

Structural Aspects of Business Process Diagram Abstraction

Sergey Smirnov
Business Process Technology Group
Hasso Plattner Institute at the University of Potsdam
D-14482 Potsdam, Germany
sergey.smirnov@hpi.uni-potsdam.de

Abstract—As companies more and more often turn to documenting their business processes in models, the task of managing large model collections becomes essential. There is a number of techniques simplifying this task, e.g., construction of customized process views and business process model abstraction. The latter aims at deriving abstract process representations from existing low-level models omitting details irrelevant for the current task. A number of papers on process model abstraction conceptualized the abstraction problem and proposed algorithms handling simplistic models. To the best of our knowledge there is no work discussing abstraction of models in BPMN. In this paper we present an abstraction approach, addressing specific features of BPMN 1.2. The abstraction approach is order-preserving and is capable of handling graph-structured process models.

Keywords—business process modeling, business process model abstraction, decomposition of a BPD, BPMN 1.2.

I. INTRODUCTION

Each process model captures aspects of a real business process relevant to the task at hand. Thus, it is no wonder that one process may have several models differing in the level of details or capturing the process from the perspectives of various stakeholders [1]. A demand to have multiple descriptions of a business process corresponding to different levels of precision is a common place [2]. Maintaining these numerous models in sync is laborious and error-prone. However, if a company possesses detailed models, high level ones can be derived from existing by means of abstraction—generalization of models reducing insignificant details and retaining information relevant for a particular problem. This task is in the focus of business process model abstraction (BPMA). Therefore, BPMA eliminates the pricey and error-prone maintenance problem, providing the flexibility to create a model with the desired abstraction level on demand.

To be useful, abstraction must deliver a model that preserves the general process logic. This implies that 1) business semantics of a coarse-grained element in a resulting model embodies the business semantics of several logically related objects of a low-level model, and 2) overall process structure is preserved. The first point requires the abstraction to deliver objects meaningful from a business point of view. Let us look at a process where a customer buys goods paying with a money transfer: (s)he signs into the online banking account, provides the transfer information, and confirms the

transfer. A sequence of the three latter activities semantically corresponds to a more coarse-grained activity “pay by transfer”. Albeit business semantics has great importance for abstraction, we do not address it in this paper: we neither consider the business meaning of model objects being abstracted, nor the semantics of the result. Rather, the focus is on the preservation of the overall structure, since business semantics can not be approached, while structural aspects are not studied. We pay attention to the process structure and semantics of BPMN language elements (do not mix it with business semantics). The approach seeks for process model fragments which are self-contained from a structural point of view. Meanwhile, abstraction of a signal event and a timer event makes the difference.

In [3], [2], [4], [5] we have already motivated BPMA and discussed its various aspects, from conceptual ideas to concrete algorithms. The proposed algorithms evolved from the simplest one in [2] to more sophisticated in [5] and, finally to [4]. However, these approaches address abstraction of models in rather simple notations. While [2] considers Event-driven Process Chains (EPC) [6], the other two study a notation distinguishing activities, gateways, and control flow only. Business Process Modeling Notation (BPMN) [7], which has become a de facto standard in process modeling, provides far more objects. Adjusted to the existing methods abstraction of Business Process Diagrams (BPD) might seem trivial. The results of this work aim to unveil this myth: the rich expressiveness of BPMN results in challenges not addressed before.

The goal of this paper is to describe how process model abstraction can be realized for BPMN models. The main challenge is numerous element types distinguished by BPMN specification. This contrasts to the decomposition and abstraction techniques, which assume rather simplistic structure of a process model. In this paper we bridge the existing gap and describe how the available techniques can be employed for abstraction of BPMN models. Without loss of generality we rest upon decomposition of a process model into canonical single entry single exit (SESE) fragments [8].

In Section II we elaborate on the term business process model abstraction and explain the basics of the approach. Section III introduces an auxiliary notation and a method for abstracting models in this notation. The auxiliary notation

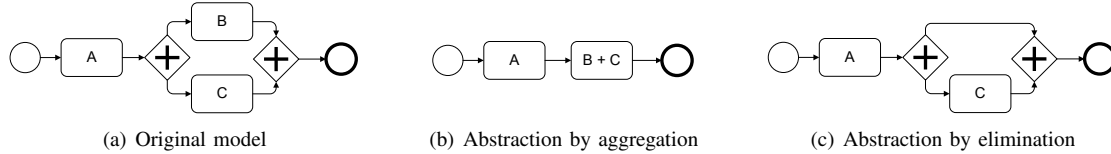


Figure 1. Examples of aggregation and elimination

helps to cope with the complexity of BPMN. Section IV continues this discussion: it shows how abstraction of BPMN can be systematically approached and how existing abstraction techniques can be used. Proposing concrete methods for abstraction it also discusses alternatives. In Section V we review related work. Section VI concludes the paper.

II. ABSTRACTION OF BUSINESS PROCESS MODELS

The term *abstraction* has various connotations in different domains. In software engineering abstraction is tightly coupled with the term *model*, as any model is an abstraction of a real object. Classes in object-oriented approach (see [9]) and process models in business process modeling (see [10]) are examples of models. To avoid any ambiguity we provide the following definition of abstraction.

Definition 1: Business process model abstraction is generalization of a process model preserving the overall process logic and leaving out insignificant process details in order to retain information relevant for a particular purpose.

Process model abstraction can be seen as the transformation of a detailed process model aiming at reduction of insignificant details. Hence, information loss is the desirable outcome of abstraction. Abstraction is driven by a purpose and, as such, one can require various “details” to be reduced: activities, events, or whole process model execution paths. In [3] we identified several use cases for process model abstraction. For each use case it was shown which objects of the model are subject for abstraction. When business users talk about process model abstraction, they usually think of activity generalization. Their primary concern is about putting elementary steps together to design a model with more coarse-grained activities. Several research projects (see [11], [12]), as well as our experience [3], [4], prove that activities are often in the focus of abstraction. Thus, we use activities as the abstraction subject. Meanwhile, even if activities are being abstracted, other model objects are inevitably involved in the abstraction.

We realize process model abstraction as a series of transformations. Since abstraction leaves out model details and does not enrich the model with new information, we implement abstraction via two basic operations: *aggregation* and *elimination*. Aggregation replaces several model objects with one aggregating object. The properties of an aggregating object are derived from the properties of aggregated objects. Therefore, aggregation preserves information in the model. Elimination simply omits objects without leaving

information about them. Figure 1 compares the effect of the two operations, showing the original process model in Figure 1(a) and abstractions of activity B using aggregation (Figure 1(b)) and elimination (Figure 1(c)). While aggregating activity $B + C$ in Figure 1(b) references activities B and C in the label, the model in Figure 1(c) does not mention B at all. A more valuable feature of aggregation is derivation of non-functional properties of an aggregating activity from the properties of aggregated objects (e.g., activity execution time or cost). Elimination does not allow maintaining non-functional properties.

The abstraction approach rests on the assumption that activities to be abstracted are known before abstraction starts. Every activity is abstracted independently from others. Model transformation evolves as a series of *abstraction steps*. In every abstraction step one activity from the set is processed. The output of the current abstraction step is a new process model, which is the input for the next step. Model transformation continues until every insignificant activity is handled.

We require the abstraction to be order-preserving: not introducing new ordering constraints and omitting those only between abstracted elements. Assume that in the current abstraction step activity A should be concealed. We employ the notion of a process model fragment—the connected subgraph of a graph representing a process model. Let the abstraction step affects model fragment f_A , containing activity A . The abstraction replaces fragment f_A with activity F . If activity B belongs to f_A , information about the ordering constraints between A and B is lost. However, the order-preserving abstraction must guarantee that for any pair of activities that do not belong to f_A , e.g., activities C and D , the ordering constraints between them are preserved. In addition, the order-preserving abstraction must assure that the ordering constraints between any activity not in f_A , e.g., activity E , and any activity in f_A , activities A or B in our example, are the same as between activities E and F .

The discussed abstraction approach purely relies on the process structure. Therefore, the abstraction may transform unsound process models into sound. For instance, this can happen if a process fragment with a deadlock is abstracted. To avoid such situations we assume the original process models to be sound. At the same time if the original model is sound we aim at deriving the sound model.

III. ABSTRACTION OF AUXILIARY MODELS

BPMN expressiveness is high due to a large number of language elements and complex relations between them. BPMN 1.2 provides 6 types of activities and 6 types of gateways; the notation distinguishes associations, message and sequence flow, whilst the sequence flow is subdivided into normal and exception flows. Abstraction of BPDs has to take into account this rich semantics.

In [5] we have shown how process model abstraction is realized for a modeling notation distinguishing activities, gateways, and control flow edges. Further we reference this notation as *auxiliary process modeling notation* and models in this notation as *auxiliary models*. As the approach considers abstraction of activities, the auxiliary notation gives enough information to fulfill the task. We propose to use the auxiliary notation for BPD abstraction. Obviously, a significant gap exists between the expressiveness of BPMN and the expressiveness of auxiliary notation. Thus, we seek a method to reduce the complexity of a BPD to the level of auxiliary notation and then perform the abstraction. First, let us formalize the notion of a process model in auxiliary notation.

Definition 2: $(N, E, type)$ is an *auxiliary business process model*, where:

- $N = N_A \cup N_G$ is a set of nodes. $N_A \neq \emptyset$ is a set of activities and N_G is a set of gateways; the sets are disjoint.
- $E \subseteq N \times N$ is a set of directed edges between nodes representing control flow.
- (N, E) is a connected graph.
- Every activity has at most one incoming and at most one outgoing edge.
- There is at least one activity with no incoming edges—a start activity, and at least one activity with no outgoing edges—an end activity.
- $type : N_G \rightarrow \{and, xor, or\}$ is a function that assigns to each gateway a control flow construct.
- Every gateway is either a split or a join; splits have exactly one incoming edge and at least two outgoing; joins have at least two incoming edges and exactly one outgoing.

Abstraction algorithm for auxiliary models rests upon a well-established technique decomposing a graph into *canonical SESE fragments*. A SESE fragment is a special kind of fragment, which has exactly one incoming and one outgoing edge. The node sets of two canonical SESE fragments are either disjoint or one contains the other. Let f_1 and f_2 be canonical SESE fragments. They might be in one of the two relations: *parent-child* or *predecessor-successor*. If the node set of f_1 is the subset of f_2 node set, then f_1 is the *child* of f_2 (f_2 is the *parent* of f_1). If f_1 is the child of f_2 and there is no f_3 , such that f_3 is the child of f_2 and f_3 is the parent of f_1 , then f_1 is the *direct child* of f_2 . The parent-

child relation defines a hierarchy of SESE fragments, which is unique. If the outgoing edge of f_1 is the incoming edge of f_2 , then f_1 directly precedes f_2 (and f_2 directly succeeds f_1).

Abstraction of one activity, let it be a , starts with seeking for the SESE fragment directly containing a (denoted with se_{se_a}). Having se_{se_a} at hand it is possible to derive $se_{se_{min}}$ —the minimal SESE fragment which contains at least two activities, one of which is a . If there exists a canonical SESE fragment $se_{se_{a'}}$, which is in the predecessor-successor relation with se_{se_a} , then $se_{se_{min}}$ is a SESE fragment with the incoming edge of the predecessor and the outgoing edge of the successor in the pair $(se_{se_a}, se_{se_{a'}})$. If such canonical SESE fragment does not exist, then $se_{se_{min}}$ is a SESE fragment which is the parent of se_{se_a} . Fragment $se_{se_{min}}$ is aggregated into one activity, whilst the incoming and outgoing edges of $se_{se_{min}}$ become the incoming and outgoing edges of this activity, respectively. The presented abstraction method is order-preserving as argued in [5]. Notice that for BPD abstraction it is enough to know the edges defining fragment $se_{se_{min}}$.

IV. BPD ABSTRACTION MECHANISM

While auxiliary model abstraction is primarily concerned with handling of control flow, abstraction of activities in BPMN should address additional model aspects. In this section we present the abstraction algorithm for BPDs, starting with the algorithm overview and looking into the details of each algorithm step afterwards.

A. Managing Problem Complexity

Perception of BPMN connecting objects plays the key role in the abstraction algorithm. BPMN 1.2 distinguishes associations, message flow, and sequence flow, subdivided into normal flow and exception flow. The auxiliary notation makes use of control flow only. Further we motivate and explain the relations between connecting objects of BPMN and control flow of auxiliary model.

Sequence flow explicitly defines the execution order, while message flow describes message exchange, indirectly influencing the execution order. Following this differentiation we treat the sequence flow as the control flow (capturing it as edges in the auxiliary notation), but do not treat the message flow as such (not capturing it in the auxiliary notation at all). Due to this, processes in different pools are handled independently.

Sequence flow is subdivided into normal flow and exception flow. The normal flow prescribes “normal” evolution of a process, while exception flow shows exception handling. To capture exceptional situations BPMN uses attached intermediate events: a sequence flow leading from such an event corresponds to exception handling. Such a flow can merge into the normal flow of a BPD (e.g., see exception flow leading from attached intermediate event of

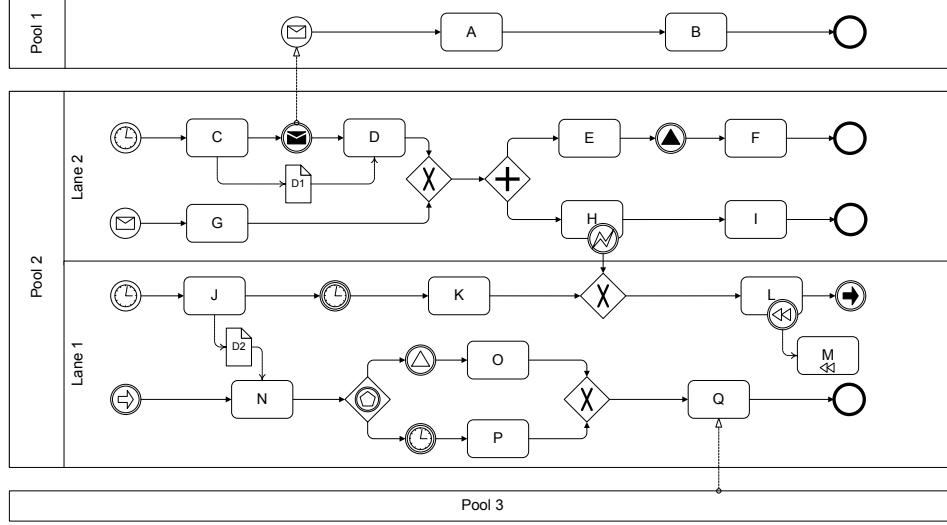


Figure 2. Example process model

activity *H* in Figure 2). From this point of view there is no fundamental difference between the two flow types and abstraction mechanism treats normal flow and exception flow as equal.

Associations are not treated as control flow, as they do not directly define execution constraints. If an association is used to show activity compensation, an attached intermediate compensation event is attached to a compensated activity, and the directed association leads from the event to a compensation activity (see activity *M* in Figure 2). However, BPMN 1.2 allows exactly one activity to be used for the compensation and it can not “merge” into sequence flow.

The described perception of BPMN connecting objects allows finding model parts which are handled independently. A sequence flow may be enclosed into a pool or an expanded subprocess. If a pool contains several processes not connected with each other by sequence flow, abstraction of an activity in one of them has no impact on others.

The choice of an appropriate method for activity abstraction depends on the activity allocation type. We distinguish three allocation types: I) allocation within a sequence flow, i.e., an activity is connected by a sequence flow with other elements, II) activity is a compensation activity, and III) allocation within an ad-hoc subprocess. Abstraction of activities in the sequence flow is based on the abstraction of auxiliary model. As we argued, compensation activities are not related to the sequence flow and constitute a separate allocation type. An ad-hoc subprocess does not provide graphical information about the execution order of the contained objects. An activity in such a subprocess neither belongs to a sequence flow, nor is used as a compensation activity. We put such activities into a separate allocation type.

When an activity is referred to an allocation type, the

Algorithm 1 Abstraction of an activity in a BPD

- 1: **abstractBPD**(BPD *diagram*, **Activity** *activity*)
 - 2: **if** *activity* is compensation activity **then**
 - 3: eliminate *activity*, corresponding compensation event, and association
 - 4: **return**
 - 5: **if** *activity* in *adHocSubprocess* **then**
 - 6: collapse *adHocSubprocess*
 - 7: **return**
 - 8: find sequence flow *flow* where *activity* is allocated
 - 9: construct auxiliary model *auxiliaryModel* for *flow*
 - 10: perform abstraction in *auxiliaryModel*
 - 11: make necessary updates in *diagram*
-

corresponding abstraction method is applied. Abstraction of an activity in a sequence flow implies identification of a sequence flow containing the activity and finding a SESE fragment aggregating it. To find the fragment we construct an auxiliary model, corresponding to the sequence flow and turn to abstraction algorithm described in Section III. The details of auxiliary model construction are provided in the next subsection. The flow objects within the SESE fragment are aggregated into one activity, while related non-flow and connecting objects might require an update. Examples of such objects are data objects, associations and message flow. The BPD update procedure is discussed in subsection IV-C.

Abstraction of activities with allocation types II and III is straightforward and is realized directly in a BPD. As exactly one activity is used for compensation, its abstraction is trivial: the activity together with the corresponding event and association between them are eliminated. Abstraction of an activity within an ad-hoc subprocess is achieved through collapsing the subprocess.

Algorithm 1 formalizes the discussed procedure. It describes one abstraction step, i.e., abstraction of one activity in a BPD. The inputs of the algorithm are *activity*—an activity to be abstracted and *diagram*—the diagram containing the activity. First, the algorithm seeks for the type of activity allocation. If the activity is used for compensation, it is eliminated and the abstraction completes (lines 2 – 4). If the activity is allocated within an ad-hoc subprocess, the subprocess is collapsed and abstraction completes (lines 5 – 7). These two allocation types are checked first, since their processing requires the least effort. If abstraction has not been completed, sequence flows are analyzed. The activity is localized within a sequence flow, for which the auxiliary model is constructed and abstracted. Finally, the BPD is updated. Algorithm 1 is very high level. In the remainder of this section we present every step in detail.

B. Auxiliary Model Construction

Construction of an auxiliary model has three phases. In the first phase normalization of a sequence flow is performed. Normalization aims at bringing a sequence flow to the state when mapping into auxiliary model becomes feasible. Normalization includes five rules:

Multiple incoming sequence flows if an activity has more than one incoming sequence flow, a XOR join merging these flows and leading to the activity is added.

Multiple outgoing sequence flows if an activity has more than one outgoing sequence flow, an AND split following the activity and parallelizing outgoing flows is added.

Attached intermediate events an activity with an attached intermediate event is replaced by the XOR split succeeded by the activity. While one outgoing flow of the XOR leads to the activity, the other corresponds to the outgoing flow of the attached intermediate event in the initial model.

Multiple start events if a process model has multiple start events, the model is complemented by the necessary number of XOR split gateways and edges, assuring that the process model has exactly one start node.

Multiple end events if a process model has multiple end events, the model is complemented by the necessary number of XOR join gateways and edges, assuring that the process model has exactly one end node.

The last two rules can be implemented using the approaches discussed in [13] or [14]. These works describe the approaches assuring that a process model has exactly one start and one end event.

The goal of the second phase is to obtain sequence flows free of events. We call such flows preliminary. Each preliminary flow is the result of a normalized sequence flow traversal. For every normalized sequence flow the traversal begins with a start event and considers only sequence flow objects. Such objects as data objects, text annotations, and attached intermediate events are ignored and do not appear

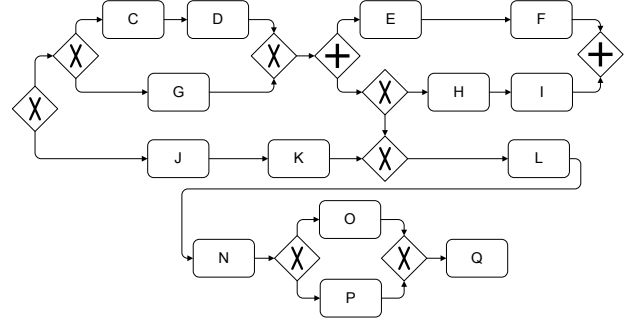


Figure 3. An auxiliary model for the sequence flow in “Pool 2” in Figure 2

in the preliminary flow. A pair of throwing and catching link events is used as off-page connectors and equals to a sequence flow. If during a sequence flow traversal a throwing link event is come across, it is not included in the flow. Instead, the pair link catching event is sought and only its successor is included in the process model.

Finally, a model in auxiliary notation is built from a preliminary flow according to the following three rules:

Rule 1 Every activity in the preliminary flow is mapped into an activity in the auxiliary notation.

Rule 2 Every gateway in the preliminary flow is mapped into a gateway in the auxiliary notation. A data-based gateway in a BPD is mapped to a gateway of the corresponding type in the auxiliary notation. For instance, a data-based XOR gateway is mapped to a XOR gateway in the auxiliary notation. An event-based gateway is mapped to a XOR gateway, while a complex gateway—to an OR gateway.

Rule 3 For every activity/gateway in the preliminary flow the succeeding activity/gateway is found. In the auxiliary model the pair of corresponding nodes is connected with an edge capturing the execution order of the preliminary flow.

Figure 3 presents an auxiliary model corresponding to the sequence flow in pool “Pool 2” of the example process model. After the auxiliary model is built and abstracted, the changes have to be propagated back to the BPD. Functions ref_{in} and ref_{out} enable this, setting up correspondences between nodes of auxiliary model and edges of preliminary flow. As such, it is possible to learn which fragment of a BPD has to be updated.

Definition 3: Function $ref_{in} : N \rightarrow \mathcal{P}(A)$, where A is the set of edges in a BPD and N is the set of nodes in the auxiliary model derived from this BPD, assigns to each node in an auxiliary model a set of edges in a BPD. Given node $n' \in N$ in the auxiliary model, a BPD edge belongs to the result of function ref_{in} if 1) the BPD edge is the incoming edge of n and 2) BPD node n results in construction of node n' . Function $ref_{out} : N \rightarrow \mathcal{P}(A)$ assigns to each node in an auxiliary model a set of edges in a BPD. Given node $n' \in N$ in the auxiliary model, a BPD edge belongs to the result of function ref_{out} if 1) the BPD edge is the outgoing edge of

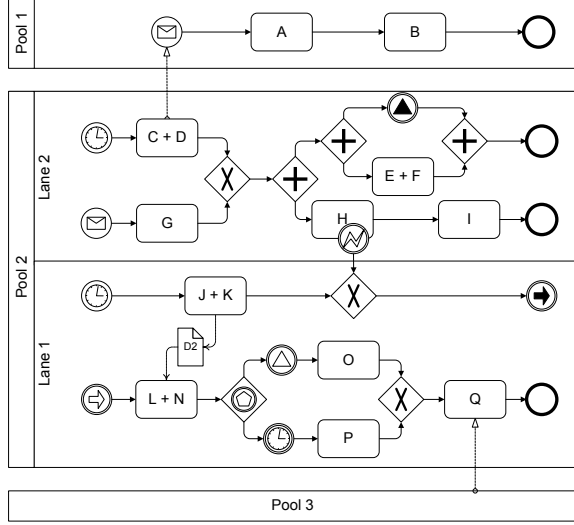


Figure 4. Abstraction of example process model in Figure 2

n and 2) BPD node n results in construction of node n' .

C. BPD Update

To complete the abstraction, the necessary updates have to be done in the BPD. We illustrate these updates by the process model in Figure 4, which originates from abstraction of activities D , E , J , and N in the example model. By means of functions ref_{in} and ref_{out} we obtain the edges bounding a SESE fragment in the BPD. This fragment is aggregated into one activity. The effects of BPD preprocessing has to be eliminated through application of rules inverse to preprocessing rules. Finally, certain types of aggregated flow objects and non-flow objects require extra processing.

Events which appear in the sequence flow are concealed within the fragment along with other flow objects. There is a number of exceptions from this rule. If an abstracted fragment contains several start events, at least one should be preserved. To show that several start events are concealed, we prescribe to use a *multiple start event* proposed by BPMN specification. A similar approach can be used for aggregation of several end events. Aggregation of intermediate message events (or activities with a message flow) leads to assigning the corresponding message flow to the aggregating activity. The direction of the message flow is preserved. For instance, abstraction of activity D leads to aggregating it with C and concealing the message event. Hence, the outgoing message flow is assigned to the aggregating activity.

The scope of a signal event may include several pools and span the whole process model. Hence, a signal event is safely aggregated within an activity only if there is no pair signal event outside the aggregated fragment. Otherwise, the signal event should be preserved in the model. Since the ordering constraints within the aggregated fragment are lost, we propose to allocate the signal event on the branch parallel

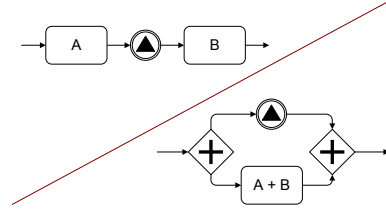


Figure 5. Signal event handling

to an aggregating activity (see Figure 5). If there are several signal events, each of them is allocated on the dedicated branch in an AND block. Figure 4 demonstrates the effect of aggregating activities E and F with the throwing signal event in between. This approach can be applied to intermediate compensation events as well.

To handle non-flow objects affected by the abstraction we propose to use the idea of encapsulation. If a data object is accessed only by the activities within the aggregated fragment, the data object is eliminated. Elimination is possible, since the scope of this data object is restricted to the fragment. If there are activities outside the fragment accessing the data object, the data object is preserved. Instead of associating the data object with the aggregated flow element, we associate it with the aggregating activity. Again, examples are provided in Figure 4, where data object $D1$ is concealed during abstraction of activity D , while $D2$ is preserved and associated with aggregating activities $J+K$ and $L+N$.

If an aggregated activity had a compensation event with the compensation activity, the objects related to compensation are eliminated, as their attachment to the aggregating activity is misleading. A text annotation associated with an aggregated object is associated with the aggregating activity.

Groups visualize logical relations between diagram objects. If abstracted objects belong to one group, the aggregating activity is also included in this group. The task becomes challenging when abstracted activities belong to various groups or some activities belong to a group, while others do not. We will consider such situations in the next subsection.

D. Alternatives and Challenges

Earlier we have presented methods for abstraction in BPDs. Depending on the activity allocation in the model, a certain abstraction method is applied. Decision on the appropriate abstraction method is derived from the communication with an industry partner and our research experience (see [2]). However, in certain cases there is room for alternatives. Here, we discuss the alternative abstraction methods for particular situations, possible enhancements, and the challenges of the approach.

We proposed to abstract an activity within an ad-hoc subprocess via collapsing the parent subprocess. This causes

huge information loss as all the information about the subprocess content is lost. We foresee two alternatives to this abstraction method. The first one is simple elimination of the activity to be abstracted. This method is appropriate when the remaining content of the subprocess is too important for the user. The second alternative is to perform aggregation of two activities in the parent ad-hoc subprocess: the activity to be abstracted and some other. Obviously, model structure does not hint on the choice of the second activity to be used for aggregation. Hence, these two alternatives assume human intervention.

Abstraction of process models with intensive communication between pools may result in two activities with numerous (more than one in each direction) message flows in between. Such an abstract model is overloaded with the message exchange information, but formally tells nothing about the order in which messages are sent and received. The reader understands that there is a “conversation” between the two parties. To model the conversation, two message flows, one in each direction, are sufficient. This operation makes the model simpler, although preserving the general logic.

We foresee a number of challenges in abstraction of BPMN models. Two activities to be aggregated into one may belong to various lanes, while aggregating activity may be assigned only to one. Thus, a decision about the allocation of the aggregating activity is needed. Similar situation takes place for groups: if activities from different groups are aggregated, the assignment of the aggregating activity to a group should be done. In general, there are 3 possible strategies: leave the new activity unassigned, add the activity to every group, or add it to certain groups. Again, this decision can not be derived from the model and should be delegated to the user.

Another challenge is aggregation of error, cancel, compensation, signal, and terminate end events. If at least one of these events is aggregated and does not appear in the abstract model, the semantics of the process changes dramatically. Anomalies, such as deadlocks, missing throwing error or cancel events, may appear in the process. In general case, abstraction of fragments containing such events requires human intervention.

V. RELATED WORK

We realize process model abstraction through structural transformations. There is a large body of research on process model transformations, primarily originating from the domain of process model analysis. The challenge of process model analysis is a large model state space. Hence, the main idea of works in this area is state space reduction through application of special graph transformation rules. Examples of such works are [15], [16], [17], where graph reduction rules facilitate analysis of process model soundness. Reduction rules can be employed for the abstraction task, as it is shown in [2]. The main limitation of this

approach is the difficulty in proving that the set of rules is full, i.e., it enables abstraction of process models with any structure. An approach proposed by Vanhatalo, Völzer, and Leymann in [18] is free of this drawback. Instead of relying on reduction rules, the authors decompose a process model into canonical SESE fragments, which reduce problem state space as well. In contrast to reduction rules, the fragments are not explicitly given, but sought according to their properties. In [19] Vanhatalo, Völzer and Koehler developed the idea, and presented a more granular process model decomposition.

Methods for process model decomposition originate from analysis of program structure. In [8] Johnson, Pearson, and Pingali developed a method for decomposing a program control flow graph into canonical SESE fragments. In [20] the authors presented an algorithm for more fine-granular decomposition of a graph—decomposition into triconnected fragments.

Finally, we would like to mention a series of works on process view construction. In [21] the authors develop an approach enabling derivation of order-preserving process model views. Bobrik, Reichert, and Bauer presented a generic approach to construction of process views in [11]. The approach is rule-based and aims at derivation of process views for different purposes. In [12] the authors presented a method enabling a transition from the detailed private processes of a company to more abstract models, suitable for communication with external partners. Unfortunately, the approach is restricted to block-structured models.

All the works mentioned above did not consider handling of processes captured in BPMN, considering various modeling notations from simplistic graph-based to UML activity diagrams. As we have shown, abstraction of BPDs can be tricky and requires non-trivial extension of available methods.

VI. CONCLUSION

Business process model abstraction has been motivated in the series of papers. These works approached the problem with generic solutions, focusing on the suitable algorithms. However, abstraction of models in notations, having rich expressiveness as BPMN does, has been barely considered.

The main contribution of this paper is the abstraction method addressing various elements of BPMN 1.2. Although the method is driven by abstraction of activities, it also describes how other elements are influenced. The proposed abstraction approach preserves the ordering constraints of the initial model and is capable of handling graph-structured process models. Abstraction is realized as a series of transformations called abstraction steps; it uses aggregation and elimination as basic operations.

We foresee that the results of this paper can be extended in several directions. First, the approach requires practical

validation. Although we summarized the practical experience from the collaboration with an industry partner, analysis of the method applicability will strengthen the results. The next step is a prototypical implementation of the approach presented in this work. Another direction of the future work is an extension of the abstraction approach towards deriving aggregations with meaningful business semantics.

REFERENCES

- [1] M. Weidlich, A. Barros, J. Mendling, and M. Weske, "Vertical Alignment of Process Models - How Can We Get There?" in *BPMDS*, ser. LNBP. Springer, 2009, to appear.
- [2] A. Polyvyanyy, S. Smirnov, and M. Weske, "Reducing Complexity of Large EPCs," in *EPK GI-Workshop*, Saarbrücken, Germany, 11 2008.
- [3] —, "Process Model Abstraction: A Slider Approach," in *EDOC*, 2008.
- [4] —, "The Triconnected Abstraction of Process Models," Hasso Plattner Institute, Tech. Rep. 26, 2008.
- [5] —, "On Application of Structural Decomposition for Process Model Abstraction," in *BPSC*, Leipzig, Germany, 3 2009.
- [6] G. Keller, M. Nüttgens, and A.-W. Scheer, "Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)"", Veröffentlichungen des Instituts für Wirtschaftsinformatik University of Saarland, Tech. Rep. Heft 89, 1992.
- [7] OMG, *Business Process Modeling Notation*, 1st ed., January 2009.
- [8] R. Johnson, D. Pearson, and K. Pingali, "The Program Structure Tree: Computing Control Regions in Linear Time," in *PLDI*. ACM Press, 1994, pp. 171–185.
- [9] G. Booch, R. A. Maksimchuk, M. W. Engel, B. J. Young, J. Conallen, and K. A. Houston, *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison-Wesley Professional, April 2007.
- [10] M. Weske, *Business Process Management: Concepts, Languages, Architectures*. Springer Verlag, 2007.
- [11] R. Bobrik, M. Reichert, and T. Bauer, "View-Based Process Visualization," in *BPM*, ser. LNCS, vol. 4714. Springer, 2007, pp. 88–95.
- [12] R. Eshuis and P. Grefen, "Constructing Customized Process Views," *Data Knowl. Eng.*, vol. 64, no. 2, pp. 419–438, 2008.
- [13] G. Decker, O. Kopp, F. Leymann, K. Pfitzner, and M. Weske, "Modeling Service Choreographies Using BPMN and BPEL4Chor," in *CAiSE*, ser. LNCS, vol. 5074. Springer, 2008, pp. 79–93.
- [14] J. Vanhatalo, H. Völzer, F. Leymann, and S. Moser, "Automatic Workflow Graph Refactoring and Completion," in *ICSOC*, ser. LNCS, vol. 5364, 2008, pp. 100–115.
- [15] J. Mendling, H. Verbeek, B. van Dongen, W. van der Aalst, and G. Neumann, "Detection and Prediction of Errors in EPCs of the SAP Reference Model," *Data Knowl. Eng.*, vol. 64, no. 1, pp. 312–329, 2008.
- [16] W. Sadiq and M. Orłowska, "Analyzing Process Models Using Graph Reduction Techniques," *Information Systems*, vol. 25, no. 2, pp. 117–134, 2000.
- [17] B. van Dongen, M. Jansen-Vullers, H. Verbeek, and W. van der Aalst, "Verification of the SAP Reference Models Using EPC Reduction, State-space Analysis, and Invariants," *Comput. Ind.*, vol. 58, no. 6, pp. 578–601, 2007.
- [18] J. Vanhatalo, H. Völzer, and F. Leymann, "Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition," in *ICSOC*, ser. LNCS, vol. 4749. Springer, 2007, pp. 43–55.
- [19] J. Vanhatalo, H. Völzer, and J. Koehler, "The Refined Process Structure Tree," in *BPM*, ser. LNCS, vol. 5240. Springer, 2008.
- [20] J. Hopcroft and R. Tarjan, "Dividing a Graph into Triconnected Components," *SIAM Journal on Computing*, vol. 2, no. 3, pp. 135–158, 1973.
- [21] D. Liu and M. Shen, "Workflow Modeling for Virtual Processes: an Order-preserving Process-view Approach," *Information Systems*, vol. 28, no. 6, pp. 505–532, 2003.