Limitations and Potentials of modern BPM Systems for High-Performance Shadow Processing in Business Processes of Digital Companies

Martin Schmollinger[1], Jürgen Krauß[2]

Abstract: Digital companies need information systems to implement their business processes end-to-end. BPM systems are promising candidates for that, because they are highly adaptable due to their business process model-driven operation mode. End-to-end processes contain different types of sub-processes that are either procedural, data-driven or business rule-based. Modern BPM systems support modeling notations for all these types of sub-processes. Moreover, end-to-end processes contain parts of shadow processing, so consequently, they must be supported in a performant way, too. BPMN seems to be the adequate notation for modeling these parts due to its procedural nature. Further, BPMN provides several elements that enable the modeling of parallel executions which are very interesting for accelerating shadow processing parts of the process. The present paper will observe the limitations and potentials of BPM systems for a high-performance execution of BPMN models representing shadow processing parts of a business process.

Keywords: Business Process Management, Parallel Computing, Process Engine, BPMS, BPMN

1 Introduction

Successful digital companies rely on digital processes that can be adapted very fast to new business opportunities and models. In order to tackle these digital transformations, business processes need to be supported end-to-end by information systems. Modern business process management systems (BPMSs) [Du13], [FR14], [We12], [Ka95] claim to be a platform for this challenge. BPMSs enable the development of model-driven process applications. The core of these applications are business process models that are executed by process engines. By that, they provide a much higher degree of transparency for developers and for business people than conventionally developed applications. On the one hand, the approach improves business monitoring capabilities needed for process optimization and on the other hand enables shorter development cycles.

Current notations of the OMG [Ob16] for process modeling target different aspects of end-to-end processes. While BPMN (Business Process Model and Notation) [Ob11] is preferable for modeling parts of the process that can be described best in a procedural way, CMMN (Case Management Model and Notation) [Ob14] is more suitable for knowledge-

[1] Reutlingen University, Herman Hollerith Zentrum für Services Computing, Alteburgstr. 150, 72762 Reutlingen, bpm@reutlingen-university.de
[2] Reutlingen University, Herman Hollerith Zentrum für Services Computing, Alteburgstr. 150, 72762 Reutlingen, juergen.krauss@reutlingen-university.de

intense and data-centric parts. Further, DMN (Decision Model and Notation) [Ob15] standardizes models for business rules in processes. Obviously, BPMSs that support all these different standards in combination have the most potential for implementing end-to-end processes in digital companies, successfully. Looking at end-to-end processes, we often encounter parts of shadow processing, i.e. the processing of business transactions without any human interaction. Consequently, to benefit in the same way from the advantages of BPMSs as the other parts of the processes, these parts should be modeled by the developer, as well, and executed by the process engine in an efficient way.

Due to its procedural nature, BPMN is suitable for modeling shadow processing parts of a business process. Further, BPMN includes notation elements for modeling parallel executions in business processes that may accelerate the shadow processing parts. High-performance shadow processing assumes real parallel execution by the process engine. In current BPMSs, the process engine is executed in one thread and may spawn multiple threads additionally for parallel execution.

In fact, OMG's BPMN specification defines an execution sematic for all BPMN elements, but there is still a high degree of freedom for implementers concerning real parallel execution. As a consequence, real parallel execution is not part of the model and therefore not transferable from one BPMS to the other. Further, although several BPMSs implement the BPMN specification to the letter, their capabilities concerning real parallel execution may be totally different and have to be implemented for each system in a proprietary way.

In the present paper, we work out this observation for Camunda BPM [Ca16], a popular open source BPMS. It is currently the only system on the market supporting all three notations from the OMG and therefore an interesting and, as mentioned before, a high potential candidate as a platform for end-to-end processes in digital companies. We will show the system's out-of-the-box capabilities for parallel execution and will judge its performance in comparison to an application implementing the parallel execution directly using threads. In fact, this paper does not claim to give a general evaluation for all BPMSs on this huge market [Ga15]. However, Camunda's process engine is based on a popular approach called Process Virtual Machine (PVM) [Sm05] [Cr06], hence, we see the system as a good representative for a wide range of current BPMSs. Additionally, Camunda BPM itself claims to be very progressive in this area.

2 BPMN 2.0 Specification and Implementation

BPMN is a modeling notation for business processes models maintained by the Object Management Group (OMG) [Ob11], [Ob16]. In order to start the observation of parallelism in BPMN, it is not necessary to regard every element from the specification. Hence, we limit the amount of elements to tasks, parallel gateways, sequence flows and tokens. Tokens are no graphical elements, but a theoretical concept. They are markers which traverse the elements of the model on their way and provide snapshots of execution state. Thereby, they indicate the condition and progress of a process and create a common base for discussions about the behavior of a process. A task is an atomic activity on a fine-grained level of detail. An example is “write email” or “send email”. Without further knowledge about the business process, it seems not appropriated to break the tasks down to a more fine-grained level. Simultaneously, it could be functionally reasonable to not define the task as “write and send email”, because the part of writing could be done by a human worker while sending the email might be automatic task. Furthermore, the process could be monitored and is identifiable if the user needs time to write the email or if the machine has issues to send it [LR97].

The BPMN specification defines different constructs to achieve parallelism. Initially, we examine the parallel gateway to execute tasks simultaneously. According to the BPMN specification real parallel execution is optional, hence, the obtained degree of parallelism of the execution depends on the respective BPMS. Functionally, the parallel gateway has two purposes. When branching, the gateway is receiving a token. This token will be cloned and one is routed to each outgoing sequence flow. When merging, the parallel gateway is joining the previously created tokens arriving at the incoming sequence flows. The gateway waits for all incoming tokens on each incident sequence flow, merges the tokens and again, clones the token for each outgoing sequence flow.

After the review of the BPMN 2.0 specification, we take a further look into the implementation of Camunda BPM to determine, if the theoretical foundation is properly implemented. Information about the engine can be found in the User Guide for the current version (7.4) [Ca15a] and the API [Ca15b]. Initially, we use Service Tasks, which invoke services by calling Java code. According to the Camunda User Guide [Ca15a], the parallel gateway is “the most straightforward gateway to introduce concurrency in a process model”. The specifications from Camunda BPM and OMG about branching and merging are identical.

While, according to the OMG, tokens are just a theoretical concept, Camunda BPM gave them technical and functional values. As explained above, they enable users to monitor process instances during execution. A process instance can hold multiple tokens for example because of the use of parallel gateways. Hence, the set of tokens of a process instance defines the current state of its execution.

Summarized, Camunda BPM seems to be able to execute parallel sequence flows as specified in BPMN 2.0. We try to verify this with a simple experiment.

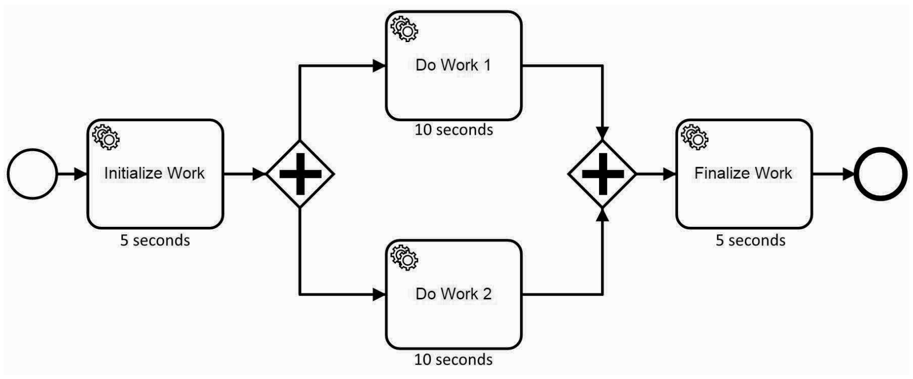

Fig. 1: Experiment for parallel execution

As depicted in Fig. 1, the experiment is composed of a Service Task “Initialize Work” (five seconds duration) followed by a branching parallel gateway with two outgoing sequence flows. One sequence flow arrives at Service Task “Do Work 1” the other at “Do Work 2” (both ten seconds duration) followed by a merging parallel gateway. The process ends with Service Task “Finalize Work” (five seconds duration). The parallel Service Tasks invoke Java code which works for ten seconds. The assumption is an overall running time for the execution of about 20 seconds. Surprisingly, the execution time is about 30 seconds. Using the monitoring capabilities of Camunda BPM shows that the tasks are executed sequentially which explains the running time.

Summarized, the parallel gateway in Camunda BPM does not support parallelism by default. Since the OMG alllows parallelism, Camunda’s implementation is very passive in a sense, so far. Next, we take a deeper look in the mechanisms of the process engine to find an approach to achieve parallelism.

3 Advanced Configurations

The previous experiment showed that Camunda BPM is not able to execute tasks in parallel by default. However, this just takes effect with this simple approach. Camunda BPM offers various settings to modify the procedure of process execution. Before the effects and by association the results of the paper can be discussed, this chapter will explain the settings and mechanisms of the engine. First of all, we will understand how the engine executes the example. Then we will move on and discuss different settings which alter the execution.

3.1 Transactional Processing

Initially, we have to take a closer look at the execution of process instances. During the execution of a process instance, the engine processes transactions. Transactions are based on the ACID paradigm by Haerder and Reuter [HR83]. One process instance is mapped on a minimum of one transaction. While we move on through this chapter, we get a more detailed view on the mechanisms of transactions in the system. By default, the engine executes every process instance and by that, every transaction in a single thread. The thread executes every sequence flow sequentially until the execution of the current process instance is finished. The experiment in chapter 2 (Fig. 1) can be described by following steps: the main thread starts at the start event. It traverses the sequence flow and executes Service Task "Initialize Work". After that, the parallel gateway forks into two sequence flows. The main thread first traverses the sequence flow and executes Service Task "Do Work 1". After a traversal to the parallel gateway it wants to join and has to wait for the other sequence flow. The main thread jumps back, traverses the sequence flow and executes Service Task "Do Work 2". Now the parallel gateway is able to join both incoming sequence flows. Afterwards, the thread executes Service Task "Finalize Work" and finishes the process at the end event. This procedure perfectly explains the behavior from chapter 2.

3.2 Asynchronous Continuation

Camunda provides a mechanism called asynchronous continuation. It can be activated before or after gateways and tasks and enables two functions. First, the asynchronous continuation alters transactions. Whenever the engine reaches a point of asynchronous continuation the current transaction will be persisted to a database. In this way it serves as a manually usable mechanism to set boundaries of transactions and to split them. This leads to the effect, that in case of repetitions after exceptions, the state of the processed instance is saved up to the last point of asynchronous continuation. That way a developer can determine the behavior in case of exceptions and roll back situations.

Second, the even more important function of asynchronous continuation for parallel execution is the introduction of the job executor. Whenever the engine arrives at a point of asynchronous continuation the following transaction is signed to a list as a job. The job executor assigns this job to a thread pool and executes them. However, for consistency reasons the job executor will not execute jobs from the same process instance in parallel by default. Jobs from the same process instance will be assigned to the same thread from the pool. As a result the job executor avoids optimistic locking exceptions after parallel processing of tasks. The engine addresses issues with consistency and persistence during synchronization of parallel sequence flows with optimistic locking based on Kung [Ku81].

Optimistic locking assumes that parallel read access on shared memory is generally unproblematic and therefore does not have to get locked. So, generally all tasks can read global transaction data and get a local copy. Parallel tasks could now be processed in parallel and alter this copy. However, at some point, e.g. at a joining parallel gateway, the local copies need to get synchronized. Each task has to check that no other task has altered the global data since the last read operation. If the data was not changed, the local data can be written and the transaction is committed. If it has been already changed (Optimistic Locking Exception) the transaction has to be rolled back and is executed from the beginning again. Summarized, the asynchronous continuation are a step towards parallelism. While the save points allow us to be more specific on transaction boundaries, the introduction of the job executor with the thread pool is the mechanism we need to utilize threads beside the main thread.

3.3 Non-exclusive Tasks

As mentioned above, the job executor executes jobs from the same process instance in sequential order by default, to avoid optimistic locking exceptions. However Camunda BPM enables the developer to set a tasks as "non-exclusive". This leads to two differences in execution. The job executor will not verify the dependence on a non-exclusive task to another one. Every non-exclusive task is assigned to a single thread. Furthermore, optimistic locking exceptions are not avoided which leads to repetition of execution as described in the previous chapter.

4 Towards Real Parallelism

In chapter 2, we introduced a basic application with the attempt to achieve parallelism. Now we know about the mechanisms asynchronous continuation and non-exclusive tasks which sounds promising. We can improve the situation by adding a point of asynchronous continuation before and after each service parallel tasks. Further we mark both as non-exclusive. These settings can be done by changing the attributes of the Service Tasks using the Camunda Modeler. The graphical representation of the model stays the same. Fig. 2 shows the changes on the model.

Due to the previous changes, the kind of processing has been fundamentally changed. Instead of a single thread the job executor processes Service Task "Do Work 1" and Service Task "Do Work 2" in parallel. The first Service Task that finishes commits, the second one can commit too, because there is nothing to do between the last safe point and the parallel gateway that could be repeated due to an optimistic locking exception. Although the execution is in parallel now, we observed further limitations which will be discussed in the next sections.

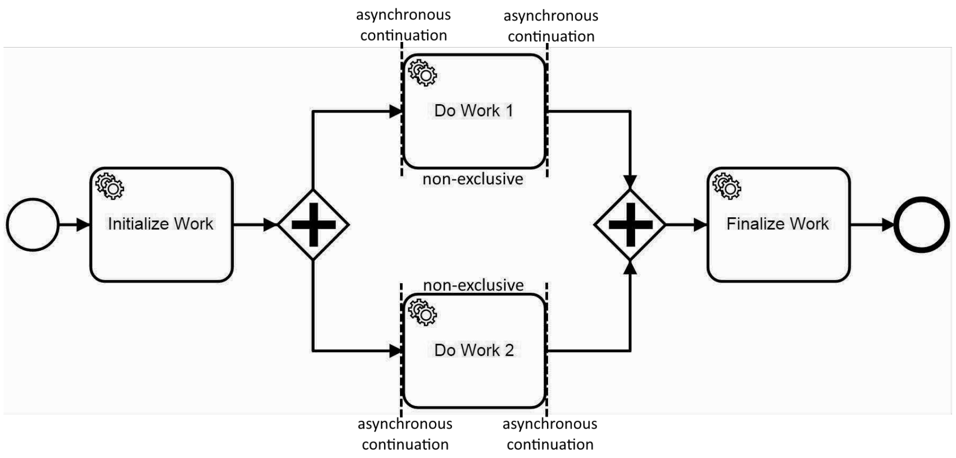
asynchronous continuation
asynchronous continuation
Do Work 1
non-exclusive
Initialize Work
Finalize Work
non-exclusive
Do Work 2
asynchronous continuation
asynchronous continuation

Fig. 2: Parallel sequence flows with advanced configuration

4.1 Transaction Timeouts

As explained earlier in the paper, the process engine uses transactions to work off execution paths of a process instance, consistently. That applies to jobs executed by the job executor, too. In general, it is not recommended to allow long running transactions, since they lock resources and increase the risk for deadlocks. Hence, there is a timeout mechanism that rolls back the transaction after a certain time (e.g. five minutes). This is a contradiction to high-performance shadow processing, where we might have long running tasks. An easy (but rather dirty) solution to this problem is to increase the transaction timeout to a value that probably cannot be reached by the parallel jobs. This can be done by application server specific configurations on component or server level. Another solution to this problem is to substitute the Service Tasks by a combination of two BPMN Send and Receive Tasks (see Fig. 3).

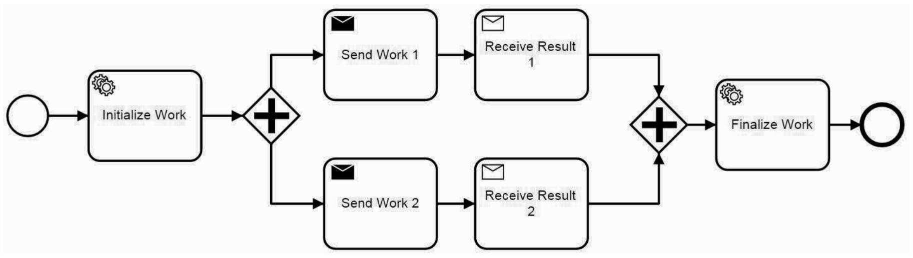

Fig. 3: Process model using Send and Receive tasks

Each Send Task produces a job for the job executor using the asynchronous continuation mechanism. Additionally, the job itself is called asynchronously (using `@Asynchronous` of the Enterprise JavaBeans API) so that the execution continuous directly after the method call and arrives at the Receive Task. The Receive Task is a wait state and will be reactivated by a message from the job running in the background. The job itself is executed outside the transaction management of the process engine since the corresponding Enterprise JavaBean method is configured in this way (annotation `@NOT_SUPPORTED`). As a consequence, we do not receive any transaction timeouts during the execution of the job.

Solving the problem with transaction timeouts in that way leads to another issue. The jobs for the job executor are stored in a data base. Before the job executor starts a job, it locks the job in the data base. After the job is finished, it is removed from the data base. If the job runs longer than 5 minutes (default), the lock is released and the job executor restarts the job again although the job is still running. If we have long running jobs, we need an additional workaround for this behavior. Either, we implement a retry recognition in the Service Task itself, or we set the timeout of the lock release to an unreachable value.

4.2 Scalable Models

So far, the degree of parallelism is integrated explicitly in the process models using a parallel gateway. Each parallel execution is modeled explicitly by a sequence flow and so is not scalable. In many situations, a scalable model would be much better because it is independent of the execution environment. E.g. a process engine could start the process instance with a parameter defining the number of threads to use for the executions in dependency on the available computing resources (e.g. cpu cores). For these situations BPMN provides multi-instance properties for tasks, sub-processes and even pools or swim lanes. In the following, we just consider multi-instance tasks with respect to space limitations. Adapting this to our simple example leads to the following model (see Fig. 4). In this case, it is possible to configure the multi-instance task in the way that multiple jobs are created in parallel using multi-instance asynchronous continuation before and after together with a non-exclusive execution. The number of tasks that will be created can be parameterized with a process variable that is passed while starting the process. Each job in the multi-instance task represents a separate execution. The multi-instance task itself is realized like a barrier synchronization. The process execution itself does not continue before all jobs are done.

The prevention of transaction timeouts for long-running jobs in this model is the same as described in section 4.1. Unfortunately, there is currently no feasible realization with send and receive tasks in multi-instance units so that we have to increase the transaction timeout and the lock lease timeout of the job executor in the configuration.

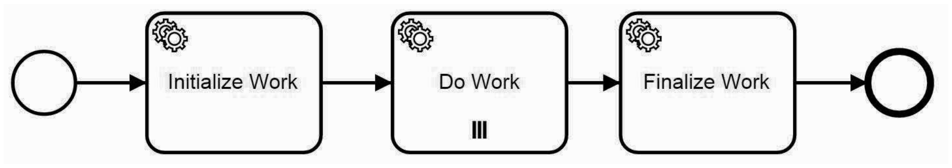

Fig. 4: A BPMN model containing a Service Task with a parallel multi-instance property.

5 Experiments

In this chapter, we want to judge the performance of the process application compared to a common Java application using threads. We implement a simple parallel algorithm that determines the maximum prime number in a list. Such problems are easy to parallelize since all threads can work independently on a disjunctive part of the data (data parallelism). Our parallel tasks divide the list of numbers into equal sized parts. Each task determines the maximum prime number for its own part. In the final step, a single task determines the overall maximum. A prerequisite for parallel processing is shared memory for the parallel executions. E.g. each tasks must have access to the same list of numbers and stores its intermediate result in the shared memory. In general, BPMSs provide the concept of process variables. A process variable is a data structure accessible during the execution of a process instance. At first sight, this concept could be used as shared memory in Camunda BPM. Unfortunately, modifying process variables in parallel executions leads to even more rollbacks due to optimistic locking exceptions, since the engine guarantees their consistency. Hence, we preferred another approach. Camunda BPM is implemented with Java and uses Java EE technologies (especially component technologies like Contexts and Dependency Injection and Enterprise JavaBeans). Process applications run inside a Java application server. Service Tasks reference methods of Java components and the latter are able to use dependency injection to reference each other. Therefore, it is possible to inject the same Java components in different Service Tasks. These components serve as shared memory for process instances. The developer is responsible to synchronize the concurrent access to these shared components using Java's concurrency mechanisms.

5.1 Experimental Setup

In our experiments, we want to compare the process applications using a multi-instance Service Task (see Fig. 4) with a Java application using a thread pool. The algorithm is realized with the process model in Fig. 4 in the way that Service Task "Initialize Work" makes the initialization, Service Task "Do Work" realizes the data parallel part and Service Task "Finalize Work" generates the final result. The experiments are made on a Lenovo T530 with 16 GB and an Intel Core i7 CPU providing 4 physical cores. Hyper-Threading Technology delivers two processing threads per physical core.

In order to keep the running times comparable, we prepared the list of numbers in the way that the determination of the largest prime is in $\Theta(n^2)$ with n being the size of the list of numbers.

5.2 Results and Interpretation

In our experiments, we used three different sizes for the list (n) and 4 different numbers of threads (p). The tables (see Tab. 1 and Tab. 2) depict the running times in milliseconds (ms) for each combination in the white columns. The grey columns show the gained speedup. Since the sequential part of our algorithm is insignificant, we calculate the speedup $S_p=T_1/T_p$, where T_1 is the running time with one thread and T_p for p threads. Tab. 1 and 2 summarize the results of our experiments.

p\n	5.000.000	S_p	10.000.000	S_p	20.000.000	S_p
1	21259	1,00	61162	1,00	174282	1,00
2	11826	1,80	33796	1,81	95964	1,82
4	11773	1,81	31157	1,96	95521	1,82
8	7605	2,80	17425	3,51	56971	3,06

Tab. 1: Average running times in ms and speedup for p threads and n numbers (Camunda BPM)

p\n	5.000.000	S_p	10.000.000	S_p	20.000.000	S_p
1	21094	1,00	59702	1,00	168817	1,00
2	10861	1,94	30755	1,94	87003	1,94
4	5805	3,63	16641	3,59	48402	3,49
8	4318	4,89	12403	4,81	34877	4,84

Tab. 2: Average running times in ms and speedup for p threads and n numbers (Java Application)

Looking at the tables, we realize that both variants behave very similar for p=1 and p=2. For p=2, the process engine has approximately 10% worse running times which can be explained with a constant overhead produced by the engine. The achieved speedups for both applications are very good, considering that the algorithm has a small sequential part for generating the final result with one thread (Amdahl's law).

With p>2, the speedup of the process application deteriorates in contrast to the Java application. The reason for this is that the thread pool of the Camunda's job executor uses only 2 threads for 4 jobs (p=4) and 5 threads for 8 jobs (p=8). As a consequence, not all threads are executed in parallel, which deteriorates the running times and speedups, explicably.

For p=8 both variants can improve the overall running times but cannot reach a satisfying speedup, because the management and synchronization of the threads produce an appreciable overhead. In our opinion, the experiments confirmed that the usage of a process engine has great potential for parallel computation. The worse running times could be improved by a better configuration or implementation of the job executor.

6 Conclusions

We have shown that it is possible to execute tasks in parallel. Obviously, it has to be noted that the results are valid for the Camunda BPM Engine. Since parallelism is not mandatory by the BPMN specification, it is probable that each BPMS has a different implementation. The range could be from a simple sequential to a real parallel execution. As a result, the process applications are not transferable between different BPMSs. In the worst case, the only artefact which can be transferred at is the BPMN model, while the whole configuration and implementation considering parallelism and performance has to be redeveloped. This raises the question whether the parallelism could not be introduced to the BPMN specification to achieve a standardization in the area.

On the road to a general concept for a standardization, a first step would be to create a comprehensive analysis about the most common BPMSs. Beside the fact that it is possible to execute tasks in parallel, the paper showed that the provided solutions are not yet sufficient and mature. The BPMS focuses on data consistency. Further the default settings which prevent parallelism and the difficulties with the optimistic locking and repetitions indicate that it is not in the nature of the process engine's concept. A developer seems to work against the engine and implements workarounds which are compromises between parallelism and consistency. Nevertheless, the provided settings and their advancements in the last years and the demand for parallel execution on the support boards of developer companies proof that the topic has an increasing impact.

Finally, the experiments showed that the performance of parallel executions in process applications are not that efficient as they could be. This can be explained by the overhead for transactional management and consistency as well as by insufficient implementations. Scientific workflow systems as Kepler [Ke16] or Vistrails [Vi16] claim better results under similar conditions, by data flow control [JHM04], [Lu06]. Downsides are less support and standardization as for business software, slow cycles of development and no extensive consistency. To improve the performance of shadow processing in digital end-to-end business processes, it would be worth surveying if combinations of different operation modes could be realized in current BPMS like Camunda BPM.

References

[Ca15b] Camunda: Camunda User Guide. https://docs.camunda.org/manual/7.4/user-guide/.

[Ca15a] Camunda: Camunda API. https://docs.camunda.org/javadoc/camunda-bpm-platform/7.4/.

[Ca16] Camunda: Camunda.org Website. https://camunda.org/.

[Cr06] Craig, I. D.: Virtual Machines. Springer, London, 2006.

[Du13] Dumas, M. et al.: Fundamentals of Business Process Management. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[FR14] Freund, J.; Rücker, B.: Real-life BPMN. CreateSpace, Charleston, SC, 2014.

[Ga15] Gartner: Market Guide to Business Process Management Platforms. 2015

[HR83] Haerder, T.; Reuter, A.: Principles of transaction-oriented database recovery. In ACM Computing Surveys, 1983, 15; pp. 287–317.

[JHM04] Johnston, W. M.; Hanna, J. R. P.; Millar, R. J.: Advances in dataflow programming languages. In ACM Computing Surveys, 2004, 36; pp. 1–34.

[Ke16] Kepler project: Kepler Website. https://kepler-project.org/.

[Kr95] Karagiannis, D.: BPMS. In ACM SIGOIS Bulletin, 1995, 16; pp. 10–13.

[Ku81] Kung, H. T.: On optimistic methods for concurrency control. In ACM Transactions on Database Systems, 1981, 6; pp. 213–226.

[LR97] Leymann, F.; Roller, D.: Workflow-based applications. In IBM Systems Journal, 1997, 36; pp. 102–123.

[Lu06] Ludäscher, B. et al.: Scientific workflow management and the Kepler system. In Concurrency and Computation: Practice and Experience, 2006, 18.

[Ob11] Object Managment Group: Business Process Model and Notation Version 2. http://www.omg.org/spec/BPMN/2.0/.

[Ob14] Object Managment Group: Case Management Model an Notation Version 1.0. http://www.omg.org/spec/CMMN/1.0/.

[Ob15] Object Managment Group: Decision Model and Notation Version 1.0. http://www.omg.org/spec/DMN/1.0/.

[Ob16] Object Managment Group: OMG Website, 2016.

[SN05] Smith, J. E.; Nair, R.: Virtual machines. Versatile platforms for systems and processes. Elsevier, Amsterdam, 2005.

[Vi16] VisTrails: VisTrails Website. http://www.vistrails.org/index.php/Main_Page.

[We12] Weske, M.: Business process management. Concepts, languages, architectures. Springer, Berlin, 2012.