C. Simon. Business Process Design by View Integration. In *Business Process Management Workshops*, volume 4103 of *LNCS*, pages 55–64. Springer, 2006.
9. M. Rosemann and W.M.P. van der Aalst. A Configurable Reference Modelling Language. *Information Systems*, 32(1):1–23, March 2007.
10. M. Sabetzadeh and S. Easterbrook. View Merging in the Presence of Incompleteness and Inconsistency. *Requirements Engineering*, 11(3):174–193, 2006.
11. M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik. A Relationship-Driven Framework for Model Merging. In *MISE '07: Proceedings of the International Workshop on Modeling in Software Engineering*, pages 2–8, Washington, DC, USA, 2007. IEEE Computer Society.
12. K. Sarshar and P. Loos. Comparing the Control-Flow of EPC and Petri Net from the End-User Perspective. In *3rd International Conference on Business Process Management (BPM 2005)*, volume 3649 of *LNCS*, pages 434–439, Nancy, France, September 2005. Springer.
13. A.-W. Scheer. ARIS Toolset: A Software Product is Born. *Information Systems*, 19(8):607–624, December 1994.
14. S. Uchitel and M. Chechik. Merging Partial Behavioural Models. *SIGSOFT Software Engineering Notes*, 29(6):43–52, 2004.