# Batch Activities in Process Modeling and Execution

Luise Pufahl and Mathias Weske

Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Strasse 2-3, 14482 Potsdam
{luise.pufahl,mathias.weske}@hpi.uni-potsdam.de

**Abstract.** In today's process engines, instances of a process usually run independently to each other. However, in certain situations a synchronized execution of a group of instances of the same process is necessary especially to allow the comparison of business cases or to improve process performance. In this paper, we introduce the concept of batch activities to process modeling and execution. We provide the possibility to assign a batch model to an activity for making it a batch activity. As opposed to related approaches, the batch model has several parameters with which the process designer can configure individually the batch execution. A rule-based batch activation is used to enable a flexible batch handling. Our approach allows that several batches can run in parallel in case of multiple resources. The applicability of the approach is illustrated in a case study.

**Keywords:** batch activity, process modeling, synchronization of instances.

## 1 Introduction

In today's organizations, modeling of business processes and their execution based on process-oriented systems has a high relevance. Business operations are usually specified by process models with focus on the single business case. A process model describes a set of activities jointly realizing a business goal and the execution constraints between them [16]. At runtime, for each business case, a process instance is created. When designing a process, it is typically assumed that process instances are completely independent from each other [4]. Also in process engines, instances are usually executed individually. Nevertheless, certain dependencies between process instances may require a synchronization. In this paper, we introduce an approach for coordinating the activity execution of different process instances motivated by the following example.

Fig. 1 shows the *Train ticket refund* process of a train company in which passenger claims are received and checked. When a passenger experienced a delay of more than one hour, the company provides a voucher card with an amount of 50% of the train ticket price. A process instance is started when a claim for refund is received from a passenger . Then, for each activity in the process model, an activity instance is created as soon as a process runs. Usually, activity instances are executed independently from each other. For instance, the company enters the data of each claim individually and checks whether the claim is correct. However, activities in process models can be observed for which it is beneficial or even required to synchronize the execution of a group of business cases,

S. Basu et al. (Eds.): ICSOC 2013, LNCS 8274, pp. 283–297, 2013.

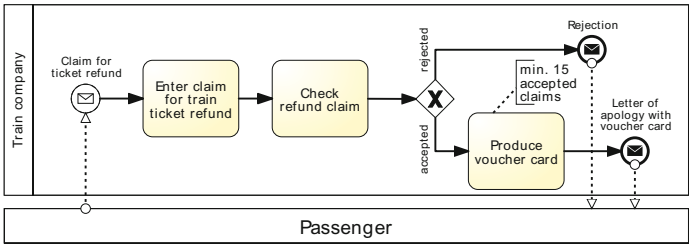



**Fig. 1.** Train ticket refund process

e.g., the activity *Produce voucher card*. The train company activates the machine for producing the voucher cards only when 15 cards are requested in order to save setup costs. Such a type of activity, we call *batch activity* which is defined as an activity clustering a set of active activity instances together and synchronizing their execution according to pre-defined rules [4, 13]. Currently, the batch activation rule is informally noted as comment on the respective activity (cf. Fig. 1). The goal of this paper is to formalize the design of batch activities and to give a blue print for them. Two types of use cases for which a batch activity is needed, can be differentiated [2]:

- **Achieving an increased process performance**: A process may have an activity with high setup costs, i.e., preparation costs to start an activity (e.g., setups of machines, familiarization periods for a type of work or traveling distances). In our example, we have the setup costs of the voucher card machine. By synchronizing the activity enactment of several cases, the train company can save those costs and can be more efficient in their process execution.
- **Comparing business cases**: A process may have an activity where business cases are ranked according to specific criteria. In order to be able to compare them, several cases have to be grouped together, e.g., a ranking of application candidates.

A common assumption is that batch requirements can be solved with multi-instance patterns [14] which are supported by several modeling languages. For example, the widely applied process modeling language BPMN (Business Process Modeling Notation) provides the concept of multi-instance activities. When a multi-instance activity is started in the context of a process instance, multiple activity instances are initialized simultaneously running independently from each other. Synchronization may be organized with regards to starting the subsequent activity, but their execution is not aligned. Since multi-instance activities have an opposite execution paradigm which splits one instance as opposed to synchronize multiple existing instances, a new concept for a batch handling needs to be developed.

Also, most process modeling languages do not support the design and configuration of batch activities; they are often enacted as batch manually or by special software. Organized manually by human resources, the rules of a batch activity can be unclear or the batch execution may simply be forgotten resulting in lower process performance. Otherwise, a batch activity may be controlled by specific software. Since the batch configurations are then not traceable for the process owner and the participants, the batch activity settings cannot be controlled by them and adaptations result in high efforts.

In this paper, we propose a concept to integrate batch activities in process modeling and execution. The integration has the advantages that (i) rules are clear for the process owner, the participants, and the process engineer, (ii) no manual implementation is required, (iii) batch activities can be included into potential process simulations, and (iv) monitoring as well as analysis of executed batches can be done based on process logs.

The paper is structured as follows. In Section 2, we discuss requirements for integrating the concept of batch activities into process modeling and execution. Then, we present our approach to enable the design, configuration and execution of batch activities in Section 3, where we also discuss the applicability of our approach in a case study. In Section 4, related work is discussed followed by a conclusion in Section 5.

## 2 Requirements of Integrating Batch Activities

In the following, we present several requirements to integrate batch activities in process modeling and execution, summarized and related to each other in Fig. 2. On the one hand, they arise due to the different execution semantics of a batch activity (cf. path (b) in Fig. 2) in comparison to the regular one (cf. path (a) in Fig. 2) (R1). On the other hand, we collect them based on descriptions of the batch service problem. The batch service problem was investigated by the queuing research, e.g., in [8–10], and is described as follows: "Customers arrive at random, form a single queue in order of arrival and are served in batches" [1], whereby the basic object of investigation is when to start a batch (R2). Queuing researchers investigated it for different configurations of queue (R3) and server (R4-6). In the following, we will discuss the identified requirements in detail.

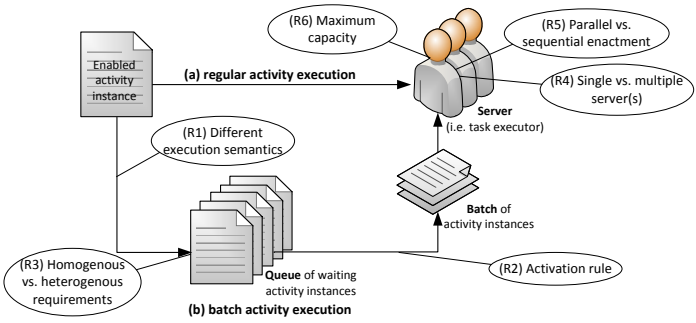



**Fig. 2.** Requirements regarding integration of batch activities

*Requirement R1 - Different execution semantics:* A usual activity instance passes different states during its lifetime [16]. In a simplified version, these are *init*, *ready*, *running*, and *terminated*. With the start of a process, the activity instance enters the *init* state and changes to the *ready* state when all pre-conditions for the activity are fulfilled. Then, it is immediately offered by the process engine to the respective task executor (cf. path (a) in Fig. 2). The task executor can be either a software service, a human, or a non-human resource. When a resource or service starts the work of the activity instance, it enters the *running* state and with completion, it is in the *terminated* state.

In contrast, the offer of a batch activity instance has to be delayed by the process engine in order to provide the task executor a group of instances being executed as batch. Here, it is required that the process engine collects all enabled activity instances according to a queue discipline, e.g., first-come first-served (FCFS) or last-come first-served (LCFS), and assigns them to a batch as depicted in Fig. 2. In this paper, we choose FCFS as queuing system, because it is a commonly applied policy [8].

*Requirement R2 - Activation rule:* With a batch activity, the control of starting a group of instances is allocated to the process engine to achieve an increased process performance or to enable the comparison of business cases. The larger the size of a batch, the lower are the average setup costs per activity instance, but the higher are the waiting time per instance. So, costs can be reduced, when the train company waits at minimum for 30 instead of 15 claims for which a voucher card is produced. However, the risk for the train company increases to lose passengers, because of dissatisfaction regarding the service time. Rules have to be specified and enforced in order to achieve an optimal trade-off between setup costs and waiting time. As stated, this optimization problem – when to activate a batch service and provide it to the server – is investigated by the queuing research for which they propose different optimization policies. In this paper, we want to present two often discussed rules [10]:

- **Threshold rule**: Originally called the *general bulk service rule*, it states that a batch is started, when the length of the waiting queue with customers is equal or greater than a given threshold (i.e., a value between one and the maximum server capacity) and the server is free [9]. Several studies investigate how to determine an optimal value for the threshold under varying assumptions concerning the distribution of arrival and service times as well as capacity constraints of server and queue (an overview is for example given by Medhi [8]). In this paper, we assume that the threshold value is given by the process designer who may derive it from expert knowledge, simulations, or statistical evaluations. This rule can be extended by a maximum waiting time so that a group of less than the threshold is also served, when a certain waiting time of the longest paused one is exceeded [9].
- **Cost-based rule**: Originally called the *derivation policy*, it states that a batch is started, when the total waiting costs of all customers in the waiting queue is equal or greater then the total service costs and the server is free [15]. The total waiting costs are the sum of costs for each waiting customer based on the given penalty costs per time period. The total service costs can be either a constant value or they are a function considering the number of customers.

Besides these two, other types of activation rules exist. Thus, it should be possible to provide different types of activation rules from which a process designer can select one for a batch activity and fill it with required user inputs.

*Requirement R3 - Homogeneous vs. heterogeneous requirements:* In studies from queuing research, it is discussed that arrived customers may be homogeneous or heterogeneous in their demand [10]. If they are heterogeneous, different types of batches have to be formed. Also activity instances can be heterogeneous regarding their inputs respectively required outputs. In our train example, all *Produce voucher card*-instances

are currently homogeneous. The activity instances will become heterogeneous concerning their output requirements, for example, if the train company produces two types of voucher cards; white colored ones for amounts lower €150 and better protected once in silver for amounts equal or greater than €150. In this case, two different batch types have to be created. However, we assume in this work that all activity instances are homogeneous.

*Requirement R4 - Single vs. multiple server(s):* Often, studies from queuing research assume that only a single server is available [8]. Hereby, it is assumed that the availability of the server is controlled and the activation of a batch depends on it. Business process management differentiates between the control flow perspective, where the batch activity is part of, and the resource perspective concentrating on the modeling of resources and the allocation of work. Russell et al. [11] present several resource patterns for offering, allocating, and detouring work after an activity was enabled (e.g., the role-based allocation). We require that the batch activity execution should not interfere with the concepts of the resource perspective. Therefore, we assume the general case that a batch can be provided to multiple available resources. Here, the engine should be able to run several batches in parallel when they are needed.

*Requirement R5 - Parallel vs. sequential enactment:* Batch processing occurs in two versions: parallel and sequential execution [7]. In parallel batch execution, the activity instances of a batch are processed by the task executor simultaneously, because the server capacity is greater than one. An example for it is the voucher card machine of the train company which can produce more than one card in a run. In sequential batch execution, the task executor enacts the activity instances one after another. They are processed as batch, because they share the same setup, e.g., the setup of a machine, a traveling distance. An example is the task of controlling exams where the examiner needs a certain familiarization phase for each examination question and then checks the answers of all students for one question.

*Requirement R6 - Maximum capacity:* An often discussed constraint of the batch service problem is the maximum capacity of the task executor respectively the maximum number of cases that the executor can handle in a sequence [8]. For instance, the voucher card machine may be able to produce at maximum 25 cards in a run. This capacity determines the maximum size of a batch, which is 25 for the *Train ticket refund* example. If a batch is activated and offered to its executor, but not yet started, because the executor is not available, it should be still possible to add further activity instances until the maximum is achieved or the batch is started. This leads to an increased process performance.

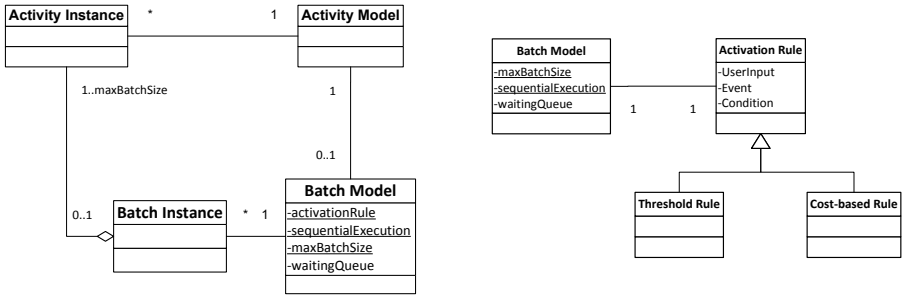
## 3 Integrate Batch Activities in Process Modeling and Execution

In this section, a general approach to model and configure a batch activity in process modeling languages is presented. The de facto standard BPMN is used to illustrate it. For the approach, we augmented the process meta model by Weske [16] with the required concepts for batch activities which is explained in detail in Section 3.1. For the enactment of batch activities in a process engine, an enhanced engine architecture

and execution semantics is proposed in Section 3.2. We evaluate the applicability of our approach based on a case study in Section 3.3.

### 3.1 Modeling and Configuration of a Batch Activity

In this work, we want to provide the possibility to design and configure a specific activity as batch activity in order to synchronize the execution of its instances. Therefore, we extended the process meta model by Weske [16]. The meta model describes that a process model consists of nodes and edges. A process model acts as blueprint for a set of process instances which are related to exactly one process model. A node in a process model can represent an event, a gateway, or an activity model. Similar as the node, the activity model is associated to an arbitrary number of activity instances (see Fig. 3(a)) for which it describes the key characteristics, e.g., resource assignment, input and output data. An activity model can be a system, user interaction, or manual activity.




(a) Batch activity meta-model (b) Rule-based initialization concept

**Fig. 3.** Extension of process meta model for batch activities. In (a), we show that a batch activity can be designed by assigning a batch model with several configuration parameters to an activity model. In (b), we show that each batch model gets assigned an activation rule which can be from different types (here from type threshold or cost-based) to enable a rule-based batch initialization.

We extend these concepts by the batch model (see class diagram of Fig. 3(a)); an activity becomes a batch activity, if its activity model is associated to a batch model which in turn can only be associated with exactly one activity model. A batch model describes the conditions for batch execution and can be configured based on the parameters *activationRule*, *sequentialExecution*, and *maxBatchSize* by the process designer.

- The *activationRule* provides the possibility to specify a policy when a batch is enabled and offered to the task executor. Therefore, the process designer selects an activation rule type (e.g., threshold rule) and provides required user inputs (see R2).
- The *sequentialExecution* is of type boolean. In case of *false*, all instances of a batch are provided at once to task executor for parallel execution. In case of *true*, instances are provided one after another to the executor for sequential execution (see R5).
- The *maxBatchSize* is of type integer and represents the maximum capacity of the task executor. It specifies the number of instances which can be at maximum in a batch. It can be limited by user input or unlimited without user input (see R6).

In Fig. 4, we show an exemplary configuration by means of the batch activity *B* in the example process *P*. In this context, we illustrate a batch activity in BPMN by a double framed activity. For the batch activity *B*, the process designer selected the threshold rule as activation rule with a threshold of two and a maximum waiting time of one hour (i.e., the batch is offered to the task executor latest after an hour). The batch activity was configured such that at maximum three instances of *B* can be in a batch and they are all executed in parallel, because the *sequentialExecution* is set to false.

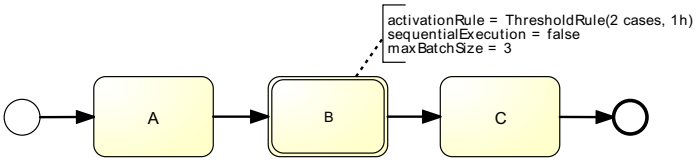



**Fig. 4.** Example process *P* with three activities whereby *A* and *C* are usual single case activities and *B* is a batch activity illustrated here with a double border

In summary, a batch model associated to an activity model describes with its configurations the behavior for an arbitrary set of batch instances. A batch instance represents one batch and is responsible for its initiation and execution. Several batch instances being associated to exactly one batch model can exist simultaneously to allow the parallel execution of batches (see R4). Thus, when a batch instance is currently executed, another instance can already be initiated when it is required, and can be allocated, e.g., to an alternative resource. We will give more details on how the parallel run of batch instances can be organized and implemented in the next section.

Enabled activity instances are associated to a batch instance. Each batch instance has its own waiting queue where all its assigned activity instances are collected in order of their arrival (see R1). At minimum, the queue has the size of one, because a batch instance is only initialized when it is required by at least one instance, and at maximum it has a size of the user-specified *maxBatchSize*. A batch instance also passes the states *init*, *ready*, *running* and *terminated*. Thereby, a batch instance changes from the *init* to *ready* state as soon as the predefined activation rule is fulfilled. Then, the batch of activity instances is offered to the task executor. When a resource accepts it, the batch instance enters the *running* state and as soon as the batch work is completed, it changes into the *terminated* state. Additionally, we extended the life cycle for batch instances so that they can also be in state *maxloaded* after being *ready* and before being *running* when it reaches the specified maximum size of a batch. After a batch instance is initialized and as long as it does not enter the *maxloaded* or *running* state, activity instances can be bound to it. So, we ensure that an optimal number of activity instances is added to a batch instance.

As described, the process designer selects an activation rule type for a batch model and configures it with the required user inputs. Thus, each batch model is associated with an activation rule as shown in Fig. 3(b). We assume that process engine suppliers provide different types of activation rules in advance, e.g., the threshold rule or cost-based rule from queuing research presented in Section 2. In general, an activation rule

relies on the concept of ECA (Event Condition Action) rules. Basic elements of an ECA rule are an event E triggering the rule, a condition C which has to be satisfied, and an action A being executed in case of fulfillment of the condition [3].

Thus, we define an activation rule as a tuple $E \times C \times A$, whereby the action $A$ is always the enablement of the associated batch instance. An event $E$ is either an atomic event (e.g., a state change of the batch waiting queue or a specific time event) or a composite event being a composition of atomic events through logical operators, as for instance *AND* or *OR*. A condition $C$ is a boolean function. The input elements to such a function can be system parameters (e.g., *actual length of waiting queue*), user inputs (e.g., THRESHOLD), or a combination of both (e.g., total service costs = (VARIABLE COSTS $*$ *actual length of waiting queue*) + CONSTANT COSTS) connected by a relational expression. The composition of several atomic conditions with logical operators is called composite condition.

An example for the threshold rule is given below. In this activation rule, the user inputs are indicated by capitals and the system parameters are italicized. It consists of a composite event saying that the rule is triggered when a new activity instance was added to the waiting queue of the associated batch instance *b* or when no new one was added for a specific period, i.e., the user-specified maximum waiting time divided by ten. With triggering the rule, the given composite condition is checked. It states that either the length of the waiting queue has to be equal or greater than the user-specified threshold or the lifetime of *b* has to be equal or greater than the user-specified maximum waiting time. If the condition evaluates to true, *b* gets enabled.

```
ActivationRule Threshold rule
  On Event     (Instance added to b.waitingQueue) OR
                (No instance since MaxWaitingTime/10)
  If Condition (b.waitingQueue.length ≥ Threshold)OR
                (b.lifetime ≥ MaxWaitingTime)
  Do Action    Enable batch instance b
End ActivationRule
```

We enable the integration of batch activities into process modeling with few extensions on the existing process meta model. In the next section, we propose an architecture and execution semantics for batch activities.

### 3.2 Execution of a Batch Activity

In Fig. 5, we present an abstract architecture of a usual process engine (cf. white elements). Process models are saved in a repository on which the process engine has read access. As soon as a start event occurs for a process, the engine initializes an instance of this process. Thereby, the process instance controls the initialization and enablement of each of its activity instances based on the control flow specification in the process model. Exemplary, we show in Fig. 5 process instances of our example process *P* with its activity instances. The process engine is able to offer and allocate the work of activity instances to task executors. For service activities, the respective service is invoked by the engine. User interaction activities are provided via a task management component

with a graphical interface to the process participants. This approach aims at relating a batch configuration to any activity type (i.e., system, user interaction, or manual activity). Thus, we abstract from the service invocation and task management components.

In order to execute a batch activity, the architecture is extended with batch instances and a batch factory (cf. shaded elements in Fig. 5 for the example batch activity *B*). For each batch activity, a batch factory exists which is responsible for mapping activity instances to batch instances and for initializing new batch instances when required. Thus, activity instances request the batch factory – in Fig. 5 the instances of the batch activity *B* – for being associated to a batch instance. A batch instance in turn communicates with one or more associated activity instances as well as with the process engine which can offer the batch to the executor.

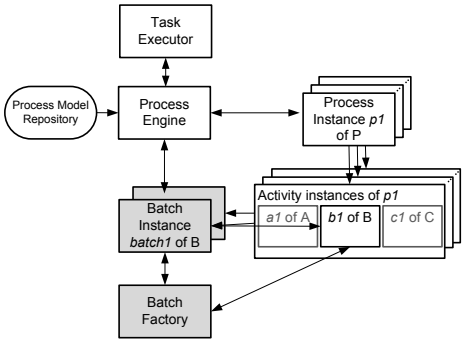



**Fig. 5.** Process engine architecture (white elements) with extensions for batch activity execution (shaded elements)

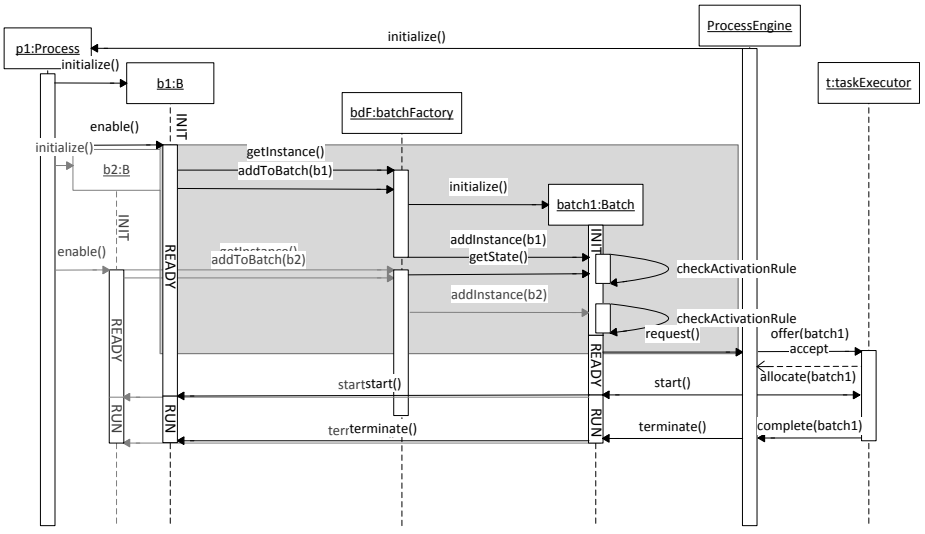



**Fig. 6.** Execution semantics of a batch activity

An example scenario of the activity instance *b1* of the batch activity *B* being part of the process instance *p1* and associated to the batch instance *batch1* is represented in the sequence diagram of Fig. 6. It illustrates the execution semantics of an activity instance of a batch activity which is a refinement (cf. shaded box in Fig. 6) of the common activity instance execution. The main difference is that instead of offering the activity directly to the task executor after its enablement, it is added to a batch instance which then controls its start and termination. In the sequence diagram, we label the activation

line of the *activity instance* and *batch instance* entities with the states they are currently in. Next, we discuss the sequence diagram in detail.

As usual, the process engine initializes the instance *p1* of the process *P* and the instance initializes all its activity instances including *b1*. Based on the control flow specification, the process instance enables *b1* as soon as the preceding activity *A* was terminated. Differently to the usual activity instance, *b1* requests the *batchFactory* for being added to a batch instance with the function *addToBatch*. We propose to implement the *batchFactory* class as singleton for having only one responsible object for the mapping of activity instances to batch instances. It decides whether it can add an activity instance to a still not *maxloaded*, *running*, or *terminated* batch instance or whether it has to initialize a new batch instance. Thereby, the requests by the activity instances are sequentialized such to prevent an inconsistent state of the system. We developed the following algorithm for the *batchFactory* class:

**Algorithm 1.** Algorithm for the function *addToBatch*

**Require:** $i$:activity instance
  **if** $availBatch$ not null **then**
    **if** $availBatch$.getState ! = INIT || READY **then**
      $availBatch$ = initialize new batch();
    **end if**
  **else**
    $availBatch$ = initialize new batch();
  **end if**
  $availBatch$.addInstance($i$);

The *batchFactory* class has an attribute *availBatch* where the potential batch instance to which activity instances can be still added, is saved. When the algorithm is called with an activity instance *i* and the *availBatch* is empty, a new batch instance is initialized and set as *availBatch*. If it is not empty, the state of the corresponding batch instance is checked with the *getState*-function. In case, it is not in state *init* or *ready*, a new batch instance is initialized and set as *availBatch*. After these checks, the given activity instance is added to the current *availBatch*. With the algorithm, we ensure that a maximal load of a batch can be achieved, but if a batch instance has reached its maximum capacity or is already executed, no new instances are added to it.

In our example, the threshold rule with a threshold of two was selected as activation rule for the batch activity *B* (cf. Fig. 4). When the activity instance *b1* was added to the waiting queue of *batch1*, its activation rule is checked, because it is an event which triggers to check the condition. Due to the waiting queue length of one, the condition is evaluated to false and the action is not executed. For reason of complexity reduction, we represent the activation rule in Fig. 6 as function and not as own object. With the second added activity instance *b2*, *batch1* enters the *ready* state, because now the condition that the waiting queue length is greater or equal to the threshold is fulfilled. Then, the process engine is requested to offer *batch1* to the task executor. With acceptance of the task executor, the engine allocates the work of all associated activity instances at once and starts the batch instance. Having entered the *running* state, the batch starts all its activity

instances of its waiting queue. At some time, the task executor completes the batch. Then, the engine terminates the batch and *batch1* in turn terminates its activity instances *b1* and *b2*. In case that the sequential execution was selected for a batch activity, the engine acts slightly different compared to the parallel execution: After the batch was accepted by a resource, the engine starts the batch instance. The batch instance provides in a loop its associated activity instances one after another over the engine to the task executor. The currently provided activity instance is then started and terminated by it.

After we presented the architecture and execution semantics of a batch activity, we will illustrate our approach using a case study in the next section.

### 3.3 Case Study

We already provided some insights into the *Train ticket refund* process in the introduction on which we will focus in our case study. Fig. 7 shows a variant of it as BPMN model.

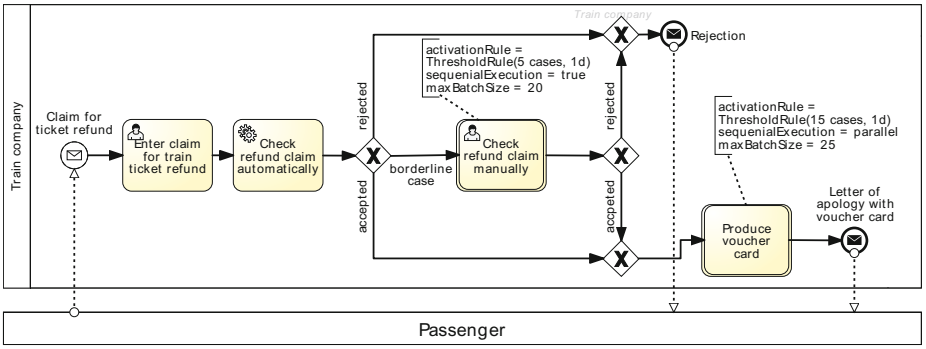



**Fig. 7.** Train ticket refund process with manual claim check

After a claim by a passenger is received in a form, the data of the form is entered into the process system of the train company. We assume that our company receives 150 claims per day on average. Based on the entered data, the process system runs an automatic check whether the passenger claim is accepted or rejected. Some claims – in average 5% – are classified as borderline cases where the system check could not result in a clear decision, e.g., when a passenger experienced a delay of 56 minutes instead of an hour. An employee has to check them manually a second time.

We assume that the employee needs a familiarization phase of approximately four minutes for this task to get afresh familiar with proceeding guidelines and rules. In order to save setup time, the employee shall process at minimum a set of five cases. The employee can organize this for her-/himself. However, the employee may get disturbed from each case arriving in the work list while working on other tasks. We propose to install a batch activity so that the employee gets offered the work only when necessary. The requirements are that at minimum five cases should be processed in a sequence, but not more than 20 cases due to the risk of decreased motivation, and no case should wait longer than one day until being provided to task executor to ensure short response time for passengers. We capture them in our batch activity (cf. *Check refund claim manually* in Fig. 7).

**Table 1.** Activity instance log of *Check refund claim manually*: Each row presents an activity instance, i.e., when it entered a certain state and its relating batch instance

| id | *init* | *ready* | *running* | *terminated* | batch |
|---|---|---|---|---|---|
| ... | ... | ... | ... | ... | |
| i31 | 13/03/11 09:31 am | 13/03/11 10:20 am | 13/03/11 03:40 pm | 13/03/11 03:51 pm | 11 |
| i37 | 13/03/11 09:58 am | 13/03/11 11:01 am | 13/03/11 03:51 pm | 13/03/11 03:57 pm | 11 |
| i40 | 13/03/11 12:01 am | 13/03/11 12:17 am | 13/03/11 03:57 pm | 13/03/11 04:02 pm | 11 |
| i48 | 13/03/11 02:33 pm | 13/03/11 02:46 pm | 13/03/11 04:02 pm | 13/03/11 04:09 pm | 11 |
| i51 | 13/03/11 02:54 pm | 13/03/11 03:14 pm | 13/03/11 04:09 pm | 13/03/11 04:16 pm | 11 |
| i59 | 13/03/11 03:07 pm | 13/03/11 03:32 pm | 13/03/12 04:16 pm | 13/03/12 04:23 pm | 11 |
| i64 | 13/03/11 04:02 pm | 13/03/11 04:13 pm | 14/03/12 04:13 pm | 14/03/12 04:25 pm | 12 |
| i76 | 14/03/12 10:21 am | 14/03/12 10:45 am | 14/03/12 04:25 pm | 14/03/12 04:31 pm | 12 |
| i83 | 14/03/12 02:40 pm | 14/03/12 02:49 pm | 14/03/12 04:31 pm | 14/03/12 04:36 pm | 12 |
| ... | ... | ... | ... | ... | .. |

In Tables 1 and 2, we show exemplary extracts of the activity and batch log of *Check refund claim manually* in order to illustrate the batch activity execution. A new activity instance *i31* initialized at 9:31 am gets enabled at 10:20 am and requests the batch factory for being added to a batch. Base on its algorithm, the batch factory initializes a new batch instance *b11*, because the current *availBatch b10* is already *maxloaded*. The batch factory sets *b11* as *availBatch* and adds *i31* to its waiting queue. As time proceeds, new enabled instances (*i37*, *i40*, *i48*) are added to *b11*. For each, the activation rule of the batch instance is triggered, because the number of waiting instances is increased, but it does not evaluate to true. With activity instance *i51*, the waiting queue length is equal to the threshold of five. The activation rule evaluates its condition to true and the action to enable the batch instances is executed. With *b11* being in the *ready* state (at 3:14 pm), it is offered to the task executor which are all employees having the role of *claim inspector*.

A new enabled instance *i59* arrives which is assigned to *b11* by the batch factory, because *b11* is still in the *ready* state and so still the *availBatch*. At 3:40 pm, a claim inspector accepts the batch and starts it. The batch instance *b11* changes into the *running* state. Due to the choice of sequential batch execution, *b11* triggers the state change of the first assigned activity instance *i31* and provides it over the process engine to the claim inspector. When *i31* is terminated, the batch instances starts the next instance. For the newly enabled activity instance *i64*, a new batch instance *b12* is created, because *b11* is *running* already. With completion of *i59* at 4:23 pm – the last one in the waiting queue of *b11* –, the batch instance changes into *terminated*.

**Table 2.** Batch instance log: Each row presents moment of state change by a batch instance

| id | state | time |
|---|---|---|
| ... | ... | ... |
| b10 | *maxloaded* | 13/03/11 10:18 am |
| b10 | *running* | 13/03/11 10:50 am |
| b10 | *terminated* | 13/03/11 11:45 am |
| b11 | *init* | 13/03/11 10:20 am |
| b11 | *ready* | 13/03/11 03:14 pm |
| b11 | *running* | 13/03/11 03:40 pm |
| b11 | *terminated* | 13/03/11 04:23 pm |
| b12 | *init* | 13/03/11 04:13 pm |
| b12 | *ready* | 14/03/12 04:13 pm |
| b12 | *running* | 14/03/12 04:17 pm |
| b12 | *terminated* | 14/03/12 05:05 pm |
| ... | ... | ... |

On the next day, only few enabled activity instances for manual claim check arrive. The activation rule is triggered, because instances are not added to the waiting queue of *b12* for a longer time period. At 4:13 pm, the condition that the lifetime of *b12* is equal or greater than the maximum waiting time of one day is fulfilled and the enablement of

*b12* is conducted. Few minutes later at 4:17 pm, the batch is accepted by a claim inspector. Again, the activity instances are started one after another and the batch instance *b12* terminates at 4:36 pm with the end of its last activity instance *i83*.

The batch activity *Produce voucher card* in the process of Fig. 7 is an example for parallel batch execution. As we already discussed parallel execution in detail in Section 3.2, we will omit the discussion here due to space requirements.

## 4 Related Work

In an early work, Barthelmess and Wainer distinguish activities in work-case based activities acting exclusively on particular business cases and batch activities for which a set of cases have to be bought together [2]. They argue that batch activities are needed to improve process performance or to compare business cases. Since than, only few attempts to integrate batch activities into process modeling and execution were proposed. In Table 3, we show them with respect to their coverage of the requirements of Section 2.

**Table 3.** Evaluation of related work

| | Aalst et al. [13] | Sadiq et al. [12] | Liu et al. [4] | Mangler et al. [6] | This |
|---|---|---|---|---|---|
| R1: Different execution semantics | + | + | + | + | + |
| R2: Activation rule | - | - | o | + | + |
| R3: Homog. vs. heterog. requirements | - | - | + | + | - |
| R4: Single vs. multiple server(s) | + | + | - | o | + |
| R5: Parallel vs. sequential enactment | - | - | - | - | + |
| R6: Maximum capacity | + | + | + | - | + |

fully satisfied (+), partially satisfied (o), not satisfied (-)

One of them is the *proclet* framework by van Aalst et al. [13]. It defines a process as a set of interacting proclets representing process fragments via channels. A batch activity can be realized by a proclet which receives a predefined number of messages by instances of a cooperating proclet, enacts the batch task and sends back messages to the corresponding instances (R1). The other proclet may cover all single-case activities of the process. A sequential batch execution is not possible. Proclets allow different resource allocations (R4), but the size of a batch has to be explicitly given at design time (R6) leading to a inflexible batch execution: The number of cases for comparison cannot be defined dynamically and waiting for specific number of instances can result in decreased process performance. In this work, we provide a flexible batch execution approach by giving the possibility to select an activation rule for a batch activity.

Sadiq et al. [12] propose to establish *compound activities* in workflow systems with a grouping- and ungrouping-function generating one activity instance based on several ones and splitting it after task execution (R1). This activity instance could be provided based on different resource patterns to task executors (R4). The grouping-function can be either auto-invoked with a predefined number of required instances (R6), which has similar drawbacks as proclets, or user-invoked, creating a batch with user-selected instances. The user-invocation means a manual batch organization where rules are not explicitly defined and errors can occur. In contrast, our approach offers the possibility to explicitly define the batch execution rules with automatic enforcement of those.

Also, Liu et al. [4] want to integrate a new type of activity for batch execution into workflow systems which is called batch processing activity (BPA). As the BPA is limited to the threshold rule (R2), the process designer defines a threshold and the maximal capacity for a BPA as well as a grouping characteristic. Based on this characteristic, a grouping and selection algorithm (GSA) groups activity instances arrived in the central waiting queue of the BPA and selects a group to submit it to a server (R3). The functionality of the GSA is not further discussed in this work; a proposal can be found in a later work [5]. The authors limit their approach such that each BPA has only one server available. Liu et al. [4] establish a scheduling algorithm for the GSA which observes the state of BPA's waiting queue as well as server and initializes it when the server is idle and the threshold rule is fulfilled. With this algorithm, only one specific case of a direct allocation to one resource is covered. In our work, we allow that several batches can run in parallel so that all patterns of the resource perspective can still be used.

Mangler and Rinderle-Ma [6] provide an approach for a rule-based activity synchronization of instances of the same process as well as of instances of different processes. The synchronization is organized before or after specific rendezvous points. They propose that an external synchronization service can subscribe itself for being informed about the progress of certain process instances. This service can trigger to stop their execution as well as their continuation due to predefined ECA rules. According to their approach, a batch activity can be organized as follows: Before certain process instances start with a specified activity, the subscriber stops them and the references to them are saved in a buffer (R1). When a certain condition is met, the subscriber triggers the continuation of the respective process instances (R2). With several subscribers focusing on same rendezvous points, but different types of instances, also heterogeneous demands by activity instances can be served (R3). However, the authors do not discuss, how this batch of instances is provided to the task executor (R4). Furthermore, advanced technical knowledge is requested for the implementation of the synchronization service and its ECA rules. In this paper, we provided an approach with which a batch activity can be designed without any technical background and then automatically be executed.

## 5 Conclusion

In this work, we propose an approach to integrate the concept of batch activities into process modeling and execution. Therefore, we first defined several requirements based on the batch service problem in queuing research and the different execution semantics of a batch activity compared to a regular one. Next, we extended the process meta model so that a batch model can be optionally associated to an activity making it a batch activity. With its different parameters, a process designer can configure the batch execution by selecting an activation rule and defining the maximum batch size as well as the way of execution (i.e., parallel vs. sequential). With these configuration parameters, the rules for a batch activity are explicitly documented which facilitates the communication with process stakeholders. The batch model describes the behavior for a set of batch instances where each batch instance manages one batch execution in order to allow that several batches can run in parallel in case of multiple available resources. We presented an architecture and execution semantics to show how the parallel batch execution can be organized and implemented in a process engine. The applicability of our approach

was illustrated based on a case study. The study demonstrated that an automatic execution of batch activities by a process engine removes effort from process participants to organize it manually. Furthermore, it improves monitoring and analysis of a batch activity which results can be also used for enhancing batch configurations.

Our approach covers all defined requirements except that activity instances can be heterogeneous in their demands and may have to be grouped into different types of batches. This may be solved by developing an additional configuration parameter for the batch model to express grouping characteristics. Furthermore, our approach assumes that only activity instances of the same process can be grouped into a batch. The BPMN concept of *call activities* can provide a possibility for bringing activity instances of different processes together. When the batch activities are part of the process model, the process designer has an increased responsibility for their correct configuration. In order to support that, further validation and verification techniques should be developed which take into account batch activities. In future work, we want to address these limitations.

## References


1. Bailey, N.: On queueing processes with bulk service. Journal of the Royal Statistical Society. Series B (Methodological), 80–87 (1954)
2. Barthelmess, P., Wainer, J.: Workflow systems: A few definitions and a few suggestions. In: Organizational Computing Systems, pp. 138–147. ACM (1995)
3. Laliwala, Z., Khosla, R., Majumdar, P., Chaudhary, S.: Semantic and rules based event-driven dynamic web services composition for automation of business processes. In: SCW, pp. 175–182. IEEE (2006)
4. Liu, J., Hu, J.: Dynamic batch processing in workflows: Model and implementation. Future Generation Computer Systems 23(3), 338–347 (2007)
5. Liu, J., Wen, Y., Li, T., Zhang, X.: A data-operation model based on partial vector space for batch processing in workflow. Concurrency and Computation 23(16), 1936–1950 (2011)
6. Mangler, J., Rinderle-Ma, S.: Rule-based synchronization of process activities. In: CEC, pp. 121–128. IEEE (2011)
7. Mathirajan, M., Sivakumar, A.I.: A literature review, classification and simple meta-analysis on scheduling of batch processors in semiconductor. IJAMT 29(9-10), 990–1001 (2006)
8. Medhi, J.: Stochastic Models in Queueing Theory. Academic Press (2002)
9. Neuts, M.F.: A general class of bulk queues with poisson input. The Annals of Mathematical Statistics 38(3), 759–770 (1967)
10. Papadaki, K.P., Powell, W.B.: Exploiting structure in adaptive dynamic programming algorithms for a stochastic batch service problem. EJOR 142(1), 108–127 (2002)
11. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M., Edmond, D.: Workflow resource patterns: Identification, representation and tool support. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 216–232. Springer, Heidelberg (2005)
12. Sadiq, S., Orlowska, M., Sadiq, W., Schulz, K.: When workflows will not deliver: The case of contradicting work practice. In: BIS, vol. 1, pp. 69–84. Witold Abramowicz (2005)
13. van der Aalst, W.M.P., Barthelmess, P., Ellis, C.A., Wainer, J.: Proclets: A framework for lightweight interacting workflow processes. IJCIS 10(4), 443–481 (2001)
14. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)
15. Weiss, H.J., Pliska, S.R.: Optimal control of some markov processes with applications to batch queueing and continuous review inventory systems. CMS-EMS, Discuss. Paper (214) (1976)
16. Weske, M.: Business Process Management: Concepts, Languages, Architectures, 2nd edn. Springer (2012)