Parent-Child Relation Between Process Instances

Luise Pufahl(✉) and Mathias Weske

Hasso Plattner Institute at the University of Potsdam, Potsdam, Germany
{Luise.Pufahl,Mathias.Weske}@hpi.de

Abstract. Business process management systems are well equipped to support the enactment of business processes. However, relations between process instances have not sufficiently been taken into account. To improve the execution of related process instances, batch activities have been introduced, which are based on jointly executing process activities. When analyzing real-world business processes, we encountered situations in which activities of specific process instances do not have to be executed at all. To conceptualize these situations, this paper introduces parent-child relationships between instances of a process. The approach is implemented in a cloud-based BPMS, and the technical contribution is embedded in a design methodology. A simulation shows that the cycle time and process execution costs can be significantly reduced by using parent-child relationships between process instances.

Keywords: BPMN · Redesign · Relations of process instances · Parent-child

1 Introduction

Business process management allows organizations to specify, execute, monitor, and improve their business operations [15] using business process management systems (BPMS) [3,6]. In current BPMS, process instances run independently from each other, disregarding relations between them. To improve the execution of related process instance, recent works introduce batch activities for synchronized execution of process instances [9,11,12].

When analyzing real-world business processes, we encountered situations in which activities of specific process instances do not have to be executed at all. An example is an incident process of a large IT service provider. In case of mass disruption, many incidents targeting the same issue arrive in a short period of time. When detected, the first incident of this type becomes a parent incident. Further incidents, the children, can be assigned to this parent. When the parent incident is resolved, its assigned child incidents can take over the result. By re-using results, the assigned children can skip all activities related to solving the incident. These parent-child relations are often already used, but hard-coded in IT systems. We propose to make parent-child relations explicit in process

M. La Rosa et al. (Eds.): BPM Forum 2016, LNBIP 260, pp. 20–37, 2016.
DOI: 10.1007/978-3-319-45468-9_2

models where they are traceable for all stakeholders and can be updated easily in comparison to a hard-coded solution.

This paper introduces parent-child relationships between process instances. As will be shown by a simulation study, this approach leads to an improved process performance. In this work, requirements are elicited based on an interview with a German IT Outsourcing company for integrating parent-child relations into business processes whereby four additional control-flow elements are identified. With these insights, the parent-child relation is integrated into BPMN process models, the industry-standard. A generalized parent-child BPMN process model is given which can be applied to any use case. Additionally, the internal behavior of all introduced activities is described which serves as basis for implementation. This works provides a functional as well as an effectiveness evaluation of the concept. For functional evaluation, the generalized parent-child process is applied to the incident process and its result is implemented in a cloud-based BPMS. The effectiveness evaluation based on a simulation where the basic incident process is compared to the parent-child incident process provides insights in how far a parent-child relation can improve the process efficiency.

The remainder of this paper is structured as follows. Section 2 introduces the motivating example, the incident process, for requirement analysis. Section 3 provides theoretical foundation, based on which we introduce the concept to integrate parent-child relation in BPMN processes in Sect. 4. Section 5 discusses the functional as well as effectiveness evaluation on the incident use case and describes lessons learned. Section 6 is devoted to related work and Sect. 7 concludes the paper.

2 Motivating Example and Requirements

This section presents a motivating example for the parent-child relation, the incident process. For requirement elicitation, an interview was conducted. The interview and the elicited requirements on a process model, which integrates a parent-child relation are discussed in this section.

The process model of Fig. 1 visualizes a simplified version of the incident process described by ITIL V3 [4]. An incident can be received via different channels, e.g. by an email or a call of a user. When an incident is reported, it is logged (i.e., the important information is captured) and categorized. In the next step, the incident is prioritized to define its urgency. Then, the first level support starts with the initial diagnosis. If it is categorized as to be *escalated* by the first level support, it is forwarded for further diagnosis to the 2nd and later maybe also to the 3rd level support. When investigation and diagnosis of the incident is finalized, it is resolved and communicated to the user. Finally, the incident is closed.

In case of massive disruption, incidents that target the same issue arrive, e.g., 100 users call that their email is not working. The handling of massive disruption is currently not captured by ITIL V3. Therefore, we interviewed a German IT outsourcing company to capture the requirements. This outsourcing company

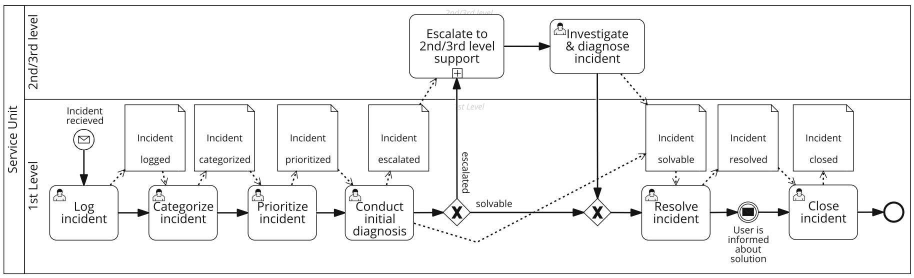

Fig. 1. Simplified incident process as described by ITIL V3.

has hard-coded the incident process in a self-made software for supporting it. If more than two incidents targeting the same issue are identified, they are handled by the outsourcing company as follows: one of them is selected to be the *master incident* to which all others are linked. The master incident is then handled. When it is resolved, the solution is forwarded automatically to all assigned incidents. With this approach, the outsourcing company makes sure that only one solution is followed and streamlines the communication to the user. Further, process performance is improved as the working steps and the solution is only once documented and can be automatically broadcast to similar ones. Currently, this approach is hard-coded in an IT system. Since the master assignment approach is not traceable for the process owner and the participants, its settings cannot be controlled by them and adaptations result in high efforts. We propose to implement it in a process model where it can be accessed and updated by all stakeholders. The process model can be then used for implementation.

From a control flow view point, the described approach requires that the master incident follows the normal flow, but a subset of incidents can skip by re-using the solution of the master all steps after the categorization until sending the message to the user. It has to be ensured that the re-usage of the master solution by the assigned instances only takes place when the master is in a state where its solution is available. For example, the *closed*-state ensures that a solution is available which is not updated anymore. Realizing the parent-child relation for the given example, the process model in Fig. 1 has to be extended with the following four aspects:

(1) An additional activity which checks the existence of a potential master (the parent) after the categorization of the incident
(2) An alternative flow where
(3) child-incidents are assigned to the identified master and can skip all following activities until the solution is communicated to the user
(4) An activity on the alternative flow which applies the result of the master incident and is only enabled, if the master is in state *closed*

In the remainder of this paper, we will present a BPMN diagram serving as template to realize parent-child relations. Thereby, the template will consider the just listed requirements. The following section introduces the foundations for our concept.

3 Foundation

We propose to set up a parent-child relation by means of process data whereby the child instance data references the data of the parent. Therefore, we proceed with introducing formalisms for process and data modeling. Starting with a generic process model definition, we require it to be syntactically correct with respect to the used modeling notation. Behaviorally, we require that it terminates for all execution paths of the model in exactly one of probable multiple end events and that every node participates in at least one execution path, i.e., the process model must be lifelock and deadlock free. Formally, a process model is defined as follows.

Definition 1 (Process Model). A *process model* $m = (N, D, DS, \mathcal{C}, \mathcal{F}, \mathcal{D}, \mathcal{A}, type)$ consists of a finite non-empty set $N \subseteq A \cup E \cup G$ of control flow nodes being activities A, events E, and gateways G (A, E, and G are pairwise disjoint), a finite non-empty set D of data nodes and the finite set DS of data stores used for persistence of data objects (N, D, DS are pairwise disjoint). $\mathcal{C} \subseteq \mathcal{N} \times \mathcal{N}$ is the control flow and $\mathcal{F} \subseteq (\mathcal{A} \times \mathcal{D}) \cup (\mathcal{D} \times \mathcal{A})$ is the data flow relation specifying input/output data dependencies of activities. $\mathcal{D} \subseteq (\mathcal{D} \times \mathcal{DS}) \cup (\mathcal{DS} \times \mathcal{D})$ is the data persistence of data objects and $\mathcal{A} \subseteq (\mathcal{A} \times \mathcal{DS}) \cup (\mathcal{DS} \times \mathcal{A})$ is the data access relation of activities. Function $type : G \rightarrow \{AND, XOR\}$ gives each gateway a type. $\diamond$

We refer to a data node $d \in D$ being read by an activity $a \in A$, i.e., $(d, a) \in \mathcal{F}$, as *input data node* and to a data node d being written by an activity a, i.e., $(a, d) \in \mathcal{F}$, as *output data node*. Figure 1 shows a process model in BPMN notation [10] with one start event, one end event, eight activities (one of them with internal behavior – a sub-process), two XOR-gateways, one intermediate message event and multiple data nodes read and written by activities. Each data node has a name, e.g., *Incident*, and a specific data state, e.g., *logged* or *categorized* (can be represented by a short form *Incident[logged]*). An activity $a \in A$ can have several input and output data nodes, grouped into input sets and output sets. Different input/output sets represent alternative pre-/post conditions for $a \in A$. For example, activity *Conduct initial diagnosis* has two output nodes: *Incident[escalated]* and *Incident[solvable]*, each part of an own output set such that only one of them has to be fulfilled. A data store represents any information system or database. A data node $d \in D$ which is connected with a data store $ds \in DS$, i.e., $(d, ds) \in \mathcal{D}$ indicates that all information of it is stored in this location. In contrast, an activity $a \in A$ connected with a data store $ds \in DS$, i.e., (a, ds) or $(ds, a) \in \mathcal{A}$ indicates that the activity requests or updates the data store. Each data node refers to a data class; here: *Incident*.

Definition 2 (Data Class). *A* data class $c = (J, S)$ *consists of a finite set J of attributes and a finite non-empty set S of data states (J and S are disjoint).* $\diamond$

A data class describes the structure of data nodes in terms of attributes and possible data states which are in a logical and temporal order. The function $\chi : D \rightarrow C$ returns for data node d the corresponding data class c. If we want to express that a data node can be in any state, then the corresponding node gets assigned an asterisks as data state acting as placeholder for each possible state described by the data class. On the execution level, an arbitrary set of data objects exits.

Definition 3 (Data Object and Data State). A *data object* $o = (c, s_o)$ references a data class c describing its structure and allowed data states. Let V be a universe of data attribute values. Then, the *data state* $s_o : J_c \rightarrow V$ is a function which assigns each attribute $j \in J_c$ a value $v \in V$ that holds in the current state of data object o. $\Diamond$

At any point in time, each data attribute of an object can get assigned a value. If it is not defined, the value is set to $\perp$.

Executions of process models are represented by process instances with each instance belonging to exactly one process model m. Each instance contains a set of data objects being tied to the life cycle of the process instance and being disposed as soon as the instance terminates [10] (i.e., case data [13]). Data objects can be made persistent if corresponding data nodes are connected via a data persistent relation to a data store (i.e., work-flow data [13]). We assume that data objects referencing the same data class are stored in one data store ds. If an activity a of a process model has reading access to a data store ds, the process instance can access all stored data objects even if they were not created by it. The function $\delta : I \times DS \nrightarrow \mathcal{P}(\mathcal{O})$ returns for an instance i a set of data objects O_i stored in a data store ds on which it is working. Thereby, $\mathcal{P}(\mathcal{O})$ is the power set of data object set O.

Process instance can be grouped based on data characteristics as introduced in [11] by using the concept of *data views*. In the scope of this paper, we ease this concept such that a *data view DV* is a projection on the values of a data object for a list of logically combined data attributes contained by a *single* data class. This list of fully qualified data attributes is called *data view definition DVD* and is provided by the process designer.

4 Formalizing Parent-Child Relations Based on BPMN

This section presents a concept to integrate parent-child relations into BPMN processes. BPMN [10], a rich and expressive modeling notation, is the industry standard for BPM.

We define a parent-child relation depicted in a process model as a dependency between instances of a process where a set of similar instances is assigned to a so-called *parent instance*. Process instances having similar data characteristics, carrying the same data for certain attributes, are considered as being similar. The assigned instances, the *children*, are allowed to skip a set of connected activities by re-using the result of the parent instance as soon as it is in a certain state in which relevant results for the children are available. Thereby, the goal of the parent-child relation is to save processing time and resource cost by avoiding redundant work. We propose to set up a parent child relation by means of case-to-case data interaction (see pattern 13 in [13]) aiming at passing the parent's data to the child instances during their execution.

In Sect. 4.1, we specify how to model a parent-child relation as BPMN template which can be applied to any use case. In Sect. 4.2, it is described how to implement a parent-child relation by documenting the internal activities' behavior of the template.

4.1 Modeling Parent-Child Relation

In Sect. 2, the motivating example shows that the following elements are needed to realize a parent-child relation: (1) an activity checking whether a parent exists, (2) a skipping flow on which (3) an activity assigns the children to the parent and (4) an activity being enabled when the parent is in a defined state to apply the result of the parent. Based on these insights, a BPMN template to establish a parent-child relation is developed given in Fig. 2. All activities with three dots represent activities which can be adapted or extended to a specific business use case. Basically, a parent-child relation is realized by three additional activities (see Fig. 2, all starting with $a\_$) and a flow for skipping certain activities of the usual flow. This is called the parent-child BPMN fragment. The fragment includes also required data nodes and stores to realize the case-to-case data interaction (see pattern 13 in [13]) for passing data of the parent to its child instances. The three additional activities are service activities such that parent-child relation is realized during process execution automatically with no additional effort for the process participants. We could also hide these service activities in a sub-process to abstract from the details of a parent-child implementation. However, this paper will show the details for explanation purposes. In the following, the parent-child process template is presented in more detail.

For realizing a potential parent-child relation, first the activity $a_{checkForParent}$ is needed on the normal flow. This service activity checks for all instances whether a parent exists. If not, the instance follows the normal flow. If yes, it uses the skipping flow on which it is assigned to the parent and waits for the parent's result. The service activity $a_{checkForParent}$ has one input set consisting of the data node $d[s]$ being from type c_d for which a corresponding parent should be identified. By defining the input data node for this activity, the process designer decides based on which data type the parent-child relation is established. Additionally, the activity has access to a data store ds, the central storage for every produced objects of type c_d to realize the access on other case data.

The activity can produce one of the following output sets: $outputset1 = \{d[s]\}$ and $outputset2 = \{d[s], parent_d[*]\}$. As introduced in the foundation, input and output sets represent alternative pre-/post conditions for an activity $a \in A$. The output sets of an activity a are represented by $out_A = \{\{..\}, \{..\}\}$, a set of sets where each element set contains data nodes in specific states presenting an

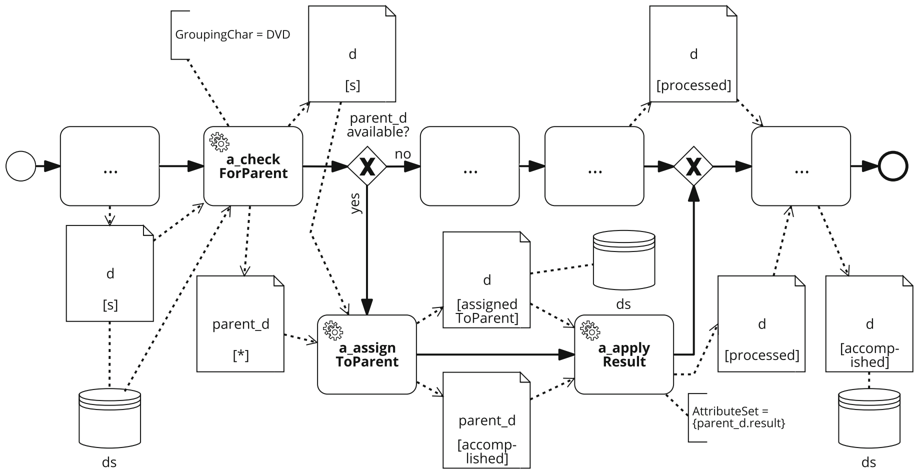

Fig. 2. BPMN process template for a parent-child relation realized by three service activities and a skipping flow.

alternative which can be produced by the activity a. The same applies for input sets of an activity a represented by $in_A = \{\{..\}, \{..\}\}$ where one of the element sets have to be available to enable the activity.

The second output set is only provided if a $parent_d$ – also an object of the data class c_d – was found in the data store ds. The asterisk-state of the parent data object indicates that its state is unknown. For being able to identify the parent data object, every data object d of type c_d has to be available in the data store ds. We assume that previously to or shortly after the activity $a_{checkForParent}$ the data node d is an output data node of any activity and is connected to the data store ds expressing a data persistence relation. In Fig. 2, for example, $d[s]$ is output of the first activity and connected to the data store ds. Further, a data view definition DVD has to be provided by the process designer as grouping characteristic to identify a potential parent. As defined in Sect. 3, a data view definition DVD consists of a qualified set of data attributes of one data class. A parent is identified, if the projection using the attributes of the data view definition on a data object from type c_d, the so-called data view, is equal to the data view of the data object o_i of the currently active process instance i. Here, we assume that the grouping characteristic is designed in a way that the result is in at most one potential parent. If this is not possible, the selection of a parent can be supported by a user decision which can be easily integrated by adding a user activity after this service activity. This user activity presents in its form a list of potential parents and the user can select then either one or no parent.

After identifying whether a parent exists, a splitting OR-gateway is added. It is the decision point between the normal flow and the skipping flow. All child process instances for which a parent was identified, follow the skipping flow. On the skipping flow, the first service activity is $a_{assignToParent}$. It assigns the data object o_i to the parent object o_{parent} by storing a reference to the parent object in the current object and transferring it into the *assignedToParent*-state. Further, it has one input set consisting of the data nodes $d[s]$ and $parent_d[*]$ and produces the following output set with data node d in a new state *assignedToParent* which is in a persistent relation to the data store ds. The persistent relation ensures that the corresponding object can be identified by other new arriving instances as a child object being excluded from the set of potential parents.

Skipping of certain activities by the child instances is only possible, because they apply certain results of the parent. The service activity $a_{applyResult}$ is responsible for this step. It has one input set consisting of the data nodes $d[assignedToParent]$ and $parent_d[accomplished]$. The state *accomplished* of the parent data node is a placeholder and represents the state where a result reusable by the children is available. We assume that each data object is stored in this state such that the parent is accessible in corresponding state. To ensure this, a data node in the state *accomplished* should be output of an activity on the normal process flow – being executed in each case– and should be in a persistent relation to a data store in the process model. For example in Fig. 2, data object d in state *accomplished* is output of the last activity and connected to the data store ds. The corresponding data store ds where the parent data object is stored, has to be requested until the corresponding object o_{parent} has the required state. An implementation of data input conditions is shown for example in the work by Meyer et al. [8]. Further, the activity has one output set consisting of the data node d in the new state *processed*. This output data node indicates that the child object was updated with the values of the parent data object o_{parent} for a set of defined data attributes by the process designer (see text annotation of activity $a_{applyResult}$ in Fig. 2). With termination of this service activity, the child instance can return with the help of a XOR-join gateway to the normal process flow and can follow the process path to the process end.

4.2 Execution Semantics of Parent-Child Relation

In this subsection, the internal behavior of the service activities is described. It serves as implementation support of processes with a parent-child relation. For the description of the internal behavior, pseudo code is used. We start with the first service activity $a_{checkForParent}$ in Algorithm 1.

Algorithm 1 requires a specified data view definition DVD by the process designer, the current process instance i, the input data node $d[s]$ and the data store ds. At first, all data objects being of the same data class as the data node $d[s]$ are retrieved from the data store ds with the help of the auxiliary function $select : DS \times D \nrightarrow \mathfrak{P}(O)$. Thereby, the *select*-function uses χ of the foundations returning for a data node d the corresponding data class c. Further, the local object o_i of the process instance i for the data node $d[s]$ is fetched. Then, for each data object in the set O_d, it is checked whether its data view is equal to the data view of the data object of the current running process instance. Thereby, the auxiliary function $dataView : DVD \times O \rightarrow V$ returns the values for the given attributes in DVD for a data object o. In case of equality, the parent o_{parent} is

Algorithm 1. Algorithm of the service activity $a_{checkForParent}$.

```
Require: $DVD$; // is specified by the process designer
Require: $i$; // current process instance
Require: $d[s]$; $ds$; // the input data node and the data store
 1. $O_d \leftarrow select(ds, d[s])$; // auxiliary function to retrieve all data objects from $ds$
    referencing the same data class as the data node $d[s]$
 2. $o_i \leftarrow i.d[s]$; // get local object for given data node $d[s]$ of the process
    instance $i$
 3. for all $o \in O_d$ do
 4.    if $dataView(DVD, o_i) = dataView(DVD, o)$ & $o_i.id \neq o.id$ then
 5.       // auxilary function dataView returns the projection for the attributes
          given in $DVD$ on a data object
 6.       $o_{parent} = o$; // if data view of current data object is equal to one of the
          data objects, then the parent is identified
 7.       $i.parent_d = o_{parent}$; // the parent data object is stored as local object of
          instance $i$
 8.       $\delta(i, ds) = \delta(i, ds).add(o_{parent})$; // add the identified parent to the set of
          persistent data objects accessed by the current instance $i$
 9.       break; // if parent was identified, loop is stopped
10.    end if
11. end for
```

identified. By additional checking that the *ids* of the two objects are not same, it is ensured that the parent is never the persistent version of the current object. The parent o_{parent} is added to the local data objects of instance i and to the set of persistent data objects $\delta(i, ds)$ in the data store ds on which the process instance i is working by the help of the *delta*-function (see foundation section). In case of successful identification, the loop is terminated.

Algorithm 2. Algorithm of the service activity $a_{assignToParent}$.

```
Require: $i$; // current process instance
Require: $d[s]$; $parent_d[*]$; // input data nodes consisting of the child and parent
Require: $d[assignedToParent]$ ; // output data node
 1. $o_i \leftarrow i.d[s]$;
 2. $o_{parent} \leftarrow i.parent_d[*]$;
 3. $o_i.parentId = o_{parent}.id$; // update current object $o_i$ with a reference to the
    parent
 4. $o_i.state \leftarrow d[assignedToParent].state$; // update state to the output data
    node state
```

If a parent was identified, the activity $a_{assignToParent}$ is executed. Its internal behavior is described in Algorithm 2. It needs the current process instance i, the input data nodes $d[s]$ and $parent_d[*]$ and the output data node

$d[assignedToParent]$ of the activity. First the local object o_i and the local parent object o_{parent} are retrieved from the process instance. Then, a reference to the parent object o_{parent} is set in o_i, here represented by the attribute *parentId*. Finally, the child object state is changed to *assignedToParent* – the state of the output data node.

After assigning the child to the parent, the solution of the parent is taken over. The service activity $a_{applyResult}$ is enabled only if the parent is in a certain state and uses the algorithm shown in Algorithm 3.

Algorithm 3. Algorithm of the service activity $a_{applyResult}$.

```
Require: $i$; // current process instance
Require: $d[assignedToParent]$; $parent_d[accomplished]$; // input data nodes
    consisting of the child and parent
Require: $d[processed]$; // output data node
Require: ATT; // set of attributes defined by the process designer
 1. $o_i \leftarrow i.d[assignedToParent]$;
 2. $O_i \leftarrow \delta(i, ds)$; // get all stored data object of the instance $i$ in $ds$ with the
    $\delta$-function
 3. $o_{parent} \leftarrow O_i.parent_d[accomplished]$// get parent object from the set of stored
    objects of $i$
 4. for all $att \in ATT$ do
 5.    $o_i.att \leftarrow o_{parent}.att$; // update the current object $o_i$ with the results of the
       parent object
 6. end for
 7. $o_i.state \leftarrow d[processed].state$;
```

Algorithm 3 needs the current process instance i, the input data nodes $d[assignedToParent]$ and $parent_d[accomplished]$, the output data node $d[processed]$ of the activity and a set of attributes ATT provided by the process designer describing which attributes values are taken over from the parent. With the *delta*-function, all persistent data objects in ds, in which the process instance i is interested, can be retrieved. From this set the parent data object o_{parent} is fetched. For each attribute of the given attribute set ATT, the value is written to the child object o_i. Finally, the child object state is changed to *processed* – the state of the output data node.

5 Evaluation

In this section, the generalized parent-child BPMN process is evaluated by a functional evaluation and an effectiveness evaluation. For the functional evaluation, the parent-child BPMN process is applied to the incident process, discussed in Sect. 5.1 whereby a methodology for the application is deduced and presented. The result is implemented in effektif.com, a cloud-based BPMN, discussed in Sect. 5.2. In Sect. 5.3, the comparison between a simulation of the basic and parent-child incident process is discussed with regards to cycle times and resource costs. In Sect. 5.4, lessons learned of the evaluation are discussed.

5.1 Application of the Concept to the Incident Process

The presented formalization of the parent-child relation in BPMN shown in Fig. 2 is applied for a functional evaluation to the motivating use case, the incident process shown in Fig. 1. During application, five steps[1] were conducted to apply the parent-child BPMN template. The steps are described in detail in the following.

In step (1), it had to be decided where in the process the parent-child template starting with the service activity $a_{checkForParent}$ and ending with the join XOR-gateway should be added. In the incident process, after logging and categorization the most important data for an incident are available. Prioritization information are only important for the actual diagnosis phase, but not for identifying a potential master incident, the parent. Therefore, the parent-child template was added after incident categorization. Next, it has to be decided how many activities can be skipped. In the incident processes, the idea is to apply the diagnosis and the solution of the master incident and to communicate the solution to each user individually. Therefore, the join-gateway for the normal and splitting flow had to be added after the activity *Resolve incident* and before the send-event. The result is shown in Fig. 3.

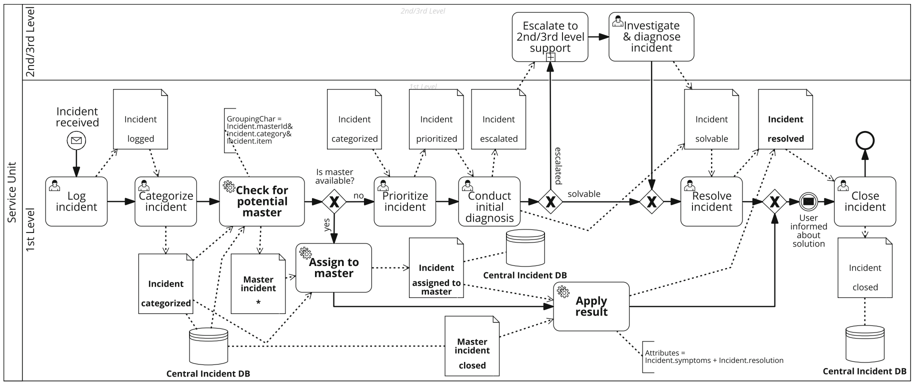

Fig. 3. Incident process extended by the parent-child relation based on the BPMN process template which is adapted to the incident use case.

In step (2), all labels of the template were adapted such that it fits to the use case. Thereby, the service activities, the data nodes with its states and the data stores were relabeled. The service activities were renamed such that the term *master* used by the practitioners is used. For the data nodes, the data class establishing

[1] The documented steps are available at http://bpt.hpi.uni-potsdam.de/Public/ParentChild.

the parent-child relation at runtime had to be chosen. Here, the *incident* data class was selected such that the data node d in the fragment is called *incident* and the parent node $parent_d$ is called *master incident*. Also, the states of the data nodes have to be partly adapted. For example, the input state s of d in the fragment, we relabeled to the output state *categorized* of the categorization activity after which the fragment was added. Similarly, the output state of the fragment *processed* is renamed to the output of the activity *resolve incident* after which normal and splitting flow are joined that is *incident* in state *resolved*. Further, the state of the parent – in the fragment given as $parent_d[accomplished]$ – in which the results can be taken over has to be defined. As shown in Fig. 3, we selected the *closed*-state of the master incident. It assures that incidents can apply the result although the master was already closed. We assume that the incidents are set later in a follow-up process to *archived* indicating that they are not relevant as a master anymore. The data store for the incidents is the *Central incident DB* such that data store ds is adapted accordingly.

In step (3), the grouping characteristic for selecting the right parent and the set of attributes which are taken over by the parent – annotations of the first and last service activity – were adapted. The data view definition consisting of the following incident attributes *category* & *item* & *masterId* was used as grouping characteristic such that the parent is identified based on the categorization information. The attribute *masterId* is also included to make sure that no child instances is selected as master. For an incident, it is important to have finally a description of the symptoms and a resolution. Therefore, *symptoms* and *resolution* are defined as the attributes taken over by the master incident in the annotation of the activity *Apply result*.

In step (4), it had to be assured that data is made persistent at the right spots in the process such that the master can be identified and later the results can be applied. This had to be done at two points: before or right after the check for the potential parent in order to make sure that the parent is available and as soon as the data object is written into the state in which the parent is required. As the most important data is available for an incident after its categorization, the incident output data node is connected to the *Central incident DB*. Additionally, the incident data node in state *closed* is connected to the data base because the child instances require the master in this state (Fig. 3). These are all steps which are needed to design the model with the parent-child relation. In order to execute the model, the internal behavior of the service activities has to be adapted as well in a last step (step (5)). This is discussed in the next sub-section.

The application of the parent-child BPMN process template shows that it is useful to adapt a process easily with a few steps and to make sure that all important elements are included. Based on it, we can summarized that the following important steps:

1. Add the parent-child template at the correct spots in the process,
2. Relabel service activities, data nodes and stores such that it fits to the use case,

3. Adapt annotations of service activities to define a grouping characteristic and the set of attributes taken over by the children,
4. Assure data persistence to enable parent identification and re-usage of results,
5. Adapt internal behavior of service activities for implementation purpose.

5.2 Implementation of Incident Process

The goal of the concept is to provide a parent-child process template which can be used for process design and implementation. Now focusing on the implementation part, we used the incident model of the former subsection and implemented it in effektif.com[2], a cloud-based workflow engine. This workflow engine was selected, because it offers a user-friendly environment where processes can be quickly implemented. effektif.com does not handle data nodes annotated in process models as all current standard BPMS [8], only process variables are supported. Therefore, data relations had to be handled manually.

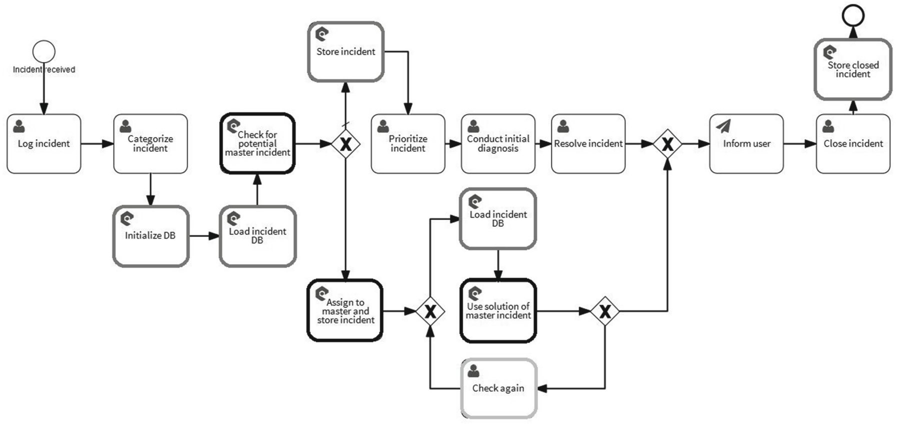

Fig. 4. Implemented incident process in effektif.com with the parent-child relation.

The implemented process is very similar to the incident process of Fig. 3. Minor differences are highlighted in Fig. 4 with a gray boarder. They are mainly due to the missing data support: For the implementation, we eased the process by leaving out the escalation of incidents to focus on the parent-child relation. By default, effektif.com does not provide a connection to a database system, but has Java Script service activities. Therefore, we implemented a web service storage accessible via REST (Representational State Transfer) calls. Due to it, some additional activities were added in the process implementation, e.g., *Initialize incident DB* for setting up the data store, if it does not exist or *Load incident DB* for selecting all data entries. The data input relations for an activity can

[2] Please see http://bpt.hpi.uni-potsdam.de/Public/ParentChild for more information.

not be checked automatically with effektif.com. Thus, a loop was implemented where all incidents are loaded from the store and then the activity *Use solution of master* is executed. If the required variables *symptoms* and *resolution* could be not filled because the master incident object is still not *closed*, a user activity is activated which can restart another try. As soon as the variables are filled, the child instance returns to the normal flow and the user is informed, here by an email activity.

The implementation shows that even if the data nodes concept is not supported by a BPMS, the parent-child relation model served as important orientation for the implementation, i.e., which data has to be accessed and where an access to the data store is needed. The pseudo code given in Sect. 4.2 for the three service activities provided support in how to implement the main java script activities to provide a correct execution of the parent-child relation. Small differences were mainly due to the missing support for data annotations and are documented in their comments. Despite the implementation template, it is still some manual work required from an IT specialist. If the methodology of the previous section is detailed further, the adaption of the service activities to the use case could be partly automized.

5.3 Simulation of the Incident Process

In this sub-section, we evaluate the impact of the parent-child relation on the process performance by a single case. Thereby, the simulation results of the basic incident process (see Fig. 1) are compared to the simulation results of the parent-child incident process (see Fig. 3) with regards to cycle times and resource costs. The interview with the German IT Outsourcing company provided us the average number of cases per day, the average processing times of each activity, the probabilities of decisions and information about the resource number and costs. All information can be found in Table 1.

For our evaluation we used the BIMP simulator[3]. The BIMP simulator offers quick creation of multi-instance simulations by importing a BPMN XML file. This is extended by information regarding the inter-arrival times of instances and activity duration distributions. For the inter-arrival time, an exponential distribution with a mean of one minute was selected. We assume here that incidents only arrive between 9:00 am and 5:00 pm (in sum around 500 cases per day). In reality, also incidents outside of this time with other distributions could be received. However, this is not further considered, because it has no huge impact on the comparison between the basic and the parent-child process. For the activity processing times, a normal distribution was selected for activities which are governed by choice fields, e.g., the categorization or prioritization. An exponential distribution was used for activities where open text fields are included and difficult cases can lead to longer processing times, e.g., the initial diagnosis or investigation and diagnosis. The simulation is eased in this regards

[3] http://bimp.cs.ut.ee/. The BPMN XML files used for the simulation can be found at http://bpt.hpi.uni-potsdam.de/Public/ParentChild.

Table 1. Average process measures for the incident use case provided in the interview with the German IT Outsourcing company.

Process measures		
	# of cases per day	500
Activity processing time	Log incident	4 min
	Categorize incident/prioritize incident	2 min
	Do initial diagnosis	30 min
	Escalate to 2nd level support	3 min
	Investigate and diagnose incident	21 min
	Resolve incident	12 min
Probabilities of decisions	Solvable\escalated incidents	80 %\20 %
	Master available\not available	20 %\80 %
Resource information	# of resources\cost in 1st level support	90\20 Euro per hour
	# of resources\cost in 2nd level support	20\35 Euro per hour

that an incident is only escalated to the second level support, but not further. For the parent-child incident process, assumptions about the processing times of the service activities has to be made. In general, it is assumed that service times are conducted in a few seconds, but sometimes the service might take longer due to server loads. An exponential distribution with a mean of 5 s was selected. The activity *Apply result* is highly influenced by the time when the master is resolved. As this relation cannot be integrated into the BIMP simulator, the distribution of the activity duration has to reflect the waiting time for the master. Therefore, a simulation of the basic incident process was conducted for all activities from incident prioritization until closing to find the average duration a parent needs until it is *closed*. As a child arrives some time after the parent, we assume that the child waits on average half of the needed time. A normal distribution of 25.5 min (51 min\2) with a deviation of 10 min for providing a high variation in the values was taken.

Table 2. Results of one simulation run of the basic incident process and one simulation run of incident process the parent-child relation.

	Cycle time			Process costs	
	Min	Avg	Max	Average case costs	Total costs
(1) Basic process	11.3 min	57.1 min	5.5 h	20.1 EUR	200719.7 EUR
(2) Parent-child process	2.5 min	50.6 min	5.2 h	15.5 EUR	155485.1 EUR
Difference of (2) to (1) in %	77.9 %	11.4 %	5.4 %	22.8 %	22.5 %

Both simulation were conducted for 10000 instance. The results consisting of cycle time and cost information are represented in Table 2. The results show that the average cycle time is reduced by some minutes, a minor improvement of

11.4 %. The main reason is that child incidents have to wait for the result of the parent incident. Further, the portion of probable child incidents is 20 % in this use case. If the portion is greater, the advantage would be even higher. The minimum cycle time has reduced by 77.9 % to 2.5 min, because if an instance is assigned to a parent which is already in state *closed*, the service activities are conducted in a few seconds. In comparison to the cycle time, the parent-child relation has a higher impact on costs. The average costs could be reduced by 22.8 %, similar to the total costs, because the child instances are handled automatically and does not generate any resource costs. To summarize, the single-case simulation shows that the parent-child relation has a minor impact in reduction of cycle time, but a high impact on reducing process costs, because the children bypass a set of user activities.

5.4 Lessons Learned and Limitations

We will now discuss the lessons leaned from the evaluation and its limitations. The effectiveness evaluation based on the simulation implicates that parent-child relations are useful: They can offer time and costs saving, although the costs saving are higher, because the child instances have still to wait for parent instance. Currently, the results are based on one use case. This should be extended to a validation of further use cases, e.g., complaint management where a set of similar complaints targeting the same issue can reuse the result of a already handled complaint. The application of our proposed concept for the integration of parent-child relations – the parent-child relation BPMN process template – shows that it offers support in integrating a parent-child relation correctly in a business process model in a few steps. The most important advantages of integrating parent-child relations in process models are that a parent-child relation is traceable for the process stakeholders, can be easily adapted and can be used for process validation (e.g. simulation) and implementation. Nevertheless, some manual work by the process designer to adapt the fragment to a concrete use case is still needed. Similar results shows the process implementation: The resulted parent-child process model and the pseudo code description provide an orientation for a correct implementation despite missing support for data annotation in BPMS, but manual work is still needed. However, the proposed methodology, the 5-step approach, deduced from application to the incident use case can be used as a prerequisite for (semi) automation of our concept. An automatized approach could use a basic process model as input and some user inputs to generate a process model extended with the parent-child relation based on them. In future, the first deduced methodology should be detailed and further evaluated.

6 Related Work

The workflow control-flow patterns of [1] are an important standard that describe workflow functionalities with regard to the control-flow as patterns. However, patterns for interrelations between process instances are not considered. A first step in this regard are the multi-instances patterns, but for them it is assumed that multiple instances are created and terminated during the execution of one process instance and does not consider relations between instances in general.

In BPMN, the signal event can be used for intra- and inter-process communication [10]. However, a signal is "like a shot into the sky" where each instance reacts which has subscribed for this signal type. A signal is not instance-specific. Therefore, we realized the parent-child relation between process instances based on the instance data.

Whereas the parent-child relation between instances of a process model received little attention in the BPM research, the batch relation, was discussed in several works e.g., in [2,7,14] and also recently e.g., in [9,11,12]. In a batch relation, instances of a batch are processed together in one step (e.g. the instances' attributes are shown in one user view) whereas in a parent-child relation, the parent executes the normal flow; the children can skip certain activities by using the result of it. In both relations, similar instances have to be identified which can be in a batch or parent-child relation. The process designer has to define on which data attributes process instances can be grouped.

In PHILharmonicFlows [5], a data-oriented modeling approach, it is possible that the user enters form values in one go for multiple data objects. This requires that the set of children is known before the parent can be processed further such that the results can be applied. In our approach, child instances can also use the result of the parent, if it is already terminated. PHILharmonicFlows considers also execution dependencies where certain object instances have to be available for process continuation, but parent-child relation are not discussed in their work.

7 Conclusion

In this paper, parent-child relation of process instances where a set of instances can skip certain activities by reusing results of a parent instance is investigated for the first time. The explicit representation of parent-child relations in process models enables the traceability, validation, optimization and correct implementation of the parent-child relation for the process stakeholders. This work provides requirements for the integration, and further a definition and formalization of the parent-child relation in BPMN, the industry standard. Our contribution consists of a parent-child BPMN process template with all necessary elements to model a parent child-relation and the internal activity specification as prerequisite for implementation. Activities of the parent-child template are all service activities to avoid additional effort for process participants.

The functional evaluation where the parent-child template is applied to a use case shows that the concept helps to adapt a process in a few steps which is then able to correctly execute a parent-child relation. The application identified a five step methodology, a first step in the direction to make the integration of parent-child relations in business processes (semi) automatized. It can be generalized by applying it to other use cases, e.g., the complaint management, to use it for an automatic integration approach. Implementation of the resulted parent-child process shows that despite of missing data support in existing BPMS, the model and the internal behavior provide an important support for a correct parent-child implementation. The results of the effectiveness evaluation by simulating a single use case indicate significant savings in cycle time and costs by a parent-child relation. This single case simulation should be extended in future. Further, we aim to evaluate the usability of our approach in a user study.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1), 5–51 (2003)
2. van der Aalst, W.M.P., Barthelmess, P., Ellis, C.A., Wainer, J.: Proclets: a framework for lightweight interacting workflow processes. Int. J. Coop. Inf. Syst. **10**(4), 443–481 (2001)
3. Camunda: camunda BPM platform. https://www.camunda.org/
4. Großbritannien Office of Government Commerce: Service operation (SO): ITIL. TSO (The Stationery Office) (2007)
5. Künzle, V., Reichert, M.: PHILharmonicFlows: towards a framework for object-aware process management. J. Softw. Maint. **23**(4), 205–244 (2011)
6. Lanz, A., Reichert, M., Dadam, P.: Robust and flexible error handling in the aristaflow BPM suite. In: Proper, E., Soffer, P. (eds.) CAiSE Forum 2010. LNBIP, vol. 72, pp. 174–189. Springer, Heidelberg (2011)
7. Liu, J., Hu, J.: Dynamic batch processing in workflows: model and implementation. Future Gener. Comput. Syst. **23**(3), 338–347 (2007)
8. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and enacting complex data dependencies in business processes. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 171–186. Springer, Heidelberg (2013)
9. Natschläger, C., Bögl, A., Geist, V.: Optimizing resource utilization by combining running business process instances. In: Toumani, F., et al. (eds.) ICSOC 2014. LNCS, vol. 8954, pp. 120–126. Springer, Heidelberg (2015)
10. OMG: Business Process Model and Notation (BPMN), Version 2.0, January 2011
11. Pufahl, L., Meyer, A., Weske, M.: Batch regions: process instance synchronization based on data. In: EDOC, pp. 150–159. IEEE (2014)
12. Pufahl, L., Weske, M.: Batch activities in process modeling and execution. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 283–297. Springer, Heidelberg (2013)
13. Russell, N., Hofstede, A.H.M., Edmond, D., Aalst, W.M.P.: Workflow data patterns. Queensland University of Technology, Tech. rep. (2004)
14. Sadiq, S., Orlowska, M., Sadiq, W., Schulz, K.: When workflows will not deliver: the case of contradicting work practice. In: BIS, pp. 69–84 (2005)
15. Weske, M.: Business Process Management: Concepts, Languages, Architectures, p. 404, Second Edition. Springer, Heidelberg (2012)