Enabling Batch Processing in BPMN Processes

Luise Pufahl and Mathias Weske

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

Abstract. Business process automation improves organizations' efficiency to perform work. Single executions of process models, called process instances, are usually executed independently in business process management systems (BPMS). In practice, we can observe examples in which a synchronized execution of groups of instances for certain activities, called batch processing, can lead to an improved performance. Batch regions is a concept to allow batch processing in business processes. This demo presents the implementation of the batch region concept in an open-source BPMN engine. It shows how, with a few extensions only, batch processing is enabled and how the consolidated view of several work items in one user form, leads to an improved work efficiency for users.

Keywords: Batch Processing, Process Enactment, BPMN

1 Introduction

Business process automation improves organizations' efficiency to perform work. Therefore, processes are often captured in process models, typically in BPMN (Business Process Modeling and Notation) [2] diagrams, the industry standard. These are then executed by a BPMS (e.g., Bizagi [1], Camunda [3], Signavio Workflow [8]). An executing instance of a process model is called process instance. Multiple process instances might run simultaneously in a BPMS. However, as Russell et al. [7] observe, "each of these is assumed to have an independent existence and they typically execute without reference to each other."

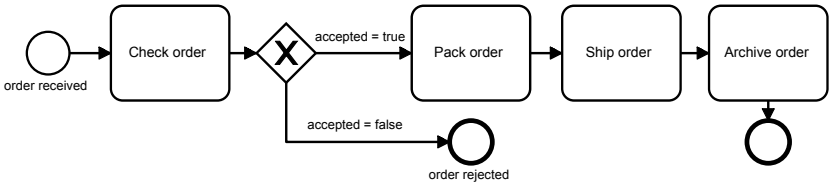

Fig. 1. Simplified online retailer process shown in a BPMN diagram.

In practice, we can observe different cases where the synchronized execution of several instances is beneficial and can improve process performance. For example, Fig. 1 shows the simplified version of an online retailer process as BPMN diagram in which

customer orders are handled. Often, online retailers do not charge any transport cost with the effect that customers place multiple orders in relatively short time frames. In such situation, several orders of the same customer could be packed and shipped together to save shipment costs. This approach, called batch processing, allows business processes which usually act on a single item, to bundle the execution of a group of process instances for certain activities in order to improve performance. Other examples which benefit from batch processing, we can observe in health care, e.g. collecting a set of blood samples for delivery to the laboratory, in insurance and finance, e.g. consolidating several letters to one customer and send them as one mail, and in administration, e.g. collecting several invoices for their approval. Usually these examples are already executed in a batch, but manually with the risk that batch processing rules might not be clear for everyone or might be ignored. Recent research approaches [4–6] provide means to integrate batch processing in business processes models and its automatic execution. In contrast to the others, the batch region concept in [6] allows an individual batch configuration based on which instances are batched with similar data characteristics over a number of activities. In this demo, we want to show an implementation of the batch region concept for BPMN processes in the open-source BPM platform Camunda [3]. Next, Section 2 introduce the batch region concept for BPMN processes; Section 3 presents the implementation details. We conclude in Section 4.

2 Batch Region Concept in BPMN - Design and Execution

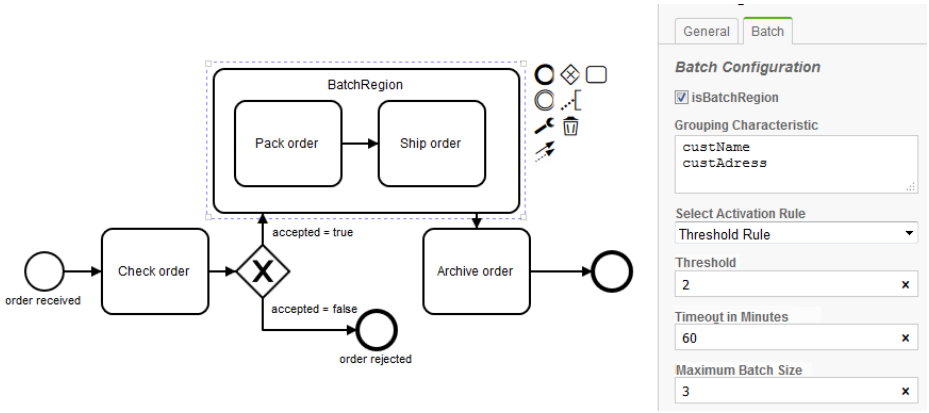

Fig. 2. Online retailer process with a batch region to save shipping costs.

A batch region in a BPMN process diagram is a special type of sub-process enabling batch processing for its activities. Fig. 2 shows the online retailer process with a batch region surrounding the activities *Pack order* and *Ship order*. With its configuration parameters (visualized in the right panel of Fig. 2), the process designer is able to specify the conditions for the batch execution. Those are (1) a grouping characteristic to cluster process instances to be processed in one batch based on data attributes (e.g., the `custName` and `custAdress` to identify similar order instances in our retailer example), (2) an activation rule to determine when a batch is activated while balancing

between waiting time and costs savings (e.g., when at least two similar order instances are available or a timeout of one hour is reached specified in a threshold rule), and (3) the maximum batch size indicating the maximum number of instances in a batch (e.g., at maximum three orders fit in one parcel). In a batch region, XOR gateways are not allowed because decisions are usually taken on individual items, but not on a item group. Further, we require that only one start event in a batch region so that the activation of a batch cluster is uniquely determined.

Each execution of the batch region is represented by a batch cluster. It collects a number of sub-process instances for batch execution whereby these are assigned based on their data values. For example, only instances with $\texttt{custName} = John$ and $\texttt{custAddress} = Madrid$ are assigned to the cluster `John_Madrid`. A batch cluster has different life cycle states shown in Fig. 3. When the first batch region activity is enabled, a sub-process instance gets assigned to an existing or new batch cluster. It is checked whether a cluster is available with the same data characteristics that is in the *init* or *ready* state. If not, a new cluster is created in the initial state *init*. If the activation rule is fulfilled, it transitions into the *ready* state.In this state, a *batch work item* including data of all instances is provided to the task performer. In the *ready* state, further instances can be still assigned to it to achieve an optimal cluster utilization. In this state, the cluster can transition into the *maxloaded* state, if its maximum batch size is reached, or into the *running* state, if the task performer begins the work item execution. In these two states, no further instances can be added. The cluster stays in the *running* state for all further activities in the sub-process. With termination of the last batch work item, it changes into the *terminated* state. Now, the instances are again independent from each other. After this short introduction into batch region design and its execution semantics, the next section describes our extensions of an open-source BPMN engine to enable batch processing.

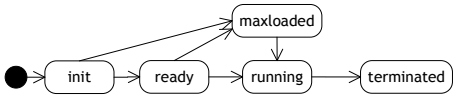

Fig. 3. Life cycle of a batch cluster

3 Tool Architecture and Implementation

For the implementation[2] of the presented batch region concept, Camunda [3] was selected, a Java-based, open source engine specifically tailored for a subset of BPMN.

First, we integrated batch region concept in the BPMN XML specification by utilizing *extension elements*, which the BPMN specification [2] explicitly supports to add new attributes and properties to existing constructs. The extension was added to the sub-process element describing the batch region configuration. Based on it, the Camunda modeler *bpmn.io* was adapted to enable a quick design of batch regions. Specifically, the sub-process element was extended as shown in Fig. 1.

The Camunda engine was extended by only four additional classes: The *batch region* class stores the configurations of each identified batch region and manages the assignment of process instances to a batch cluster. The *batch cluster* class governs the batch execution for its assigned group of process instances. The *batch behavior* class

[2] A link to repository of the implementation and a screen cast are available at `http://bpt.hpi.uni-potsdam.de/Public/BatchProcessing`.

includes the internal behavior of the activities in a batch region sub-process. In the Camunda engine, every process activity gets a task behavior (e.g., user task, service task) assigned, describing the internal activity behavior. In order to reuse these behaviors and limit the engine extension, the *batch behavior* inherits the normal task behavior and has additional methods for the batch execution defined in an interface. This is driven by the idea that one of the cluster instances leads the batch execution by first merging the data of all cluster instances and then executing the usual task behavior. Currently, this is implemented only for user tasks, but can easily applied to other task types. Further, a *batch timer* job class was added to enable the time-out defined in the threshold rule. Additionally, the BPMN parser was adapted to read batch regions' specifications and to assign every batch region activity its batch behavior.

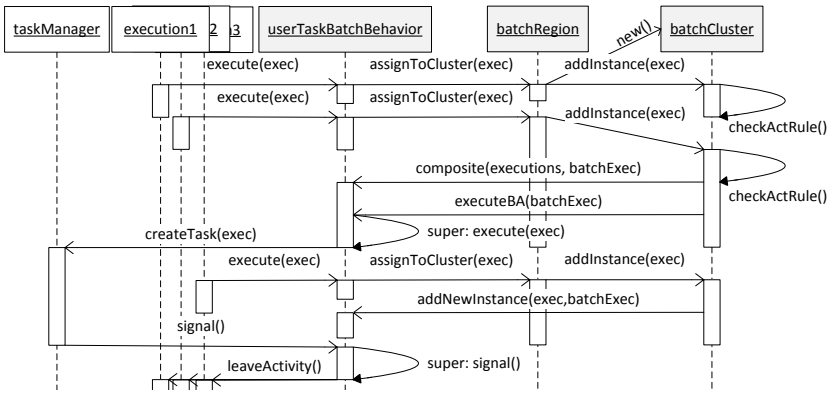

Fig. 4. Sequence diagram visualizing the interaction between added classes *batch behavior*, *batch region*, and *batch cluster* to the Camunda engine.

For the example of the *UserTaskBatchBehavior*, the interaction of the batch behavior with the batch region and the batch cluster class is shown in Fig. 4. As soon as an *Execution* object representing a process instance enables an activity with a batch behavior, it is added by the batch region to a cluster. If no batch cluster is currently available, it is first created and then the `add()`-method of the cluster is called in which also the activation rule is checked. Currently, our implementation supports the threshold rule. With its fulfillment, the cluster calls the `composite()`-method of the batch behavior merging the data of all instances. In case of the user task, a JSON variable with all instance data is created. This can be later reused during the user form design. Then, the `executeBA()`-method is called in which the batch behavior calls the `execute()`-method of its super class. Now, the normal user task behavior is executed in which a work item for the task performer is created. Fig. 5 shows the batch work item for the *Ship order* activity of the retailer example. We have used the JSON variable to visualize all orders in a table. The task performer can easily inspect all orders and has to enter the value for the logistics provider only once, as it is valid for all orders. Instead of three work items, the task performer has to process only one, leading to time and cost savings.

Our implementation provides also the feature to add new instances, while the first batch region activity is not completed, yet. The corresponding `addNewInstances()`-method simply adapts the JSON variable. With completion of a batch work item, the

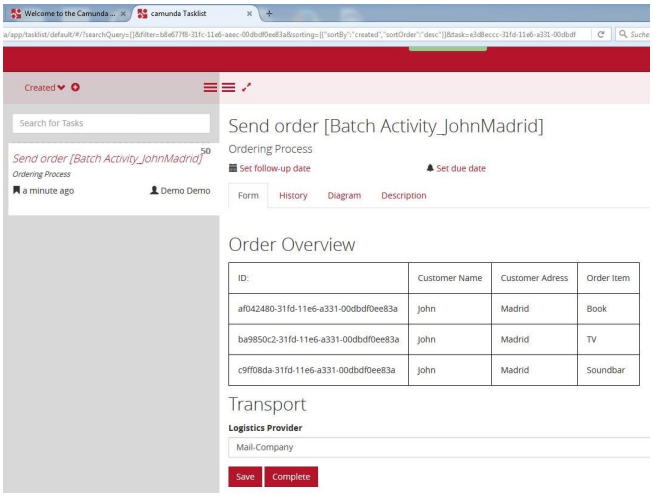

Fig. 5. Batch work item for the *Ship order* activity of the online retailer example.

task manager calls the `signal()`-method of the batch behavior distributing new added data (e.g., the logistics provider in Fig. 5) to all other cluster instances. Finally, with the last batch work item, also the batch cluster is terminated. The implementation shows that only small extensions on a BPMS are necessary, which also have no significant influence on the engine performance, to enable batch processing.

4 Conclusion

Real-world examples show the necessity of batch processing in business processes. In this paper, the batch region concept was implemented in an existing BPMS to realize batch processing for BPMN processes. Only small extensions are necessary to adapt the BPMN engine having also no significant impact on the engine performance. The demo shows that batch processing has the advantage that task performers can handle several items consolidated in one user form improving their work efficiency.

References

1. Bizagi: Bizagi Business Platform. `http://www.bizagi.com/`
2. OMG: Business Process Model and Notation (BPMN), Version 2.0 (2011)
3. Camunda: Camunda open-source BPM Platform. `https://www.camunda.org/`
4. Liu, J., & Hu, J.: Dynamic batch processing in workflows: Model and implementation. In: Future Generation Computer Systems, 23(3), pp. 338-347. Elsevier (2007).
5. Natschläger, C., Bögl, A., Geist, V., & Biró, M.: Optimizing Resource Utilization by Combining Activities Across Process Instances. In: Systems, Software and Services Process Improvement, pp. 155-167. Springer (2015).
6. Pufahl, L., Meyer, A., and Weske, M.: Batch regions: process instance synchronization based on data. In: EDOC, 2014 IEEE 18th International, pp. 150-159. IEEE, (2014).

7. Russell, N., Ter Hofstede, A. H., Edmond, D., & van der Aalst, W. M.: Workflow data patterns: Identification, representation and tool support. In: Conceptual Modeling-ER 2005. LNCS, vol. 3716, pp. 353-368. Springer (2005).
8. Signavio Workflow `https://workflow.signavio.com/`