JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2012; **24**:285–302
Published online 8 August 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.554

Toward enhanced life-cycle support for declarative processes

Stefan Zugal*,†, Jakob Pinggera and Barbara Weber

Institute of Computer Science, University of Innsbruck, Austria

SUMMARY

The need for flexible process-aware information systems resulted in a recent interest in declarative approaches, as they promise a high degree of flexibility. However, the potential of current declarative approaches is impeded by deficiencies in understandability and maintainability. This paper proposes an approach toward better understandability and maintainability of declarative processes by adopting well-established techniques from the domain of software engineering. More specifically, the ideas of test-driven development and automated acceptance testing are adopted to interweave process specification and process testing. Thereby, during modeling, testcases balance the circumstantial/sequential information mismatch as well as improve understandability by dispensing with hard mental operations and removing hidden dependencies. Because testcases are also understandable to domain experts, they foster communication between domain experts and model builders, providing a common basis for communication. During process execution, testcases, in turn, help to document the reasons for process deviations and ensure that respective deviations can be easily considered during schema evolution. Furthermore, testcases ensure that no undesired behavior is introduced through process adaptations. Copyright © 2011 John Wiley & Sons, Ltd.

Received 21 June 2011; Accepted 21 June 2011

KEY WORDS: Business Process Management; Declarative Business Process Models; Business Process Life-Cycle; Process Model Understandability; Process Model Maintainability

1. INTRODUCTION

In today's prevalent dynamic business environment, the economic success of an enterprise depends on its ability to react to various changes like shifts in customer's attitudes or the introduction of new regulations and exceptional circumstances [1,2]. Process-aware information systems (PAISs) offer a promising perspective on shaping this capability, resulting in growing interest to align information systems in a process-oriented way [3,4]. Yet, a critical success factor in applying PAISs is the possibility of flexibly dealing with process changes [1]. To address the need for flexible PAISs, competing paradigms enabling process changes and process flexibility have been developed, for example, adaptive processes [5,6], case handling [7], declarative processes [8], data-driven processes [9], and late binding and modeling [10] (for an overview see [11]).

All these approaches relax the strict separation of build-time (i.e., modeling) and run-time (i.e., execution), which are typical for plan-driven approaches as realized in traditional workflow management systems. By closely interweaving modeling and execution, the approaches mentioned earlier allow for a more agile way of planning. In particular, users are empowered to defer decisions regarding the exact control-flow to run-time, when more precise information becomes available.

1.1. Problem statement

Depending on the concrete approach, planning and execution are interwoven to different degrees, resulting in different levels of decision deferral. The highest degree of decision deferral is fostered

*Correspondence to: Stefan Zugal, Institute of Computer Science, University of Innsbruck, Austria.
†E-mail: stefan.zugal@uibk.ac.at

by late composition [11] (e.g., as enabled through a declarative approach), which describes activities that can be performed as well as constraints prohibiting undesired behavior. A declarative approach, therefore, is particularly promising for dynamic and unpredictable processes [8,12]. The support for partial workflows [12] allowing users to defer decisions to run-time [11], the absence of over-specification [8], and more maneuvering room for end users [8] can be all considered as advantages commonly attributed to declarative processes. Although the benefits of declarative approaches seem rather evident, they are not widely adopted in practice yet. Declarative processes are only rudimentarily supported and integrated process life-cycle support has not been in place yet, whereas methods and tools for supporting imperative processes are rather advanced (e.g., [13]).

Reasons for the lacking adoption of declarative approaches seem to be related to understandability and maintainability problems [14,15]. In particular, methods and tools addressing respective issues are still missing. In order to tackle these problems, we adopt well-established techniques from the domain of software engineering. More specifically, test-driven development (TDD) [16] and automated acceptance testing (AAT) [17] are combined and adapted for better supporting the declarative process life cycle. As a result, we provide a first approach toward enhanced usability of declarative process management systems.

The paper is organized as follows: Section 2 provides background information, whereas Section 3 introduces the running example. Then, Section 4 discusses possible reasons for understandability and maintainability problems of declarative processes. Section 5 presents a framework and methodology addressing these issues. Limitations are described in Section 6. Section 7 deals with related work, and Section 8 concludes with a summary and outlook.

2. BACKGROUNDS

This section provides background information on declarative processes in Section 2.1 and the "process of modeling" in Section 2.2.

2.1. Declarative processes

There is a long tradition of modeling business processes in an imperative way. Process modeling languages supporting this paradigm, like BPMN, BPEL, and UML activity diagrams are widely used. Recently, *declarative approaches* have received increased interest and suggest a fundamentally different way of describing business processes [14]. Whereas imperative models specify exactly *how* things have to be done, declarative approaches only focus on the logic that governs the interplay of actions in the process by describing: (i) the *activities* that can be performed, as well as (ii) *constraints* prohibiting undesired behavior. An example of a constraint in an aviation process would be that crew duty times cannot exceed a predefined threshold. Constraints described in literature can be classified as execution and termination constraints. *Execution* constraints restrict the execution of activities, for example, an activity can be executed at most once. *Termination* constraints, on the other hand, affect the termination of process instances and specify when process termination is possible. For instance, an activity must be executed at least once before the process can be terminated.

Imperative models take an "*inside-to-outside*" approach by requiring all execution alternatives to be explicitly specified in the model. Declarative models, in turn, take an "*outside-to-inside*" approach: constraints implicitly specify execution alternatives as all alternatives have to satisfy the constraints [14]. Adding further constraints means discarding some execution alternatives (cf. Figure 1). This results in a coarse up-front specification of a process, which can then be refined iteratively during run-time.

2.2. The process of modeling

The "process of modeling", that is, the process of creating a process model, is commonly described as iterative and collaborative [18]. Typically, several roles are involved in the process of modeling [19]:

DOI: 10.1002/smr

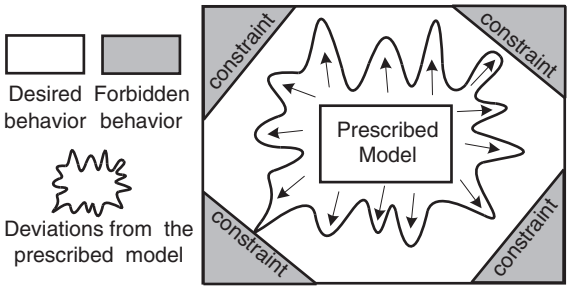

Figure 1. Declarative versus imperative [14].

- Domain expert (DE): provides information about the domain and decides which aspects of the domain are relevant to the model and which are not.
- Model builder (MB): is responsible for formalizing information provided by the DE.
- Modeling mediator (MM): helps the DE to phrase statements concerning the domain that can be understood by the MB. In addition, the MM helps to translate questions the MB might have for the DE.

In particular, the process of modeling is divided into an *elicitation dialogue* and *formalization dialogue* [20]. During the elicitation dialogue, the DE conveys information about the business domain to the MB, that is, capturing the requirements of the business process. In the formalization dialogue, the MB is responsible for transforming this informal information into a formal process model. However, as argued by Rittgen, much of the information is not gathered by the DE only, but created through the communication process itself [21].

3. EXAMPLE

To illustrate the concepts of declarative business processes as well as the proposed framework and methodology, we introduce a running example (cf. Figure 2). The example process is rather meant to provide an exemplary process within a familiar domain than a comprehensive description of how to write a paper. For the sake of brevity, we will use the following abbreviations:

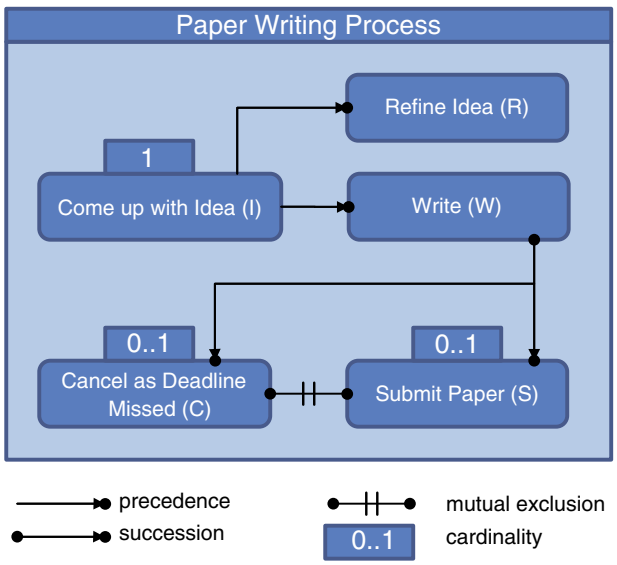

Figure 2. Exemplary declarative process.

DOI: 10.1002/smr

I Come up with idea
R Refine idea
W Write
S Submit paper
C Cancel as deadline missed

The used constraints are taken from [14], and their meaning is as follows: *precedence* specifies that the execution of a particular activity requires another activity to be executed before (not necessarily directly). For example, the precedence constraint between *I* and *R* allows execution traces $<I,W,R,S>$ and $<I,R,W,S>$, but not $<R,W,S>$. The constraint *succession* is a refinement of the precedence constraint. In addition to the precedence relation, it demands that the execution of the first activity is followed by the execution of the second one. For example, the succession constraint between *W* and *S* is satisfied for execution traces $<I,W,S>$, $<I,W,R,S>$, but not for $<I,R,S>$ (*W* not executed) and $<I,R,W>$ (*S* not executed after *W*). By using the *mutual exclusion* constraint, it is possible to define that only one of two activities can be executed for a process instance. For example, the mutual exclusion between *C* and *S* allows execution traces $<I,W,S>$ and $<I,W,C>$, but not $<I,W,S,C>$. Finally, the *cardinality* constraint restricts how often an activity can be executed. For example, the cardinality constraint (0..1) of *S* allows the trace $<I,W,R,S>$, but not $<I,W,R,S,S>$.

Having the constraints' semantics in mind, the process can be described in the following way: After an initial idea has been devised, it is possible to start working on the paper and to refine the idea at any time. If it turns out that the deadline cannot be met, the work on the paper will be stopped. Otherwise, as soon as the idea is described sufficiently well, the paper can be submitted.

This example indicates three interesting properties of declarative process models: *Firstly*, the declarative nature allows for an elegant specification of business rules, especially for unstructured processes. For an imperatively modeled process, it would be difficult to deal with multiple (possibly) parallel executions of *R* and *W*, in combination with the mutual exclusion constraint between *C* and *S*. *Secondly*, for assessing whether certain behavior is supported by the process model, the reader has to interpret the constraints in his mind, for example, "Is it possible to come up with an idea, refine it, write on a paper, but then cancel the work?" *Thirdly*, it is not obvious where to start reading the model. As there are no explicit start nodes, it is up to the reader to figure out where to start—in this case perhaps by following the precedence and succession constraints.

4. UNDERSTANDABILITY AND MAINTAINABILITY OF DECLARATIVE PROCESS MODELS

Although declarative business processes provide a big potential for flexible process execution, their adoption is currently limited. This section elaborates on factors impeding their adoption. In particular, Section 4.1 discusses problems related to the understandability of declarative process models, whereas Section 4.2 deals with maintainability issues.

4.1. Understandability

Whereas there is evidence that declarative process models suffer from understandability problems [14,15], respective countermeasures are not in place yet. As Pesic points out, "*models with many constraints can easily become too complex for humans to cope with*" [14]; subsequently, possible explanations are discussed.

Sequential and circumstantial information. An interesting perspective regarding understandability issues of declarative processes is provided by Fahland *et al*. [22,23]. According to their work, process models exhibit both *sequential* and *circumstantial* information. Sequential information describes chronological behavior (e.g., A *directly follows* B), whereas circumstantial information captures general relations (e.g., A must be executed *at least once*). Depending on the process modeling language, either sequential or circumstantial information may be favored, that is, can be explicitly described by the modeling language. Thus, in general, modeling languages can be characterized along a spectrum of explicitness between sequential and circumstantial. Imperative

DOI: 10.1002/smr

languages (e.g., BPMN, Petri Nets) reside on the sequential side of the spectrum, whereas declarative approaches (e.g., ConDec [14]) are settled on the circumstantial end [23]. Although the circumstantial nature of declarative languages allows for a specification of highly flexible business process models [14], it inhibits the extraction of sequential information (e.g., execution traces) at the same time. This, in turn, compromises understandability, as it makes it harder to see whether a process model supports the execution of a specific process instance or not.

Figure 2 illustrates the concepts of sequential and circumstantial information. For instance, the precedence constraint between *I* and *R* defines that R can only be executed if I is completed. The constraint thereby does not define *how* an execution trace containing both activities has to look like in detail (sequential information), but rather defines a certain characteristic of the trace: no occurrence of *I* before *R* (circumstantial information). Thus, only little sequential information, but mostly circumstantial information is conveyed. With respect to the mutual exclusion constraint, there is an even higher ratio of circumstantial information: it defines that one out of two activities can be executed (circumstantial), but not both—the ordering of respective activities is not taken into account at all (sequential).

Cognitive dimensions framework. According to the cognitive dimensions framework [24,25], the extraction of sequential information from declarative processes can be seen as a *hard mental operation*, as the required interpretation of constraints has to be performed in the reader's mind. From cognitive research, it is known that the human mind can only hold up to 7 ($\pm$ 2) items in short term memory [26]. Thus, it seems reasonable that the interpretation of several interrelated constraints is hard to handle. Considering again Figure 2, it becomes apparent that the interpretation of seven constraints in one's mind is no trivial task, that is, a hard mental operation.

In addition, the extraction of sequential information is hampered by the so-called *hidden dependencies* dimension of cognitive dimensions framework [24,25]. Because constraints are interconnected, it is not sufficient to look at constraints in isolation; the reader must also take into account the interplay of constraints. However, this interplay may not always be obvious, that is, can be hidden, thus making understanding difficult.

Figure 2 also illustrates hidden dependencies. For instance, after activities *C* or *S* have been completed, *W* cannot be executed anymore. This behavior is introduced by the combination of the cardinality constraints of *C* and *S* as well as the succession constraint of *W* and the mutual exclusion constraint between *S* and *C*. If *W* was executed after *C* or *S*, the process instance could not be terminated anymore as the succession constraint demands the execution of either *C* or *S* after *W*. However, neither *C* nor *S* can be executed because of the combination of cardinality—and mutual exclusion constraint and consequently the process instance cannot be terminated. Thus, the workflow engine *must prohibit* the execution of *W* (cf. [14]), which is not apparent from looking at the process model.

Readability of declarative process models. Further insights into understandability issues are motivated by [27], showing that models are not read at once, but chunk-wise, that is, bit by bit. Whereas graph-based notations inherently propose a way of reading imperative process models chunk by chunk—namely from start node(s) to end node(s)—this approach does not work for declarative models. Because declarative models do not have explicit start nodes and end nodes, it is not always obvious where the reader should start reading the model. In fact, it is not unlikely for a declarative model to have several starting points, as the focus is put on *what* should be modeled, but not precisely *how*. Similarly, experiments show that incrementally understanding declarative processes is vital for success [15,28].

Thus, when reading a declarative process model, users have to rely on secondary notation, for example, layout. For example, regarding Figure 2, one might assume that the process starts top left and ends bottom right. However, it depends on the person who created the model, whether this strategy succeeds. Another way of reading could be to follow precedence constraints and succession constraints to find the model's start. In short, it is neither obvious which strategy to choose nor which one works best.

DOI: 10.1002/smr

4.2. Maintainability

External factors like the introduction of new laws or changes in customer attitude are possible reasons for the evolution of process models. In addition, an inappropriate representation of the business domain may result in users bypassing the workflow system causing "system workarounds" [29]. Capturing deviations from the predefined process is hence desirable and indeed, sophisticated approaches have been developed for imperative workflow management systems, for example, [13]. For declarative systems, however, only rudimentary support is in place.

In addition to gathering the change requirements, the adaption of declarative process models is far from trivial. With increasing model size, declarative models are not only difficult to understand but also hard to maintain. As pointed out by Weber *et al*. [15]: "*it is notoriously difficult to determine which constraints have to be modified and then to test the newly adapted set of constraints*". An explanation of problems regarding maintainability is provided by the observation that adapting process models involves both *sense-making* tasks (i.e., to determine parts of the model that need to be changed) and *action tasks* (i.e., to apply the respective changes to the model) [25]. Whereas the action task is rather simple—adding/removing activities and adding/removing constraints—the sense-making task is far from trivial as detailed in the following.

As illustrated, declarative process models suffer from understandability problems, thus impeding the sense-making task. Besides, because of the interplay of constraints, it is hard to see how changes influence other parts of the process model. Consider, for instance, the introduction of a mutual exclusion constraint between *R* and *W* in Figure 2. This change not only restricts the relationship between *R* and *W* but also introduces a deadlock for the execution trace $<I,R>$. Whereas the execution of *R* is still possible, neither *C* nor *S* can be executed anymore, because they require *W* to be executed before. However, as *R* and *W* are mutual exclusive, *W* cannot be executed.

This example illustrates that *local* changes to the process model can have effects on other parts of the model that are not obvious, that is, the change becomes *global*. Whereas methods exist that ensure the syntactic correctness, for example, absence of deadlocks [14], computer-supported methods for ensuring the *validity* of the model, that is, whether the model adequately represents the real-world business, are not in place yet.

5. TESTING FRAMEWORK FOR DECLARATIVE PROCESSES

To address the understandability and maintainability issues sketched afore, we propose a framework for the validation of declarative processes. Section 5.1 introduces background information about the software engineering techniques TDD and AAT; these methodologies have been chosen as they address maintainability and understandability problems in the domain of software engineering [30,31]. Section 5.2 illustrates how respective techniques can be adapted to the domain of business processes and introduces the testing framework. Section 5.3 focuses on methodological aspects and their application in the declarative process life cycle.

5.1. Software testing techniques

As the proposed framework and methodology build upon techniques from software engineering, respective techniques are introduced in the following.

5.1.1. Test-driven development. Software engineering processes typically differentiate between the phases of *system design and implementation* as well as *testing*. That means, after the required functionality has been developed, the software system's defects are (more or less) systematically searched for and corrected. The idea of TDD, however, is to interweave the phases of system development and testing [16]. As the name suggests, *automated* testcases are specified *before* the actual production code is written. Whenever a new feature is introduced, a testcase is created to ensure that the feature is implemented properly. In addition, developers execute all testcases to verify that existing behavior is preserved, and the new feature does not introduce undesired behavior, that is,

DOI: 10.1002/smr

"*brakes the existing code*" [16]. Studies show that the adoption of TDD indeed leads to improvements with respect to the number of software defects and design quality (e.g., [30,31]).

It is worthwhile to note that TDD, as a byproduct, enables *regression testing*, that is, testing whether the development of new functionality in a software system preserves the correctness of the existing system [32]. In particular, testcases that have been specified during the development can directly be used for regression testing.

5.1.2. Automated acceptance testing. Similar to TDD, the idea of AAT [17] is the creation of executable testcases. AAT, however, focuses on the interaction between *customers* and *developers*. Testcases in AAT need to be understandable to customers without technical background, but still exhibit strict semantics. Thereby, *testcases* act as means of communication as they can be understood by both customers and developers, allowing for a better integration of the customer in the development process. This, in turn, supports the identification of system requirements [17]. The automated validation of the developed system against the testcases ensures that the desired functionality is actually provided by the software system. The testcases are seen as *contract of acceptance*: only if the software system passes all testcases, it will be accepted by the customer.

5.2. *Process testing framework concepts*

This section gives an overview of the testing framework (cf. Figure 3) before the individual concepts are explained in detail in Section 5.2.1.

The interplay of DE, MB, testcases, and process model is illustrated in Figure 3. The DE, possibly with help of the MB, creates a testcase specifying the intended behavior of the declarative process (1a

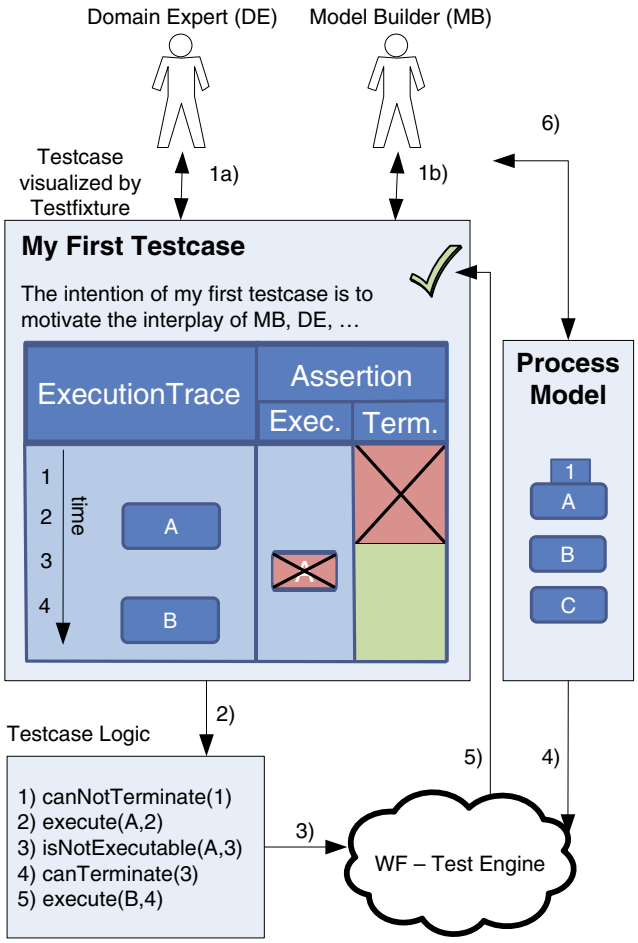

Figure 3. Framework for the validation of declarative processes.

DOI: 10.1002/smr

and 1b). In order to foster communication between DE and MB, testcases are represented by a rich graphical user interface, the so-called *testfixture*. Because the testfixture is understandable to both the DE and MB, it serves as base for discussion. To validate a testcase, the *testcase logic* is automatically extracted from the testcase (2) and fed into the test engine (3). For the validation of testcases, the test engine also needs to access the process model (4). After the test engine has completed the validation process, results are reported to the MB/DE by indicating whether the test failed and for the case it failed, the reason why (5). Depending on the feedback, the MB might decide to adapt the process model (6).

5.2.1. Testcase structure. Testcases are key components of the testing framework and are essential parts of TDD and AAT. A testcase for a declarative process model consists of *text* explaining the intention of the testcase, an *execution trace* describing the behavior to be tested, a set of *assertions* specifying the conditions to be validated, as well as a graphical representation by a *testfixture*.

Textual description. The *textual description* in a testcase helps to capture information that cannot be expressed directly in a process model but is still necessary to fully understand it. For instance, textual descriptions can be used to document why certain behavior must be present in the process model. For example, the intention of the testcase shown in Figure 3 is to give an overview of the proposed testing framework and illustrates the interplay of concepts.

Execution trace. As motivated by AAT, testcases are *executable specifications* [17]. Instead of using an informal document describing the requirements of the process model in natural language, the specification should not only be readable by humans but *also* accessible to automated interpretation. The term *execution* thereby refers to the idea of checking the described requirements using a test engine in an automated manner, that is, "the testcase gets executed".

Within a testcase, execution traces are used to capture behavior to be tested, as they contain all relevant information about the execution of a process, for example, execution of an activity or termination of a process instance. Thereby, an execution trace allows restoring an arbitrary state of a process instance by replaying the steps in the execution trace on the workflow engine. Thus, execution traces provide the basis for an executable specification, capturing behavior in a machine-readable form.

Considering the testcase illustrated in Figure 3, the execution trace can be found on the left-hand side of the testcase. The execution trace includes the execution of two activities: activity *A* at time 2 and activity *B* at time 4.

Assertion. Whereas execution traces allow specifying behavior that must be supported by a process model, they do not provide means to specify behavior that *must not* be included. For instance, regarding the process model shown in Figure 3, activity *A* must be executed *exactly once*. Put differently, activity *A* must be at least once in the execution trace, but at the same time *not* more than once.

In order to support such scenarios, a testcase also contains a set of *assertions*. Using assertions, it can be validated whether certain behavior is supported/prohibited by the process model. Because declarative process models require explicit termination, that is, the user must terminate the instance explicitly, we differentiate between execution and termination assertions:

- isExecutable (a, t), activity *a* is executable at time *t*.
- isNotExecutable (a, t), activity *a* is **not** executable at time *t*.
- canTerminate (t), the process instance can be terminated at time *t*.
- canNotTerminate (t), the process instance can **not** be terminated at time *t*.

As discussed in Section 2.1, the behavior specified in a declarative process model is prescribed by its activities and constraints. The former enumerate tasks, which may be executed, whereas the latter ones restrict their execution and the termination of the process instance. Thus, the aforementioned assertions should be sufficient to cover any condition with respect to control flow, because they can deal with activity execution as well as process termination.

DOI: 10.1002/smr

Considering Figure 3, three assertions can be found at the right-hand side of the testcase. The assertions are organized in two columns to separate execution from termination assertions. At time 1, before the execution of activity *A*, assertion *canNotTerminate (1)* (crossed out area are on the right) is specified. After *A* has been executed at time 3, *isNotExecutable (A,3)* (crossed out rectangle on the left), and *canTerminate (3)* (non-crossed area in the right column) can be found.

Testfixture. So far, we illustrated the textual description, the execution trace, and assertions of a testcase. Whereas these concepts suffice to specify the behavior to be tested, it is very likely that the DE cannot cope with these technical details. To enable the DE to read and specify testcases, *testfixtures* provide an intuitive graphical representation of a testcase. Depending on the situation, a specific visualization may be suitably best. For instance, when control-flow constraints should be tested (e.g., activity B must be preceded by activity A), a simple time-line fixture may be sufficient (cf. Figure 3). However, for testing temporal constraints [33] (e.g., activity A can only be executed once per day), a calendar-like fixture might be more appropriate. Furthermore, it might be beneficial to provide a user interface the DE is familiar with (e.g., calendar from Microsoft Outlook).

Figure 3 shows an exemplary testfixture for testing control-flow constraints. It consists of a simple timeline on the left hand side as well as an area for the specification of constraints at the right-hand side. Assertions *isExecutable* and t canTerminate are represented by empty rectangles, *isNotExecutable* and *canNotTerminate* by crossed out rectangles.

5.2.2. Testcase validation. The previous section has introduced testcases, which serve as executable specification and are created by the DE (with the help of the MB). This section describes how the testcases can be used to automatically validate the process model. Each testcase is responsible for providing a precise definition of the behavior to be tested, that is, the execution trace and assertion statements. For the actual execution of the testcase, the test engine provides an artificial environment, in which the process is executed. The procedure for the validation of a testcase is straight forward:

- Initialize the test environment.
- For each event in the execution trace and assertion, ascending by time:
  - *if event*: The test engine interprets the log event and manipulates the test environment, for example, execution of an activity. If the log event cannot be interpreted, for example, because the activity cannot be executed, the testcase validation will be stopped and the failure reported.
  - *if assertion*: Test whether the assertion holds for the current state of the test environment. For the case the condition does not hold, the testcase validation will be stopped and the failure reported.
- In case all assertions passed, report that the testcase passed. Otherwise, provide a detailed problem report, for example, report the constraint that caused the failure or the state of the process instance when the testcase failed.

With respect to the testcase illustrated in Figure 3, the test engine takes the testcase logic and the process model as input. Then, the engine interprets the testcase logic as follows:

- *canNotTerminate (1)*: Check that the process instance cannot be terminated without executing any activity.
- *Execute (A,2)*: Execute activity *A* at time 2 and complete it at time 3.
- *isNotExecutable (A,3)*: Check that *A* cannot be executed anymore at time 3.
- *canTerminate (3)*: Test whether the process instance can be terminated at time
- *execute (B,4)*: Execute activity *B* at time 4 and complete it at time 5.

All events from the execution trace/all assertions could be interpreted without problems, thus report that the testcase passed.

5.3. Test-driven modeling and the declarative process life cycle

So far, we have elaborated on understandability and maintainability issues of declarative models and introduced a testing framework adopting TDD and AAT. However, TDD and AAT are not only about testcases in isolation but also describe a methodology on how testcases are formulated. In the

DOI: 10.1002/smr

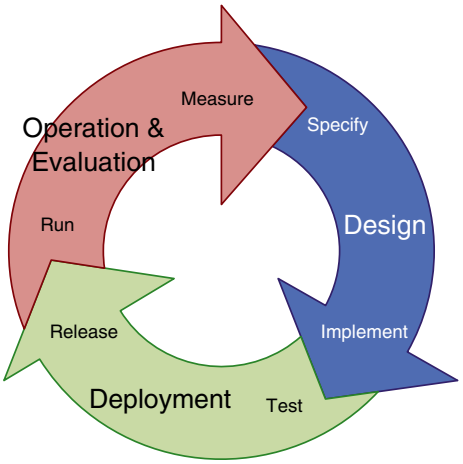

Figure 4. Process life cycle, adapted from [4].

following, we show how the methodological aspects of TDD and AAT fit into each phase of the process life cycle (cf. Figure 4), and how they provide enhanced support for declarative processes. Section 5.3.1 captures both design and deployment phases. Then, Section 5.3.2 discusses the operation and evaluation of processes.

5.3.1. Design and deployment phase. In the following, we focus on the phases of process design and process deployment. In particular, we focus on *process specification* and *process testing*.

Test-driven modeling (TDM). The process of modeling is a collaborative but still manual process (cf. Section 2.2). By adopting TDD and AAT techniques, process specification and process testing get closely interwoven. In particular, testcases serve as MM [19] mediating between DE and MB. As illustrated in Figure 5, testcases provide means to talk about both the business domain and the process model. The DE is not longer forced to rely on the information provided by the MB solely (2)—the specification of testcases (4) and their automated validation against the formal process model (6) provide an unambiguous basis for communicating with the MB (5). This does not mean that the DE and MB do not communicate directly anymore. Rather, tests provide an *additional* communication channel. Thereby, modeling minutes, which are formulated as testcases instead of informal text can be automatically validated against the process model, relieving the MB from manually checking the process model against the informal specification.

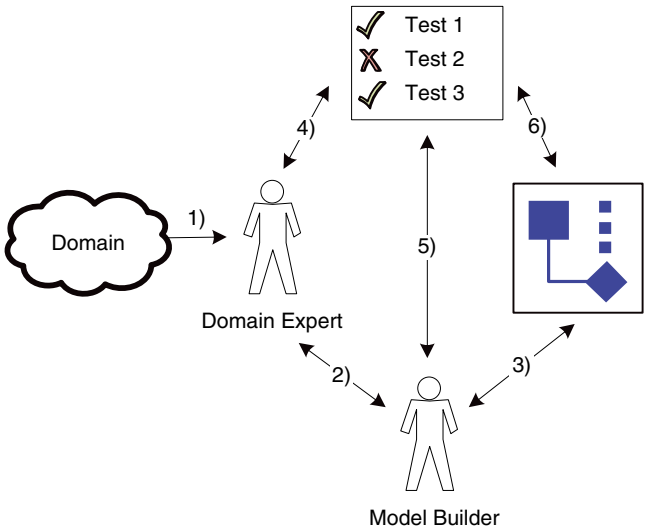

Figure 5. Communication channels.

DOI: 10.1002/smr

It is important to stress that the TDM's process of modeling is of iterative nature. Rather then specifying all testcases up-front and modeling the process afterwards against the specification, testcases and model are refined iteratively. As illustrated in Figure 6, when a new process model is specified, the DE (with the help of the MB) describes a requirement in the form of a testcase. Then, *all* testcases are validated against the process model to check whether the specified behavior is already supported by the model. For the case that all tests pass, the DE and MB can move on with specifying the next requirement, that is, testcase. If at least one testcase fails, the DE and MB will discuss whether the failed testcases are valid, that is, capture the business domain properly. If a testcase is valid, it can be assumed that the model does not capture the business domain properly and therefore needs to be adapted. However, if the discussion reveals that the testcase is invalid, the testcase needs to be adapted to represent the business domain properly. In either case, all testcases are run to ensure that the conducted adaption had the desired effect. Subsequently, new testcases are defined/adapted or the model is adapted iteratively until both DE and MB are satisfied with the testcases and process model.

Whereas the basic idea, as illustrated in Figure 6, is to start by specifying testcases, for some situations one might also start with the modeling part or a non-empty model. For instance, when working with existing process models, it is neither feasible nor meaningful to start from scratch. Furthermore, depending on the MB's or DE's skills and preferences, it is not necessary to strictly follow the test-before-model idea, for example, to start from an initial model capturing the process logic roughly and refine it using testcases. Such deviations from the original TDM process are acceptable—as long as testing and modeling stays interwoven. Otherwise, the benefits of TDM are likely to be diminished.

The idea of TDM is illustrated based on an exemplary modeling session, cf. Figures 7–9. To recapitulate, we use the following abbreviations:

I Come up with idea
W Write
S Submit paper

Starting from an empty process model, the DE lines out general properties of the process: "*When writing a publication, you need to have an idea. Then you write the publication and submit it*." Thus, possibly with help of the MB, the DE inserts activities *I*, *W*, and *S* in the testcase's execution trace (cf. Figure 7). Respective activities are automatically created in the process model. Now, the DE and MB run the test, and the test engine reports that the test passes.

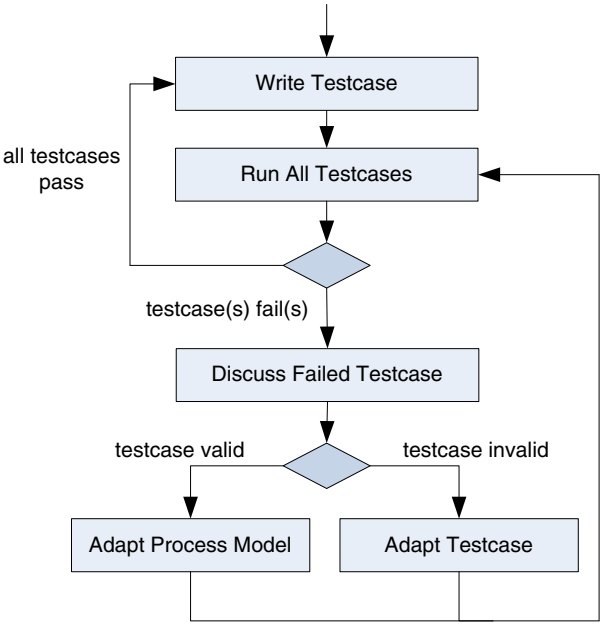

Figure 6. Test-driven modeling.

DOI: 10.1002/smr

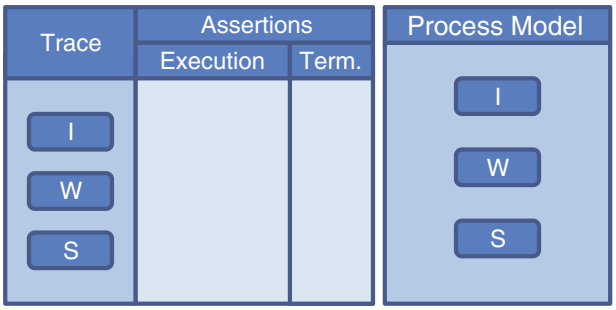

Figure 7. Testcase 1: <*I,W,S*> proposed by the DE.

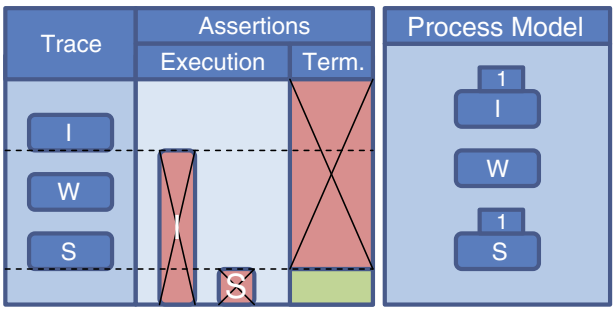

Figure 8. Testcase 1: introduction of cardinality on *I* and *S*.

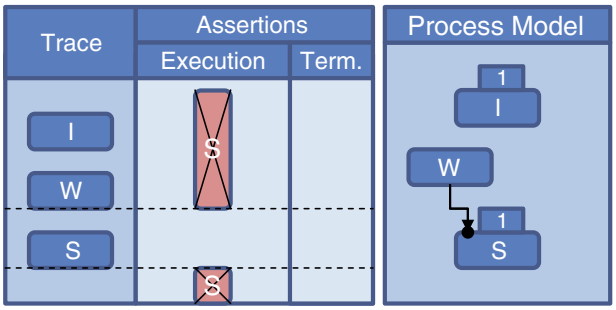

Figure 9. Testcase 2: introduction of precedence between *W* and *S*.

Subsequently, the DE and MB engage in a dialogue of questioning and answering [20]—the MB challenges the model: "*So can I submit the paper several times?*" "*You should submit the paper, but, at most once*!" the DE replies and adds: "*And you should only have a single idea—otherwise the reader gets confused*." Thus, they adapt the testcase capturing this requirement and run it (cf. Figure 8). Apparently, the testcase fails as there are no constraints in the model yet. After ensuring that the requirement is valid, the MB adapts the model—inserts cardinality constraints on *I* and *S*—and the test passes (cf.Figure 8).

Again, the MB challenges the model and asks: "*Is it possible to submit an idea without paper*?" The DE replies: "*No, you always need a written document*," and together, they specify a *second testcase* that ensures that *S* cannot be executed without at least one execution of *W* before. By automatically validating the second testcase, it becomes apparent that *S* can be executed before *W* has been finished. Thus, the MB introduces a precedence constraint between *W* and *S* (cf. Figure 9).

Even though, because of lack of space, the given example was kept small, it illustrates the benefits of TDM for the design and deployment phases, which are detailed in the following:

Improving understandability. As discussed in Section 1, declarative process models do not provide explicit support for sequential information, thereby forcing the MB to construct respective

DOI: 10.1002/smr

information in his mind. At this point, the sequential nature of testcases is exploited: because specification and testing are interwoven, testcases and models are paired together. Thereby, testcases provide an *explicit* source of sequential information. The construction of sequential information is supported by the automated validation of testcases, thus avoiding *hard mental operations*. In addition, by specifying a respective testcase, implicit dependencies between constraints can be made explicit. This, in turn, helps the MB to deal with *hidden dependencies*.

According to [16], testcases should focus on a single aspect only. For instance, the testcase shown in Figure 9 focuses on the execution of activity *S*. Thereby, a chunk of the process model is presented, focusing on a certain aspect of the model only. This, in turn, enables the MB to read the model testcase by testcase, that is, chunk-wise.

Besides, the ordering of the activities specified in the testcase proposes a way of reading the process model. Consider, for instance, the testcase illustrated in Figure 9: Activities *I*, *W*, and *S* are ordered consecutively. Thus, the reader can assume that the process probably starts with activity *I* and ends with *S*.

Foster communication. As already pointed out, our approach aims at fostering the communication between DE and MB. Wrapping up, we expect testcases: (i) to act as communication medium between DE and MB that serves as basis for discussion, (ii) structure their dialogue; and (iii) allow to focus on the modeling task, as testcases provide a way to automatically ensure that existing behavior is not touched when changing the model.

Support schema evolution. Because design is redesign [34], during schema evolution, the same principles as for process specification can be applied. The only difference, however, is that DE and MB already have a set of testcases as starting point that is extended as the process model is re-engineered.

The use of automated testcases is also beneficial for supporting schema evolution. First, *existing* testcases ensure that desired behavior is preserved by schema evolution, that is, no unwanted behavior is introduced (cf. regression testing, Section 5.1.1). Second, the specification of new testcases capturing the behavior to be introduced helps the modeler to determine which constraints need to be changed, addressing the maintainability issues discussed in Section 4.2. Similar to the specification of new process models, the first step consists of specifying a testcase that defines the behavior to be introduced/changed. Afterwards, the MB iteratively refines the testcase, creates new testcases, and adapts the model until the desired solution is finally approached. The automated nature of testcases ensures that neither requirements are forgotten, nor new requirements contradict existing ones, allowing DE and MB to focus on the requirement elicitation and the modeling task.

5.3.2. Operation and evaluation. Whereas declarative processes provide a high degree of flexibility [14], deviations from the process model can occur nevertheless. In *DECLARE* [14], for instance, it is possible to specify *optional* constraints, which can be violated during process execution. These deviations are usually documented using plain text. However, when deviations occur frequently, it is desirable to ensure that deviations are incorporated during schema evolution [13].

To support the evolution of business processes over time, we propose to capture each deviation in the form of a testcase. The execution trace of the current process instance, in combination with a textual description, can directly be transformed into a testcase. The user, who deviated, is thereby enabled to document the deviation in a form that can directly be used to guide the upcoming schema evolution. This means, when redesigning the process schema, the MB runs all testcases for the respective schema. If all testcases, including the testcases specified in course of deviations, pass, the MB will know that the new process schema version also supports the needs of users who deviated. Otherwise, testcases that fail will be discarded if the discussion between DE and MB reveals that respective behavior should not be supported in the new process model version.

Consider the process depicted in Figure 2, which allows for the submission of a paper only, resubmission is not supported. If a user needed to resubmit a paper, he would deviate from the process

DOI: 10.1002/smr

by inserting and executing an activity *Resubmit Paper*. In order to assure that the next version of the process model includes this exceptional case—or at least that the exceptional case is discussed during the schema evolution—the user creates a new testcases using the execution trace of the current process instance and a textual description to record the reason for the deviation.

5.3.3. Testcases and the process life cycle. So far, the implications on the different phases of the declarative process life cycle have been pointed out. Whereas our approach covers all phases—with focus on process design and deployment—it should be emphasized that support is not provided in isolation for each phase. More specifically, testcases provide a mean of communication and documentation throughout the process life cycle. Starting in the design phase, they aim at improving the understandability of declarative process models and foster communication between DE and MB. Moving to the phase of process deployment, the automated nature of testcases provides support for the validation of the process. During process operation, testcases can be used to document process deviations. And, again starting at the phase of process design, testcases specified during process operation provide a valuable starting point for schema evolution. Thus, it becomes apparent that testcases are neither restricted to a single phase of the process life cycle nor to a single iteration of the life cycle. Rather, testcases flow through possibly multiple iterations of the process life cycle, providing information that cannot be explicitly specified by declarative process models in isolation.

The life cycle of testcases is illustrated in Figure 10. Testcases are primarily defined during process specification (1) and can then directly be used for process testing, that is, validation (2). During process execution, new testcases may be created to document deviations (TC_{n+1} ...TC_o, cf. Figure 10—3). These testcases, in turn, can be used as input for schema evolution (1).

6. LIMITATIONS

A limitation of the work presented in this paper is the missing empirical evaluation. However, promising results about the adapted concepts from software engineering [30,31] and the similarity of business process modeling and software development [35] indicate that beneficial effects for Business Process Management can be expected. To further investigate this open issue, an empirical evaluation is already in progress (cf. Section 8).

Furthermore, the described framework focuses on declarative business process modeling languages only; the applicability to imperative languages is questionable. As testcases and imperative languages both focus on sequential information, the adoption of testcases does not help to balance the sequential/circumstantial information mismatch (cf. Section 4.1). More likely, the introduction of circumstantial information (e.g., constraints) will help to facilitate model understanding and validity of imperative process models (cf. [36]).

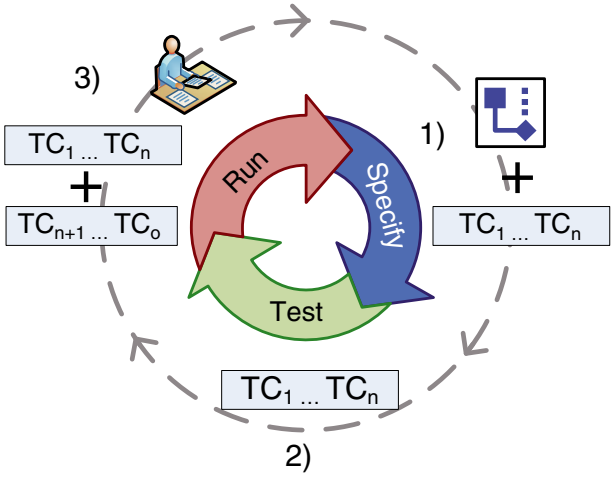

Figure 10. Process and testcase life cycle.

DOI: 10.1002/smr

In addition, the described approach addresses the control flow perspective only. However, for a comprehensive testing support, the data perspective, security perspective, and resource perspective should be taken into account, too. However, this issue can be tackled easily by providing respective fixtures. Thus, this limitation should be rather seen as a desirable extension than an actual limitation.

Finally, problems inherent to TDD and AAT also apply to TDM. For instance, the specification and maintenance of testcases require additional time. Whereas this demand is also confirmed by studies in software engineering [30,31], the same studies also stress gains in terms of quality, and subsequent cost reductions outweigh this investment. In addition, DE and MB specify testcases and model together, thus for applying TDM, two experts *willing to collaborate* are required. However, to what extent problems from TDD and AAT translate to TDM still has to be investigated.

7. RELATED WORK

Most notably is the work of Ly *et al*. [36], which also focuses on validation; but instead of declarative, adaptive process management systems are targeted. Interestingly, with respect to the classification of sequential and circumstantial information, their setting describes the opposite of the setting of this work. Their process models exhibit mostly sequential information, whereas the adopted measure to improve validity relies on circumstantial information (constraints). Thus, the aim of improving validity is found in both approaches, however, for entirely different settings.

With respect to process validity, work in the area of process compliance checking should be mentioned, for example, [37,38]. In contrast to our work, understandability of declarative languages is not of concern; the focus is put on imperative languages.

Another interesting stream of research is the verification of declarative process models. With proper formalization, declarative process models can be verified using established formal methods [14]. Depending on the concrete approach, a priori (e.g., absence of deadlocks) [14] or a posteriori (e.g., conformance of the execution trace) [39], checks can be performed. Whereas these approaches definitively help to improve the syntactical correctness and provide semantical checks a posteriori, they do not address understandability and maintainability issues.

Related work comes also from the area of process mining, where algorithms are used to extract declarative process models from execution traces [40], thereby facilitating the creation of declarative process models. However, these approaches do not help to understand the created models.

The work of Weber *et al*. [13] is also relevant with respect to the documentation of process deviations. They propose an approach called *ProCycle* to capture process deviations using Case Based Reasoning techniques. However, unlike the work presented in this paper, ProCycle focuses on imperative processes.

Finally, the work of Schonenberg *et al*. [41] describes methods for guiding the user through the execution of declarative processes. Like in our work, understandability is a key concept; however, the focus is put on the phase of process operation only.

8. SUMMARY AND OUTLOOK

The need for flexible PAISs is clearly acknowledged in the Business Process Management research community. Consequently, recent interest in declarative processes can be observed, as they promise a high degree of flexibility. However, the potential of current declarative approaches is impeded by deficiencies in understandability and maintainability. In particular, the validation of respective models is still an open issue.

This paper proposes an approach toward better understandability and maintainability of declarative processes by adopting well-established techniques from the domain of software engineering. More specifically, we adopt the idea of TDD to interweave the activities of process specification and process testing. Thereby, testcases balance the circumstantial/sequential information mismatch as well as improve understandability by dispensing with hard mental operations and removing hidden dependencies. In addition, testcases allow for reading declarative process models chunk-wise and propose a way of reading

DOI: 10.1002/smr

process models. Furthermore, as testcases are also understandable by the DE, they foster communication between DE and MB by providing a basis for discussion. During process execution, testcases, in turn, help to document the reasons for process deviations and ensure that respective deviations can be easily considered in schema evolution. When the process model is adapted in the course of schema evolution, testcases ensure that the adaptions do not interfere with existing requirements. In addition, testcases help to identify which part of the model needs to be changed.

Further work focuses on the empirical evaluation of the elaborated concepts. A prototype facilitating TDM supporting the described concepts and providing a simple calendar-like fixture is currently being implemented, and possible scenarios for empirical studies are being investigated. In addition, we will evaluate how the TDM tool can be best integrated into the activity of process specification. For supporting efficient specification of testcases, for instance, we assume that a tight integration of testcase tools and process modeling tools is needed.

REFERENCES

1. Lenz R, Reichert M. IT support for healthcare processes – premises, challenges, perspectives. *Data & Knowledge Engineering* 2007; **61**(1):39–58.
2. Poppendieck M, Poppendieck T. *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley Professional, 2006.
3. Dumas M, van der Aalst WM, ter Hofstede AH. *Process Aware Information Systems: Bridging People and Software Through Process Technology*. Wiley-Interscience, 2005.
4. Weske M. *Business Process Management: Concepts, Methods, Technology*. Springer, 2007.
5. Reichert M, Dadam P. ADEPTflex: supporting dynamic changes of workflow without losing control. *Journal of Intelligent Information Systems* 1998; **10**(2):93–129.
6. Weske M. Workflow management systems: formal foundation, conceptual design, Implementation Aspects. *PhD thesis*, University of Münster, 2000.
7. van der Aalst WMP, Weske M. Case handling: a new paradigm for business process support. *Data & Knowledge Engineering* 2005; **53**(2):129–162.
8. Pesic M, Schonenberg H, Sidorova N, van der Aalst W. Constraint-based workflow models: change made easy. *Proc. CoopIS '07*, 2007; 77–94.
9. Müller D, Reichert M, Herbst J. A new paradigm for the enactment and dynamic adaptation of data-driven process structures. *Proc. CAiSE '08*, 2008; 48–63.
10. Sadiq SW, Orlowska ME, Sadiq W. Specification and validation of process constraints for flexible workflows. *Information Systems* 2005; **30**(5):349–378.
11. Weber B, Reichert M, Rinderle S. Change patterns and change support features – enhancing flexibility in process-aware information systems. *Data & Knowledge Engineering* 2008; **66**(3):438–466.
12. Wainer J, Bezerra F, Barthelmess P. Tucupi: a flexible workflow system based on overridable constraints. *Proc. SAC '04*, 2004; 498–502.
13. Weber B, Reichert M, Rinderle-Ma S, Wild W . Providing integrated life cycle support in process-aware information systems. *International Journal of Cooperative Information Systems* 2009; **18**(1):115–165.
14. Pesic M. Constraint-based workflow management systems: shifting control to users. *PhD thesis*, TU Eindhoven, 2008.
15. Weber B , Reijers HA, Zugal S, Wild W. The declarative approach to business process execution: an empirical test. *Proc. CAiSE '09*, 2009; 270–285.
16. Beck K. *Test Driven Development: By Example*. Addison-Wesley, 2002.
17. Mugridge R, Cunningham W. *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall, 2005.
18. Hoppenbrouwers SJ, Proper EH, van der Weide TP. Formal modelling as a grounded conversation. *Proc. LAP'05* 2005; 139–155.
19. van Bommel P, Hoppenbrouwers S, Proper E, van der Weide T. Exploring modelling strategies in a meta-modelling context. *Proc. OTM '06*, 2006; 1128–1137.
20. Hoppenbrouwers SJ, Lindeman L, Proper EH, Capturing modeling processes – towards the MoDial Modeling Laboratory. *Proc. OTM '06* 2006; 1242–1252.
21. Rittgen P. Collaborative modeling – a design science approach. *Proc. HICSS '09*, 2009; 1–10.
22. Fahland D, Mendling J, Reijers H, Weber B, Weidlich M, Zugal S. Declarative vs. imperative process modeling languages: the issue of maintainability. *Proc. ER-BPM '09*, 2009; 65–76.
23. Fahland D, Mendling J, Reijers HA, Weber B , Weidlich M, Zugal S. Declarative versus imperative process modeling languages: the issue of understandability. *Proc. EMMSAD '09*, 2009; 353–366.
24. Green TR. Cognitive dimensions of notations. *Proc. BCSHCI '89*, 1989; 443–460.
25. Green TR, Petre M. Usability analysis of visual programming environments: a cognitive dimensions framework. *Journal of Visual Languages & Computing* 1996; **7**(2):131–174.

DOI: 10.1002/smr

26. Miller G. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review* 1956; **63**(81–97).
27. Gemino A, Wand Y. Complexity and clarity in conceptual modeling: comparison of mandatory and optional properties. *Data & Knowledge Engineering* 2005; **55**(3):301–326.
28. Zugal S. Agile versus plan-driven approaches to planning – a controlled experiment. *Master's Thesis*, University of Innsbruck, October 2008.
29. Sadiq S, Sadiq W, Orlowska M , Pockets of flexibility in workflow specification. *Proc. ER ' 01*, 2001; 513–526.
30. Canfora G , Cimitile A, Garcia F, Piattini M, Visaggio CA. Evaluating advantages of test driven development: a controlled experiment with professionals. *Proc. ISESE '06*, 2006; 364–371.
31. George B, Williams L. A structured experiment of test-driven development. *Information and Software Technology* 2004; **46**(5):337–342.
32. Agrawal H, Horgan JR, Krauser EW, London S. Incremental regression testing. *Proc. ICSM '93*, 1993; 348–357.
33. Lanz A, Weber B, Reichert M. Time patterns for process-aware information systems: a pattern-based analysis – revised version. *Technical Report*, University of Ulm, 2009.
34. Gilmore DJ, Green TR. Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies* 1984; **21**(1):31–48.
35. Osterweil LJ. Software processes are software too. *Proc. ICSE '87*, 1987; 2–13.
36. Ly LT, Rinderle S, Dadam P. Integration and verification of semantic constraints in adaptive process management systems. *Data and Knowledge Engineering*, 2008; **64**(1):3–23.
37. Awad A, Decker G, Weske M, Efficient compliance checking using BPMN-Q and temporal logic. *Proc. BPM '08*, 2008; 326–341.
38. Awad A, Smirnov S, Weske M. Resolution of compliance violation in business process models: a planning-based approach. *Proc. OTM '09*, 2009; 6–23.
39. van der Aalst WMP, de Beer HT, van Dongen B. Process mining and verification of properties: an approach based on temporal logic. *Proc. OTM '05*, 2005; 130–147.
40. Lamma E, Mello P, Montali M, Riguzzi F, Storari S. Inducing declarative logic-based models from labeled traces. *Proc. BPM '07*, 2007; 344–359.
41. Schonenberg H, Weber B, van Dongen B, van der Aalst WMP. Supporting flexible processes through recommendations based on history. *Proc. BPM '08*, 2008; 51–66.

AUTHORS' BIOGRAPHIES

Stefan Zugal is a Ph.D. candidate at the University of Innsbruck (Austria). Stefan is a member of Quality Engineering (QE) Research Group and member of the Research Cluster on Business Processes and Workflows at QE. Stefan received his M.Sc. degree from the Department of Computer Science, University of Innsbruck in 2008. His main research interests are declarative business process models and the understandability of business process models. Stefan has published more than 20 refereed papers in international journals, conferences and workshops.

Jakob Pinggera is a Ph.D. candidate at the University of Innsbruck (Austria). Jakob is a member of Quality Engineering (QE) Research Group and member of the Research Cluster on Business Processes and Workflows at QE. Jakob received his M. Sc. degree from the Department of Computer Science, University of Innsbruck in 2009. His main research interest is the creation of business process models. Jakob has published more than 20 refereed papers in international journals, conferences and workshops.

DOI: 10.1002/smr

Barbara Weber is an associate professor at the University of Innsbruck (Austria). Barbara is a member of the Quality Engineering (QE) Research Group and head of the Research Cluster on Business Processes and Workflows at QE. Barbara holds a Habilitation degree in Computer Science and Ph.D. in Economics from the University of Innsbruck. Her main research interests are agile and flexible processes, integrated process lifecycle support, intelligent user support in flexible systems and process modeling. Barbara has published more than 70 refereed papers, for example, in Data & Knowledge Engineering, Computers in Industry, Science of Computer Programming, Enterprise Information Systems and IET Software. Moreover, Barbara is organizer of the successful BPI workshop series

DOI: 10.1002/smr