Workflow Flexibility by Deviation by means of Constraint Satisfaction Problem Solving

Lisa Grumbach, Ralph Bergmann

University of Trier, Department of Business Information Systems II,
54286 Trier, Germany

Abstract. This paper introduces a novel approach for flexible workflow management by applying constraint satisfaction problem solving. This enables us to support workflow deviations at runtime, react to upcoming events or unpredictable circumstances, but still support the user through worklist suggestions. The developed workflow engine is completely based on declarative workflow representations, whereas procedural languages are used for workflow modeling.

1 Introduction

In small and medium-sized enterprises (SMEs) there is a strong demand for support concerning management of documents, business data, and processes as well as a need for supervision and control of all running and completed transactions [14]. Especially for employees who are unaware of common processes, a Process-Aware Information System (PAIS, [1]) would be of advantage, as they may profit from guidance concerning ideal workflow execution and task suggestion. Additionally, compliance concerning standards and guidelines would be facilitated, as benefit for the enterprises. However, the use of PAISs has not yet been broadly established in SMEs. A reason for this is that current PAISs control process execution by traditional workflow engines in which workflows are prescribed without providing any flexibility to deviate, if necessary [5, 15]. This is a particular problem in SMEs as their processes are only slightly standardized and weakly structured and may vary significantly from case to case [9, 17].

Artificial Intelligence (AI) is a key technology for various support strategies in Business Process Management (BPM), as it allows for automated decision making and thus, facilitates the users work. Allowing flexibility requires such intelligent technologies, as the user should only be guided executing a workflow and not be burdened with taking difficult decisions that could be automated. Declarative workflows are a means of implicitly offering flexibility but therefore require technologies from the field of AI for workflow control. DECLARE [2] is a tool suite for declarative workflow modeling and enactment. Declarative models consist of constraints which define undesired behaviour. Constraints are then transformed to finite-state automata which allow for reasoning about workflow states. A drawback of this approach is that this transformation process is inefficient for more than about 50 constraints [10] and thus runtime support for

In: M. Leyer (Ed.): Proceedings of the LWDA 2017 Workshops: KDML, FGWM, IR, and FGDB.
Rostock, Germany, 11.-13. September 2017, published at http://ceur-ws.org

changing circumstances is not provided. With an approach based on constraint satisfaction problem (CSP) solving we aim at achieving model changes at run-time efficiently, as constraints can simply be added or retracted without the need of a transformation. Furthermore with a combination of imperative and declarative paradigms the presented approach leads to an increased flexibility.

In this paper we present an approach for flexible workflow execution utilizing CSP solving to handle occurring deviations and to control worklist suggestions. First, the foundations concerning flexible workflow management are sketched, followed by the introduction of our new concept for combining imperative and declarative paradigms for flexible workflow execution. This approach is further described by algorithms which are based on CSP solving. The paper ends with a brief outlook on future work.

2 Foundations and Related Work

A workflow is "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules" [21]. A Workflow Management System (WfMS) supports the execution of workflows by a workflow engine that interprets the process definitions and interacts with a worklist handler, which is in charge of assigning work items to users.

Workflow Flexibility Traditional WfMS are rigid and do not allow any deviations from modeled workflows. Users feel restricted and such systems are rapidly considered as a burden. Thus, users bypass the systems, which is counterproductive for attaining the expected benefits [5]. Consequently, PAISs that allow a workflow to flexibly deviate are essential for efficiency in SMEs. Schonenberg et al. [16] distinguish between four kinds of workflow flexibility: Flexibility by Design, Change, Underspecification, and Deviation. The first three types require either complete knowledge about all possible workflow execution paths at design-time or demand a remodeling of the workflow at run-time. Hence, a flexible reaction to sudden changing circumstances during run-time is prevented, or actions are required to manually change the process instance, which is impossible for inexperienced users. "Flexibility by Deviation is the ability for a process instance to deviate at run-time from the execution path prescribed by the original process without altering its process model."[16]. Although this approach eliminates the previously mentioned disadvantages, little research exists on how to implement this approach. Only the system FLOWer [3] implements this idea to a limited extent by allowing the user to skip, undo, or redo a task or to insert a new task, but still the user has to intervene manually to obtain flexibility.

Workflow Modeling Paradigms Workflow modeling paradigms range from imperative (procedural) to declarative [6]. Imperatively modeled workflows explicitly specify all possible allowed execution paths, for example using a flow-based modeling language such as BPEL ([4]). Here, the control flow of tasks

as well as the related flow of data items is modeled, which results in a high complexity and a huge modeling effort. Declarative workflows, however, define forbidden behavior and states of the workflow. Imperative workflows only describe a subset of valid procedures, while declarative constructs describe specific undesired states, leading to the acceptance of every other state [11] and thus, implicitly providing flexibility concerning workflow execution.

Current declarative workflow approaches such as DECLARE [2], formally base on Linear Temporal Logic formulae representing constraints, which are further transformed into finite-state automata, for constraint validation, as workflow engine and for worklist handling. Though there is a differentiation between mandatory and optional constraints, and optional ones may be violated, a possibility to retract constraints is not specified and therefore no unforeseen situations can be handled flexibly. The concept of DCR graphs [8] is described as offering more flexibility, but also has no possibility to restore consistency after deviations. In later work Maggi et al. [10] developed an approach, Mobucon, based on colored automata, with the ability to detect deviations and in addition to support continuously through various strategies. A drawback of this approach is that strategies need to be determined beforehand, and cannot be changed during runtime, as the construction of a new automaton would take too long [10]. Algorithms developed by Westergaard [20] also solve this issue with efficient runtime modifications, e.g. models with up to 50 constraints are handled in seconds.

Our approach also aims at achieving efficient automated runtime modifications and thus requires the ability to react adequately to deviations, even to undesired situations. A main difference between related work and our approach is that our workflow control bases on the interpretation of incoming documents and their semantic information. The identification of semantic information has a significant impact on workflow control and can be easily defined as logical constraints. Furthermore constraints can be added or retracted ad-hoc, without the need of a time-consuming recompilation of the model. Therefore we regard the identification of executable tasks as constraint satisfaction problem.

3 Concept of the Workflow Engine

With the presented concept we aim to increase the acceptance of users, as the presented concept does not prescribe, but still guides, if needed. Additionally, transactions are logged for control and monitoring purposes. The implementation of the approach of Flexibility by Deviation as presented in this paper is embedded in the SEMAFLEX[1]-architecture [7], which semantically integrates flexible workflow management with knowledge-based document management and will be developed further in the SEMANAS project. An important characteristic is that the information about task enactment can either result from a user interaction, e.g. a manual selection of a task being performed, or due to upcoming documents, which are analyzed automatically and mapped to a certain task, whose

[1] SEMAFLEX is funded by Stiftung Rheinland-Pfalz für Innovation, grant no. 1158

enactment is derived subsequently. These logged task enactments construct the actually conducted workflow as a sequence of activities that have been performed. While the workflow engine proposes tasks which should be done next, the user is not forced to follow these suggestions. In principle, the user is able to do what s/he wants and in which order s/he wants. S/he can either follow the tasks in the worklist, suggesting the standard course of action, or do something else and upload documents created as a result of what s/he did. Through both, explicitly completing a task and uploading documents, the actual workflow is identified and recorded. Progress in turn affects the worklist handling, including detected deviations.

3.1 Combination of Imperative and Declarative Paradigms

For a suitable representation of the workflows concerning this concept, we explicitly differentiate between modeled workflow, *de jure workflow*, and executed workflow, *de facto workflow* [1]. The de facto workflow is an actually enacted instance derived from a de jure workflow. Thus, it stores the actually conducted transactions and thus might deviate from the de jure workflow.

In our approach the de jure workflow is modeled procedurally, as it is more intuitive and comprehensible than declaratively modeled workflows [12]. The workflow engine, however, is completely based on a declarative representation, as this paradigm implicitly offers flexibility concerning execution. To reach a maximum of flexibility, we transform the de jure workflow into declarative constraints, which are used to control the suggested execution order of tasks, but which are not regarded as mandatory and consequently might be violated. Hence, the de jure workflow is only considered as guidance, but deviations are tolerated. Nevertheless, some deviations are critical and should never occur, considering e.g. compliance or safety aspects. For this reason, additional mandatory constraints can be modeled manually, to explicitly specify invalid workflow states. Those are possibly connected with a severity specification, a warning message or even a proposed corrective measure, in case the constraint is violated. Such mandatory constraints can refer to the execution order of the tasks within a de jure workflow or they could be global constraints specifying order constraints across classes of workflows. Of course those mandatory constraints might actually be violated by the user, as the workflow engine never prescribes an activity and thus is not able to actively prevent violations. Nevertheless, the violation of constraints (including the mandatory ones) can be detected. Depending on the kind of constraint violation the workflow engine shall be able to react adequately. If a non-mandatory constraint is violated, the deviation is not considered as critical, but the workflow engine must reason about the next task to propose. If a mandatory constraint is violated, a warning is issued or a corrective measure is performed according to what is specified for the constraint.

3.2 Declarative Workflow Representation

For the declarative workflow representation, we utilize five different constraint types of the DECLARE language [2], which countervail possible deviations:

- Precedence(t_a, t_b): Task t_b can only be executed after t_a.
- Response(t_a, t_b): Task t_a requires the enactment of t_b.
- Existence(t_a): Task t_a is mandatory.
- Not Co-Existence(t_a, t_b): Task t_a and t_b exclude each other.
- Absence(t_a, x): Task t_a can only be executed x times.

Precedence and Response prevent undesirable skipping, concerning previous or subsequent tasks. Existence contradicts the undoing of a task. Redoing and creating additional instances of a task are intercepted by the constraint Absence. Not Co-Existence avoid invoking undesired task enactments.

Preventing the user from constraint violations in our case can only be achieved, if the user manually chooses tasks from the worklist, as only such tasks are proposed that lead to a valid workflow state. As non-mandatory constraints might be violated, the worklist could consider tasks that contradict these constraints but with lower priority. Mandatory constraints should never be disregarded. Worklist handling is easy for procedurally modeled workflows, if no deviations are possible. However, if a deviation occurs, one would not be able to suggest an appropriate further proceeding. Constraints suit this situation perfectly, as even if one is violated, it might be retracted, and still valid suggestions can be computed with the help of remaining constraints. How valid task suggestions are identified, will be explained in the following section.

A prerequisite for the presented enactment approach is that each construct of the de jure workflow, modeled imperatively, is automatically transformed into corresponding declarative expressions.

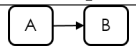
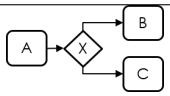
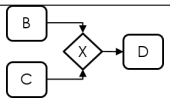
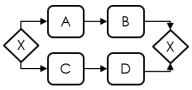
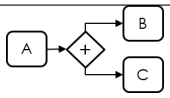
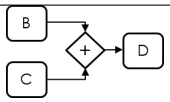
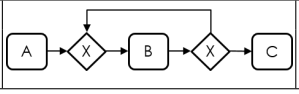
3.3 Transformation into Declarative Constructs

We consider the essential structures of imperative modeling languages on the basis of Weske [19]. The transformation rules, as described in Tab. 1, are defined analogously to [18], who in contrast uses Relation Algebra as formal specification language. The resulting constraints are used for validating upcoming enactment states of the workflow and for proposing tasks enabled for execution. The following section describes the algorithm that computes possible task suggestions on the basis of these declarative constraints.

4 Worklist Handling by means of Constraint Satisfaction Problem Solving

According to Russell and Norvig [13] a constraint satisfaction problem (CSP) is defined by a set of *decision variables* $X = \{X_1, X_2, ..., X_n\}$ and a set of *constraints* $C = \{C_1, C_2, ..., C_m\}$. Each decision variable has a *domain* D_i, which

Table 1. Mapping imperative workflow patterns to declarative constructs

Pattern	Example	Corresponding Constraints
Sequence		Precedence(A,B)
Xor-Split		Precedence(A,B) and Precedence(A,C)
Xor-Join		Precedence(B,D) or Precedence(C,D)
Xor-Sequences		Not Co-Existence(A,C) and Not Co-Existence(B,C) and Not Co-Existence(A,D) and Not Co-Existence(B,D)
And-Split		Precedence(A,B) and Precedence(A,C)
And-Join		Precedence(B,D) and Precedence(C,D)
Cycle		cf. Xor-Join and Xor-Split w.r.t. single instances of tasks

is a nonempty set of possible values for X_i. A constraint C_j is a relation over a subset of the variables $\{X_k, ..., X_l\}$, specifying the set of combination of allowed values. An assignment of values to some or all of the variables is called *state* of the problem, which is denoted as *consistent*, if it does not violate any constraint. If values are assigned to every variable, the assignment is named *complete*. A consistent and simultaneously complete assignment is called *solution*. In the following we formulate the problem of selecting the next tasks for execution in case of deviation from the de jure workflow as a constraint satisfaction problem.

4.1 Application of CSP Solving

The CSP solving algorithm is applied during workflow execution at the start of each workflow and after each task enactment. The initial state for the algorithm is a partially completed workflow, the de facto workflow, and its corresponding ideal course of events, the de jure workflow. The desired output is the set of tasks, called worklist, that can be enacted next without violating constraints.

To utilize CSP solving for worklist handling, we regard the workflow tasks as decision variables $X = T$. For each task $j_i \in J = \{j_0, ..., j_{n-1}\}$ of the de jure workflow, a variable t_{j_i} is created and added to T. Subsequently, T is supplemented with one single variable t_{end}, to be able to determine whether the workflow has completed. The elements of this set may vary while the workflow

progresses (see Sect. 4.3). The assignment of values to tasks represents a sequential order of all tasks, including not only already executed tasks, but also possible future executions of tasks. Thus, a valid order, determined by ascending integer values, of tasks is calculated. As the only thing of interest is, which task may be executed next at a specific point in time, it does not matter if any other tasks might be or have been executed in parallel. Consequently, the domain for each decision variable is a set of integer values $D_i = \{0, 1, \ldots, n\}$, with n as the number of tasks extracted from the de jure workflow including the additional variable t_{end}.

As the assignment represents a sequential execution order of all tasks, the first given constraint (see (1)) states that each assigned value of a decision variable is different from all others. Thus, only a bijective mapping of domain values to decision variables is a *solution* to the CSP.

$$
\begin{aligned}
C = \{ & \mathit{alldifferent}\,(T)\,, && && (1)\\
& t_i = c_i, \ldots && \text{(for de facto)} && (2)\\
& t_a < t_b, && \mathit{Precedence}(t_a, t_b) && (3)\\
& (t_{end} < t_a) \vee (t_a < t_b) \wedge (t_b < t_{end}), && \mathit{Response}(t_a, t_b) && (4)\\
& t_a < t_{end}, && \mathit{Existence}(t_a) && (5)\\
& (t_{end} < t_b) \vee (t_{end} < t_a), && \mathit{Not\ Co\text{-}Existence}(t_a, t_b) && (6)\\
& \sum_{i=0}^{n} f\,(\mathit{ref}\,(t_i)\,, t_a) < x, && \mathit{Absence}(t_a, x) && (7)\\
& (s_i = a_1) \Rightarrow (t_{end} < t_2), && && (8)\\
& \wedge\, (s_i = a_2) \Rightarrow (t_{end} < t_1)\} && && (9)
\end{aligned}
$$

with $\mathit{ref}\,(t_i)$ returning the id of the referenced object of the de jure workflow and $f\,(\mathit{ref}\,(t_i)\,, t_a) = \begin{cases} 1 & \text{if } \mathit{ref}\,(t_i) = t_a \\ 0 & \text{otherwise} \end{cases}$

Second, as we apply the CSP at a specific point in time during execution of the workflow, some tasks are already enacted and therefore the respective variables have a fixed assignment c_i, which is a constant value specifying the sequential execution position in the de facto workflow (see (2)). All additional constraints either result from the transformation of imperatively modeled workflow to declarative language constructs or originate from manually modeled mandatory constraints. Each type of constraint has a corresponding formal definition to be used in the CSP solving algorithm (see (3) to (7)). As some constraint violation explicitly depends upon workflow completion, t_{end} is used to determine the termination of a workflow. This is necessary to assert a mandatory enactment of a task, a required execution of a task after a certain one, or even to assure that some tasks have not been conducted. Considering the solution of the CSP every task with a higher integer assigned than t_{end} is regarded as not enacted.

Due to the construction of the CSP, we are also able to influence the control of the workflow on the basis of semantic information. Control-flow nodes are

additionally considered as constraints to further automate the control process. On this basis tasks are excluded from enactment proposals. For each xor or loop construct (cf. Fig. 1) two constraints are included (see (8, 9)). With s_i representing a decision variable, which either takes a_1 or a_2 as value, we are able to derive which path in the workflow should be followed. For workflow control the execution of the oppositional path is prevented, e.g. if the information of s_i is known to be a_1, task t_2 should not be enacted ($t_{end} < t_2$).

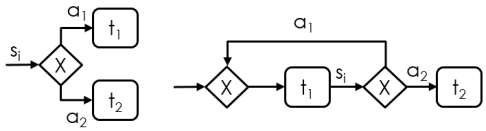


Fig. 1. Xor and loop construct

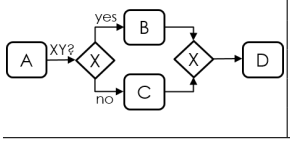
Furthermore, considering the importance of data nodes for the presented approach, additional constraints are generated on the basis of data dependencies. For example, if data node d_1 is output of task t_1 and input of t_2, a constraint $Precedence(t_1, t_2)$ is inserted in the set of constraints, as it is necessary to enact task t_1 to subsequently process d_1 with t_2.

Table 2 shows an example transformation from procedurally modeled workflow to constraints to logical formulae, which are then used by the algorithm.

4.2 Worklist Handling Algorithm

To identify all tasks that might be enacted next, a *solution* to the presented CSP, on the basis of the previously explained generated constraints, is searched for (cf. Algo.1). The algorithm takes the sets T and C as input for CSP Solving, whereas the domains D_i are derived from the size of T. The set F, also used as input, represents the de facto workflow and contains the variables from T, whose referenced tasks have been enacted. As output the variable *result* is introduced, which represents the set of tasks that might be enacted next. The value of the expected tasks, denoted here as *current*, is determined by the size of F, as this is the position which will be occupied by a task executed next. If every solution

Table 2. Example workflow and corresponding constraints

Example workflow	Declarative constraints	Logical formulae for the CSP
	Precedence(A,B), Precedence(A,C), Precedence(B,D) or Precedence(C,D), Not Co-Existence(B,C)	$A < B \wedge A < C \wedge$ $(B < D \vee C < D) \wedge$ $(End < B \vee End < C) \wedge$ $(XY = yes) \Rightarrow (End < C) \wedge$ $(XY = no) \Rightarrow (End < B)$

```
   Input  : T, C, F
   Output: result: Tasks that might be executed next
1  result = ∅;
2  foreach D_i ∈ D do D_i = {0, ..., |T| − 1};
3  current = |F|;
4  foreach x ∈ T \ F do
5  |  if solveCSP(T, D, C ∪ {x = current}) ≠ ∅ then
6  |  |  result = result ∪ {x}
7  |  end
8  end
9  return result;
```

Algorithm 1: Determination of tasks which might be executed next

to the CSP would be computed, this would result in redundant and unnecessary computations. To accelerate proceedings, we apply the CSP solving algorithm only once for each decision variable that has not been assigned a value yet, thus, has not been executed. If the CSP solving algorithm finds a solution, this task might be executed next and therefore is appended to *result*. If no solution is found, the user must not execute the task next and thus, it is not proposed.

4.3 Deviation Detection

If a task enactment occurs, variable assignments are updated and constraints are validated to analyze the state of the workflow and possibly restore consistency. Algorithm 2 illustrates this procedure. As input, the enacted task j_n of the de jure workflow is received. First, the length of the de facto workflow needs to be determined, which identifies the assigned value to the variable of the currently enacted task within the CSP. The corresponding variable t_{j_n} is included in the de facto workflow F, and constraints are extended with the value assignment of *current* to t_{j_n}. As this task enactment might result in a constraint violation, and consequently in an inconsistent workflow state, all constraints need to be validated. If no solution is found, the violated constraint needs to be identified and retracted from C to restore consistency. In case a mandatory constraint is violated, the respective warnings and corrective actions are triggered. After each task enactment the constraint set is simplified in order to prevent unnecessary computations. Based on the value assignments due to the de facto workflow, which will never change for a workflow instance, some constraints will always resolve to true, while other parts always resolve to false, even without constraint violations, e.g. disjunctive associated propositions. Assuming that the constraint set is available in conjunctive normal form $C = c_1 \wedge ... \wedge c_n$, clauses c_i are linked conjunctively and each clause represents a disjunction of literals $c_i = l_1 \vee ... \vee l_n$. The literals mostly result from the transformation of declarative workflow constructs to logical representations and thus, relate two tasks with the ordering "$<$". Other literals may be equations, such as $t_1 = 0$, depicting the de facto workflow, or *alldifferent(T)* due to the construction of the CSP. The

```
   Input  : Task j_n
1  current = |F|;
2  F = F ∪ {t_jn};
3  C = C ∪ {t_jn = current};
4  if !validateConstraints(C) then
5  |  retractViolated();
6  end
7  simplifyConstraints(t_jn);
```

Algorithm 2: Deviation Detection

literals of interest for CSP simplification are the first ones. If a task enactment of task t_i occurs, all clauses with t_i on the left side (e.g. $t_i < t_j$) in one of the literals are withdrawn, as this clause will resolve to true in any case. Furthermore single literals, which contain t_i on the right side, e.g. $t_j < t_i$, can be retracted. Those will never be fulfilled, but the remaining literals in the clause have to be.

One impact of this simplification strategy after each task enactment is that violated constraints can be easily determined. A clause consisting of a single literal, e.g. $t_i < t_j$, where the currently enacted task t_j is on the right side, is violated, as t_i has not yet been enacted, as otherwise the clause would have been withdrawn from the constraint set previously.

4.4 Extension of the CSP for Loop Patterns

Until now the approach is limited to a singular execution of tasks and not incorporating loop constructs or considering deviations like redoing a task. Thus, an algorithm is needed which alters the input sets for Algo. 1 in case of these previously mentioned scenarios. The trigger for this processing (cf. Algo. 3) is an enactment of task j_n of the de jure workflow.

To differentiate between individual task instances in case of a repeated enactment, the decision variables in T are extended with a second index variable l, e.g. $t_{(j_n,l)}$, denoting the numbering of task instances referencing one single task, here j_n, of the de jure workflow. This second index also simplifies the validation of the constraint $Absence(t_a, n)$, because n might be compared to the second index l of the tasks that have already been executed. At first, the variable $t_{(k,l)}$ corresponding to task j_n has to be found. The following condition checks whether the enacted task was the first of a loop construct. If so, the CSP is extended considering a possible further execution within the loop. Thus, a new decision variable $t_{(j_m,l+1)}$, for each task j_m in the loop, is included with an increased numbering variable $(l + 1)$. Domains have to be expanded and constraints considering the new loop tasks have to be incorporated.

5 Conclusion and Future Work

In this paper we presented a novel concept for workflow flexibility by deviation that combines procedural and declarative workflow paradigms. Upcoming doc-

```
   Input  : Task j_n
1  find t_(k,l) such that k = j_n and t_(k,l) ∈ T \ F;
2  if isFirstTaskInLoop(j_n) then
3  |  foreach j_m ∈ loop do
4  |  |  T = T ∪ {t_(j_m,l+1)}, with m = |T|;
5  |  |  foreach D_i ∈ D do D_i = D_i ∪ {|T| − 1};
6  |  |  addConstraints with usual loop constraints;
7  |  end
8  end
```

Algorithm 3: CSP extension

uments and extracted semantic information are used to determine the current state of the workflow and for control purposes. In order to react to deviations and still propose the best way of proceeding with the workflow, constraint satisfaction problem solving is applied. Future work will focus on a detailed elaboration of the single algorithms and possible improvements to achieve better results. Subsequently, the implementation will be evaluated against related approaches. Additionally, the concept will be developed further, as the workflow designer should be granted more freedom to choose how strict or flexible the constraints should be treated for worklist handling and, furthermore, which and how countermeasures could be specified.

References

1. van der Aalst, W.M.P.: Business Process Management - A Comprehensive Survey. ISRN Software Engineering 2013, 1–37 (2013)
2. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative workflows: Balancing between flexibility and support. Computer Science - R&D 23(2), 99–113 (2009)
3. van der Aalst, W.M.P., Weske, M., Grünbauer, D.: Case handling: a new paradigm for business process support. Data Knowl. Eng. 53(2), 129–162 (2005)
4. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services, Version 1.1. Tech. rep., BEA Systems, International Business Machines Corporation, Microsoft Corporation (2003)
5. Dadam, P., Reichert, M., Rinderle-Ma, S.: Prozessmanagementsysteme - nur ein wenig Flexibilität wird nicht reichen. Informatik Spektrum 34(4), 364–376 (2011)
6. Fahland, D., Lübke, D., Mendling, J., Reijers, H.A., Weber, B., Weidlich, M., Zugal, S.: Declarative versus Imperative Process Modeling Languages: The Issue of Understandability. In: Enterprise, Business-Process and Information Systems Modeling, 10th International Workshop, BPMDS 2009, and 14th International Conference, EMMSAD 2009, held at CAiSE 2009, Amsterdam, The Netherlands, June 8-9, 2009. Proceedings. pp. 353–366 (2009)
7. Grumbach, L., Rietzke, E., Schwinn, M., Bergmann, R., Kuhn, N.: SEMAFLEX - Semantic Integration of Flexible Workflow and Document management. In: Krestel, R., Mottin, D., Müller, E. (eds.) Proceedings of the Conference "Lernen, Wissen,

Daten, Analysen", Potsdam, Germany, September 12-14, 2016. CEUR Workshop Proceedings, vol. 1670, pp. 43–50. CEUR-WS.org (2016)
8. Hildebrandt, T.T., Mukkamala, R.R.: Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In: Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010. pp. 59–73 (2010)
9. Hoffmann, D.: Schlanke Formen des Geschäftsprozessmanagements Das richtige BPM-Rezept (February 2013), http://www.it-zoom.de/it-mittelstand/e/das-richtige-bpm-rezept-5287/
10. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata. In: Business Process Management - 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30 - September 2, 2011. Proceedings. pp. 132–147 (2011)
11. Pesic, M.: Constraint-based workflow management systems: shifting control to users. Ph.D. thesis, Technische Universiteit Eindhoven (2008)
12. Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.A.: Imperative versus Declarative Process Modeling Languages: An Empirical Investigation. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I. Lecture Notes in Business Information Processing, vol. 99, pp. 383–394. Springer (2011)
13. Russell, S.J., Norvig, P.: Artificial Intelligence - A Modern Approach (3. internat. ed.). Pearson Education (2010)
14. Saam, M., Viete, S., Schiel, S.: Digitalisierung im Mittelstand: Status Quo, aktuelle Entwicklungen und Herausforderungen (August 2016)
15. Schlecht, M.: Prozessmanagement in der Cloud. ERP Management 4/2013: Betriebsformen moderner Systeme 3, 33–36 (2013)
16. Schonenberg, H., Mans, R., Russell, N., Mulyar, N., van der Aalst, W.M.P.: Process Flexibility: A Survey of Contemporary Approaches. In: Advances in Enterprise Engineering I, 4th International Workshop CIAO! and 4th International Workshop EOMAS, held at CAiSE 2008, Montpellier, France, June 16-17, 2008. Proceedings. pp. 16–30 (2008)
17. Supyuenyong, V., Islam, N., Kulkarni, U.R.: Influence of SME characteristics on knowledge management processes: The case study of enterprise resource planning service providers. J. Enterprise Inf. Management 22(1/2), 63–80 (2009)
18. Wedemeijer, L.: Transformation of Imperative Workflows to Declarative Business Rules. In: Shishkov, B. (ed.) Business Modeling and Software Design - Third International Symposium, BMSD 2013, Noordwijkerhout, The Netherlands, July 8-10, 2013, Revised Selected Papers. Lecture Notes in Business Information Processing, vol. 173, pp. 106–127. Springer (2013)
19. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer (2007)
20. Westergaard, M.: Better Algorithms for Analyzing and Enacting Declarative Workflow Languages Using LTL. In: Business Process Management - 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30 - September 2, 2011. Proceedings. pp. 83–98 (2011)
21. Workflow Management Coalition: Workflow Management Coalition Terminology & Glossary (February 1999)