CSRD (2009) 23: 99–113
DOI 10.1007/s00450-009-0057-9

SPECIAL ISSUE PAPER

Declarative workflows: Balancing between flexibility and support

W. M. P. van der Aalst · M. Pesic · H. Schonenberg

Received: 30 May 2008 / Accepted: 12 January 2009 / Published online: 10 March 2009
© The Author(s) 2009. This article is published with open access at Springerlink.com

Abstract Today's process-aware information systems tend to either support business processes *or* provide flexibility. Classical workflow management systems offer good process support as long as the processes are structured and do not require much flexibility. Information systems that allow for flexibility have a tendency to lack process-related support. If systems offer guidance, then they are typically also inclined to "enforce guidelines" and are perceived as inflexible. Moreover, implementing flexible systems is far from trivial. This paper will show that using a more declarative approach can assist in a better balance between flexibility and support. This is demonstrated by presenting the *Declare* framework that aims to take care of the full spectrum of flexibility while at the same time supports the user using recommendations and other process-mining-based diagnostics.

Keywords Workflow management · Business Process Management · Flexibility · Process mining

1 Introduction

Process-aware information systems (PAISs) support operational business processes by combining advances in information technology with recent insights from management science [21]. Workflow Management Systems (WFMSs) are typical examples of such systems. However, many other types of information systems are also "process aware" even if their processes are hard-coded or only used implicitly (e.g., ERP systems). The shift from data orientation to process orientation has increased the importance of PAISs. Moreover, advanced analysis techniques ranging from simulation and verification to process mining and activity monitoring allow for systems that support process improvement in various ways.

When backing business processes with IT there is a difficult trade-off to be made. On the one hand, there is a desire to control processes and to avoid incorrect or undesirable executions of these processes. On the other hand, users want flexible processes that do not constrain them in their actions. This apparent paradox has limited the application of WFMSs thus far, since, as indicated by many authors, WFMSs are too restrictive and have problems when dealing with change [5].

Many approaches have been proposed to resolve the apparent paradox illustrated by Fig. 1. Some of them try to avoid change, e.g. by generating implicit alternative paths [8, 12], or by deferring the selection of the desired behavior [11]. Others allow for changing the model for a single instance and/or changing a process model while migrating all instances [16, 22, 40, 48]. The migration of process instances from one model to another introduces many interesting problems [5, 16, 40, 48]. For example, the "dynamic change bug" originally described in [22] shows that it may be impossible to put the process instance into a suitable state of the new model without skipping or repeatedly executing activities.

The goal of the paper is to demonstrate that *by using a declarative approach it is possible to balance between support or flexibility*. Traditional approaches tend to use procedural process models to explicitly (i.e., step-by-step)

W. M. P. van der Aalst (✉) · M. Pesic · H. Schonenberg
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: w.m.p.v.d.aalst@tue.nl

M. Pesic
e-mail: m.pesic@tue.nl

H. Schonenberg
e-mail: m.h.schonenberg@tue.nl

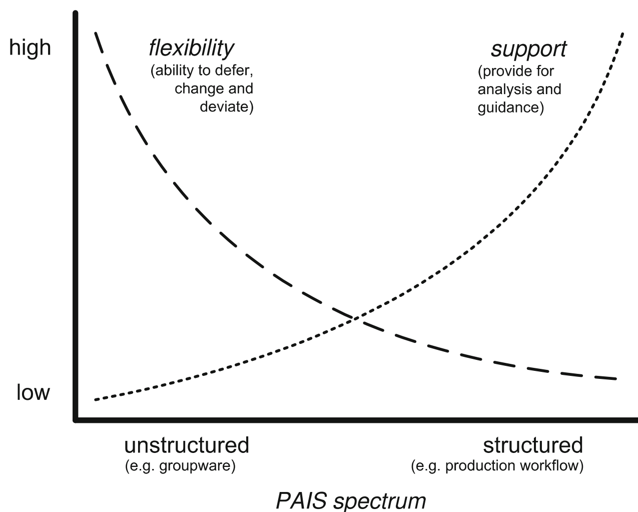

Fig. 1 Today's PAIS spectrum: Systems offer support or flexibility but not both

specify the execution procedure. The declarative approach presented in this paper is based on *constraints*, i.e., anything is possible as long as it is not explicitly forbidden. Constraint-based models, therefore, implicitly specify the execution procedure by means of constraints: any execution that does not violate constraints is possible. We have implemented the *Declare* framework that provides for multiple constraint-based languages. By using such a declarative approach, different types of flexibility become possible as shown in the remainder.

This paper is organized as follows. Section 2 presents a small "taxonomy of flexibility" using three basic types of flexibility. Moreover, the different types of flexibility are positioned in the overall life-cycle of a process. Using this as a basis, Sect. 3 discusses the support ultimately desired from a PAIS. Section 4 introduces the *Declare* framework and explains the concept of constraint-based workflow models. Section 5 relates *Declare* to the initial taxonomy presented in Sect. 2. Section 6 evaluates the support provided by *Declare* and 7 discusses the limitations of the current approach and implementation. Section 8 presents related work and Sect. 9 concludes the paper.

2 Flexibility

Flexibility has become one of the major research topics in the area of workflow management [5]. Today's WFMSs and many other PAISs have problems providing for flexibility [33, 47]. As a result, these systems are not used to support dynamically changing business processes or the processes are supported in a rigid manner, i.e., changes are not allowed or handled outside of the system. These problems have been described and addressed extensively in literature [8, 10, 22, 35, 38, 40, 42, 48]. Nevertheless, many problems related to flexibility remain unsolved.

In this section, we provide an overview of the different types of flexibility using a taxonomy. This taxonomy is based on [43]. See also [5, 9, 23, 26, 33, 39, 42, 47] for other classifications of flexibility.

To start, let us identify the different *phases* of a process (instance) in the context of a WFMS:

- *Design time.* At design time a generic process model is created. This model cannot be enacted because it is not connected to some organizational setting.
- *Configuration time.* At configuration time a generic model is made more specific and connected to some organizational context which allows it to be instantiated.
- *Instantiation time.* At instantiation time a process instance is created to handle a particular case (e.g., a customer order or travel request).
- *Run time.* At run time the process instance is executed according to the configured model. The different activities are being enacted as specified.
- *Auditing time.* At auditing time the process instance has completed, however, its audit trail is still available and can be inspected and analyzed.

Flexibility plays a role in most of these phases. At design time some modeling decisions can be postponed to run time. At run time one can decide to deviate from the model and at instantiation time one can change the process model used for the particular instance. When it comes to flexibility, we identify three *flexibility mechanisms*:

- *Defer, i.e., decide to decide later.* This flexibility mechanism deliberately leaves freedom to maneuver at a later phase. Examples are the use of a declarative process modeling language which allows for the "under specification" of processes and the use of late-binding, i.e., the process model has a "hole" that needs to be filled in a later phase.
- *Change, i.e., decide to change model.* Most researchers have addressed flexibility issues by allowing for change. Decisions made at an earlier phase may be revisited. For example, for premium customers the process may be adapted in an ad-hoc manner. The change may refer to the model for a single instance (*ad-hoc change*) or to all future cases (*evolutionary change*). In both cases, a change can create inconsistencies. For example, there may be process instances that cannot be migrated properly when conducting an evolutionary change [2, 16, 22, 38, 40, 48].
- *Deviate, i.e., decide to ignore model.* The third mechanism is to simply deviate from the model, e.g., activities are skipped even if the model does not allow for this to happen. In many environments it is desirable that people

Fig. 2 Classification of the different types of flexibility based on the phase and mechanism

	defer (decide to decide later)	change (decide to change model)	deviate (decide to ignore model)
design time	e.g., defer to run-time by using late binding or declarative modeling	N/A	N/A
configuration time	e.g., defer configuration decisions	e.g., remodel parts of the process at configuration time	e.g., violate a configuration constraint
instantiation time	e.g., defer the selection of parameters or process fragments	e.g., modify model for a particular customer	N/A
run time	N/A	e.g., change model for running instance or migrate instance to new model	e.g., skip or redo a task while this is not specified
auditing time	N/A	N/A	N/A

are in control, i.e., the system can only suggest activities but not force them to happen.

Figure 2 relates the two dimensions just mentioned. Based on the different phases and the three mechanisms, different types of flexibility are classified. Note that we did not mention any examples of flexibility at auditing time. After the process instance completes, deferring, changing, or deviating is not possible anymore or would imply fraud. Also note that the actual mechanism in different cells can be the same, e.g., deferring a choice to run-time can be done at design time, configuration time and instantiation time.

Figure 2 can be used to characterize the support for flexibility of a concrete WFMS. Unfortunately, today's systems typically provide for only a few forms of flexibility. This is limiting the applicability of PAISs since within one organizations different forms of flexibility may be required.

In the remainder, we will show that it is relatively easy to care for the various types of flexibility by using a declarative approach. However, as indicated in the introduction there may be a trade-off between flexibility and support. Therefore, we first elaborate on the support desired.

3 Support

It is relatively easy to develop systems that that are extremely flexible while providing little process support. A nice example is an e-mail program like Outlook. Outlook is not forcing users to send e-mails at particular times or in a particular order. So it can be seen as a very flexible system as it allows users to execute activities (e.g., sending e-mails) in any way they want. However, Outlook is providing hardly any support and has no knowledge of the processes users are involved in. In this paper, we only consider PAISs where some process model is used during enactment. In fact, we focus on WFMSs. These systems use an explicit process design as a starting point. This design is used at run time to control and support users. In this section we elaborate on support in the main two phases depicted in Fig. 2: *design time* and *run time*.

3.1 Design-time support

At design time a model is constructed that will be used to enact the workflow at run time. Besides providing a good editor to design the workflow process, the WFMS should also provide good analysis tools. Here we elaborate on two types of analysis: *verification* and *performance analysis*.

Verification The goal is to design processes that are correct, e.g., there should not be potential deadlocks, etc. Using verification techniques it is possible to discover problems before the design is enacted. We distinguish between *syntactical* verification and *semantical* verification. Syntactical verification aims at the discovery of errors that do not require domain analysis. For example, a model that can deadlock is incorrect independent of the intent of the process. Similarly, it is not acceptable to have processes that may not terminate or where some activities can never be executed. The notion of *soundness* [1] is a typical example of a correctness property checked through syntactical verification. Semantical verification aims at the analysis of properties that can only be formulated based on domain knowledge. For example, when two drug types are incompatible, then it should not be allowed to make a model in such a way that an instance (i.e., a patient) can receive both drug types during execution. Semantical constraints over processes express domain knowledge. Such knowledge is vital when

providing more flexibility, e.g., end users should not modify process models such that essential semantical constraints get violated. See [31] for examples of semantical correctness notions relevant when adapting processes.

Performance analysis Process models should be correct, but also the performance of a process is relevant. The flow time may be too long or utilization levels and response times can be unacceptable. To predict such situations and to support process improvement, techniques for performance analysis can be used at design time. For example, simulation can be used to predict key performance indicators and to evaluate redesigns. In this paper, we will not elaborate on performance analysis. However, it is important to realize that such functionality may be important when justifying the use of a WFMS.

3.2 Run-time support

At run time, the model made at design time is enacted. Assuming that the model is correct and has a good performance, we distinguish various types of run-time support.

Enforcing correct execution The WFMS should use the model made at design time to enforce the correct execution, i.e., work-items should be offered based on the process model and the system should prevent the execution of activities that are not enabled according to the model.

Recommending effective execution The focus of classical WFMSs is on control, i.e., the system decides on the ordering of activities. When systems allow for more flexibility the decision freedom increases. For example the defer flexibility type ("decide to decide later") may move choices from design time to run time. To support users when making decisions, the system may provide recommendations. These recommendations may depend on explicit domain knowledge. However, it is also possible to learn good strategies through process mining and then use these for guidance [44]. For example, it is possible to analyze choices made in the past with respect to flow time and then suggest for new cases the "fastest path" through the process. The user can still decide to ignore the recommendation, but at least some decision support is given.

Monitoring process instances The WFMSs should also support the monitoring of process instances. It should be possible to follow specific cases and to look at the workflow at a more aggregate level. Users should be able to get insight into the status of a single case, a group of cases, and the whole workflow.

Learning from processes The more flexible the system is, the more variations are possible when conducting work. Therefore, it is interesting to use process mining [7] to discover the real processes taking place. Starting point for process mining are the event logs, i.e., data on the actual execution of activities. The WFMS should support the recording of these events in a systematic manner and the analysis tools of the WFMS should be able to extract knowledge from these logs. The more variability that is possible, the more valuable such analysis is.

Enforcing correct changes Verification is not only relevant at design time, but also at run time when changes are applied. When using the flexibility type change ("decide to change model"), errors can be introduced into the model. Hence, again syntactical and semantical verification need to be provided to ensure the correct operation of the system.

Note that the above summary of required design-time and run-time support is far from complete. We emphasized aspects related to flexibility, e.g., the monitoring of instances becomes more relevant when users are not forced to work in a particular way.

4 *Declare*

The goal of this paper is to show that a more declarative approach to workflow management makes it easier to balance between flexibility and support. We will show that it is easy to provide for a wide variety of flexibility types without loosing key functionalities at design and/or run time.

To illustrate the advantages of declarative workflows we present a concrete framework: *Declare*. *Declare* is a constraint-based WFMS that provides for multiple declarative languages (DecSerFlow [6], ConDec [34], etc.). In fact, *Declare* is extendible and without any programming it can be configured to support additional constraint-based languages. Instead of explicitly defining the ordering of activities in models, *Declare* models rely on constraints to implicitly determine the possible ordering of activities (any order that does not violate constraints is allowed).

Declare can be downloaded from [19]. The architecture of the *Declare* system is shown in Fig. 3. The core of the system consists of the following basic components: *Designer*, *Framework* and *Worklist*. The *Designer* component is used for creating the so-called constraint templates, to design concrete process models, and to verify these model. The *Framework* enacts instances of process models. Moreover, it also conducts ad-hoc changes of running instances. While the *Framework* centrally manages the execution of all instances, each user uses his/her *Worklist* component to access active instances. Also, a user can execute activities in active instances in his/her *Worklist*.

Unlike most of the approaches, which offer a predefined set of constructs for defining dependencies between activities in process models (e.g., sequence, choice, parallelism, loops, etc.), *Declare* uses a customizable set of arbitrary

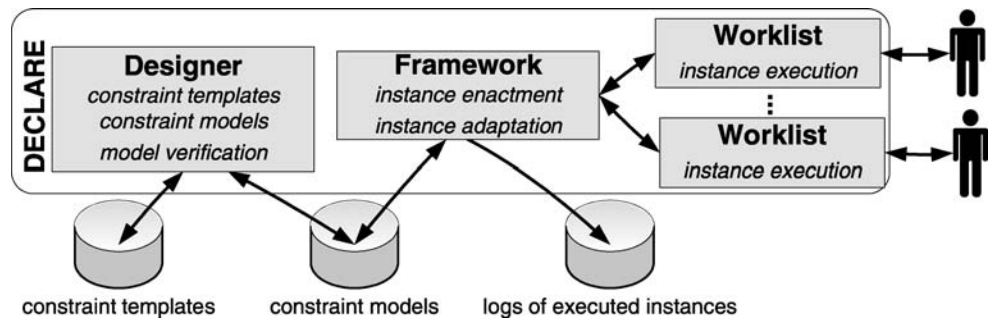

Fig. 3 The architecture of *Declare*

constructs called *constraint templates*. A declarative language can be seen as a collection of constraint templates. This explains why the *Declare* system provides for multiple languages (e.g. DecSerFlow [6] and ConDec [34]). Templates are defined on the system level in the *Designer* component. Each template has (1) a unique name, (2) a graphical representation, and (3) a formal specification of its semantics in terms of Linear Temporal Logic (LTL) [17, 24]. LTL is a special type of logic which, in addition to classical logical operators, uses temporal operators such as: always ($\Box$), eventually ($\Diamond$), until ($\sqcup$), and next time ($\bigcirc$) [17]. Figure 4 shows how the *response* template is defied. This template is graphically represented with a single line between two activities A and B, with a filled circle next to A and a filled arrow next to B. The semantics of the template is given as LTL formula $\Box(A \Rightarrow \Diamond(B))$, i.e., every time activity A is executed, B has to be executed afterwards at least once.

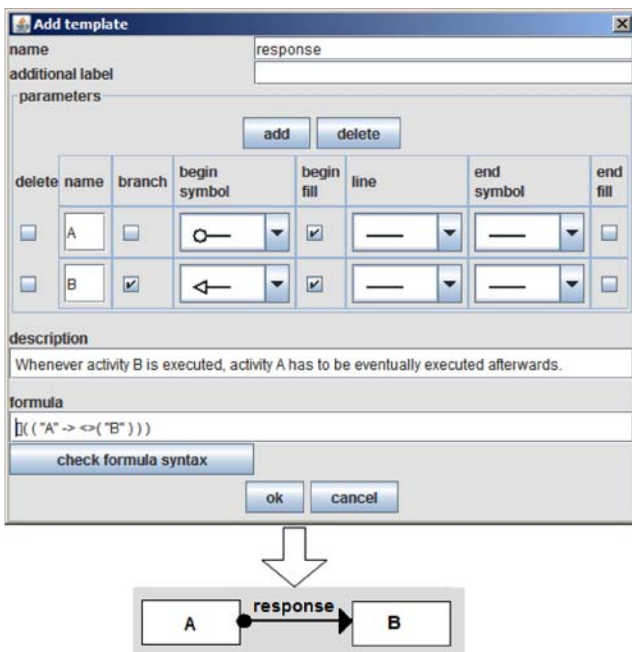

Fig. 4 Constraint template *response*

In this paper, we use ConDec [34] as the modeling language. ConDec can be seen as a collection of constraint templates. Using these templates one can make concrete models. In this paper, we will refer to such a ConDec model as "*Declare* model" or simply "model" because we do not elaborate on the different languages supported by *Declare*.

In the remainder, we use the process for handling a patient at the first aid department in a hospital with a suspicion of a fracture as a running example. Figure 5 shows the model constructed using *Declare*. All activities are depicted as boxes, constraints are depicted by relations (lines) between activities. The model contains *mandatory constraints* (solid lines) and *optional constraints* (dashed lines). For example, the dashed arrow in Fig. 5 shows an application of the *response* template defined in Fig. 6. We use the following fonts to refer to activity (task) names and constraint template names respectively: *task name* and *constraint name*.

Initially, a specialist performs activity *examination* (constraint *init*), if necessary, additional diagnosis is done by *X-ray*. Depending on the absence, presence and type of fracture, there are several types of treatment available, such as *sling*, *fixation*, *surgery* and *cast*. Except for *cast* and *fixation*, which are mutually exclusive (constraint *not co-existence*),

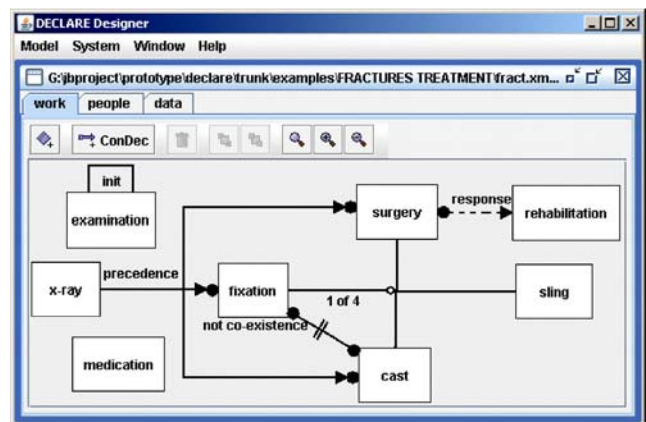

Fig. 5 Declarative model for handling patients using *Declare*

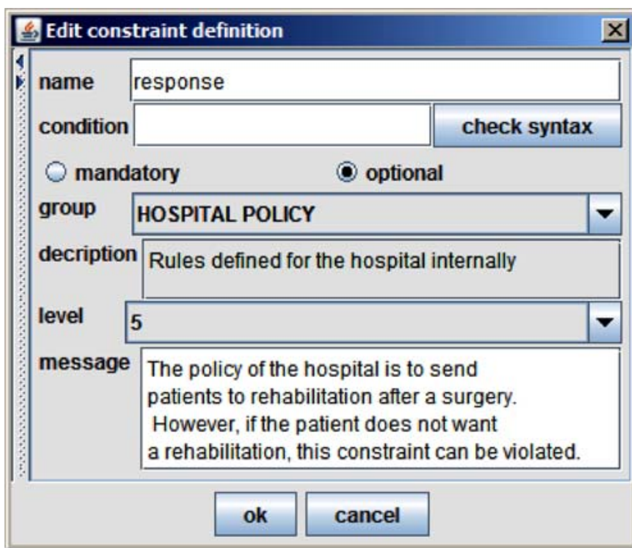

Fig. 6 Defining the optional *response* constraint

the treatments can be given in any combination and each patient receives at least one treatment (*1 of 4* constraint). Additional diagnosis (*X-ray*) is not necessary when the specialist diagnoses the absence of a fracture during *examination*. Without this additional diagnosis, the patient can only receive the *sling* treatment. All other treatments require *X-ray* to rule out the presence of a fracture, or to decide how to treat the fracture (constraint *precedence*). Simple fractures can be treated just by *cast*. For unstable fractures activity *fixation* may be preferred over activity *cast*. For patients who undergo *surgery* the specialist is advised to execute activity *rehabilitation* afterwards (optional constraint *response*). Moreover, the specialist can provide *medication*, e.g., pain killers or anticoagulants, at any stage of the treatment. Also additional examinations and X-rays can be done during the treatment.

Note that *init*, *precedence*, *1 of 4*, and *not co-existence* refer to constraint templates whose semantics are expressed in terms of LTL. Table 1 shows the relation between the constraints shown in Fig. 5, the constraint templates, and LTL. The process should start with *examination*. This constraint is specified using the *init* template. Table 1 shows its definition: $init(A) = A$. Therefore, $init(examination) = examination$. Note that in LTL-terms this means that *examination* should be the current (i.e., first) action. The *precedence* constraint template is defined by the LTL formula $precedence(A, B) = (!B)\ W\ A$, i.e., B should *not* happen before A has happened. Note that W is a temporal operator similar to $\sqcup$ (until). The "weak until" operator W in "$(!B)\ W\ A$" says that A does not have to happen if B never happens. In Fig. 5, the *precedence* constraint template is used with three B's, i.e., $(!(surgery \vee fixation \vee cast)\ W\ X\text{-}ray$ defines the semantics of this particular constraint). This means that the treatments *surgery*, *fixation*, and *cast* all require *X-ray* to rule out the presence of a fracture. However, *X-ray* is not needed if none of the treatment activities (*surgery*, *fixation*, and *cast*) occurs. Table 1 also defines the *1 of 4* and *not co-existence* constraints. $1\text{ of }4(A, B, C, D) = \Diamond(A \vee B \vee C \vee D)$ means that eventually ($\Diamond$) at least one of the four activities should occur. $\text{not coexistence}(A, B) = !((\Diamond A) \wedge (\Diamond B))$ means that it cannot (!) be the case that eventually A occurs ($\Diamond A$) and that eventually B occurs ($\Diamond B$).

The process defined by Fig. 5 allows for many execution paths. Unlike imperative languages, there is no need to include these execution paths explicitly. For example, the mutual exclusion constraint between *cast* and *fixation* is difficult to express in imperative languages, especially since the moment of choice between these two treatments is not fixed. In an imperative language one would need to decide on the moment of choice, specify the loop behavior, and determine the people making these choices. In *Declare* one can simply use the *not-coexistence* constraint with an intuitive graphical notation. In declarative languages only the rules that constrain the behavior need to be specified. Therefore, there is no need to enumerate the execution paths.

Constraint *response* between activities *surgery* and *rehabilitation* is optional as shown by the dashed arrow in Fig. 5. Figure 6 shows the definition of the constraint that is using the *response* template. Note that for optional constraints a level and a warning message can be defined. In this particular case a warning of level "5" is generated when the user is about to violate the constraint.

Figure 7 shows the *Worklist* component containing two active instances (active instances are presented in the list on the left-hand side of the screen). After executing activity *examination*, the user is currently executing activity *medication* for the second process instance. Activities *examination*, *X-ray*, and *medication* are enabled, i.e., can be executed. Activities *surgery*, *fixation*, and *cast* are disabled, i.e., cannot

Table 1 LTL expressions for constraints in Fig. 5

Template formula	Constraint	LTL expression
$init(A) = A$	*init*	*examination*
$precedence(A, B) = (!B)\ W\ A$	*precedence*	$(!(surgery \vee fixation \vee cast)\ W\ X\text{-}ray$
$response(A, B) = \Box(A \Rightarrow (\Diamond B))$	*response*	$\Box(surgery \Rightarrow (\Diamond rehabilitation))$
$1of4(A, B, C, D) = \Diamond(A \vee B \vee C \vee D)$	*1 of 4*	$\Diamond(surgery \vee fixation \vee cast \vee sling)$
$not\ \ coexistence(A, B) = !((\Diamond A) \wedge (\Diamond B))$	*not-coexistence*	$!((\Diamond fixation) \wedge (\Diamond cast))$

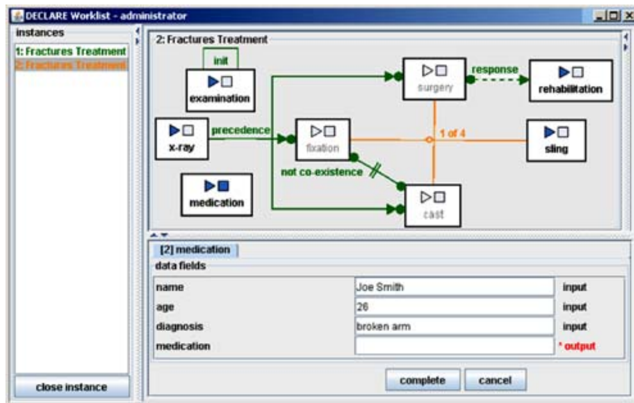

Fig. 7 Executing an instance

be executed (indicated by the gray color) due to the *precedence* constraint (activity *X-ray* is not executed yet in this instance).

The enabling and execution of activities is driven by constraints: Everything that does not violate the mandatory constraints is enabled for execution and all mandatory constraints must be *satisfied* at the end of the instance execution. For an active instance, each constraint is in one of three states: *satisfied*, *temporarily violated*, or *violated*. In the *violated* state there is no possible future that will satisfy the constraint. In the *satisfied* state there is no need to execute more activities to let the constraint evaluate to true. For example, the *init*, *precedence*, and *response* constraints are *satisfied* in the instance presented in Fig. 7. In all other states, the constraint is not (yet) satisfied, but it is still possible to satisfy the constraint by executing the appropriate activities. These states are called *temporarily violated*. For example, constraint *1 of 4* is *temporarily violated* because none of the activities *surgery*, *fixation*, *cast* and *sling* are executed, but executing one of these four activities will bring this constraint into the state *satisfied*.

Optional constraints may become violated. However, this is not possible for mandatory constraints. In order to enforce this, *Declare* constructs an automaton for each constraint [6, 34]. To achieve this we use an approach similar to [17, 24]. Based on the automaton it is possible to see whether the execution of some activity will violate the constraint. Since there is an automaton per constraint, it is easy to see which constraints are violated by which actions. Moreover, *Declare* constructs an overall automaton that is based on the conjunction of all LTL-based constraints. This way it is possible to make sure that mandatory constraints are either in state *satisfied* or state *temporarily violated*. Optional constraints can also be in state *violated*. However, when an optional constraint is about to be violated the system issues a warning (cf. Fig. 6).

So each constraint is translated into a finite state automaton that exactly represents all words (i.e., all executions) that satisfy the corresponding LTL formula. For each process instance, *Declare* creates (1) one automaton for the conjunction of LTL formulas of all constraints in the instance, i.e., the so-called instance automaton, and (2) one automaton for each constraint in the instance. As mentioned before, these automata are used for several purposes when executing the instance. First, the instance automaton is used for driving the execution: executing an activity in the instance triggers one or more transitions in the automaton and, thus, causes the state change of the automaton. Second, the instance automaton is used to determine which activities are enabled: only activities that can be triggered at the current state of the automaton are enabled. Third, the instance automaton is used to determine the state of the instance: if the current state of the automaton is accepting, then the instance is *satisfied*; if the current state of the automaton is non-accepting, then the instance is *temporarily violated*. Fourth, the automaton created for each constraint is used to determine the state of the constraint: if the current state of the automaton is accepting, then the constraint is *satisfied*; if the current state of the automaton is non-accepting, then the constraint is *temporarily violated*. Details about the usage of automata for the execution of instances in *Declare* are out of scope of this paper. We refer the interested reader to [33].

5 Flexibility with *Declare*

In this section we show that *Declare* cares for all three flexibility types introduced in Sect. 2, i.e., (1) *defer*, (2) *change*, and (3) *deviate*. To show that this is indeed the case, we first summarize the main differences with imperative/procedural languages using Fig. 8.

The black oval in Fig. 8b represents the behavioral boundary of a classical (i.e., imperative/procedural) process model that is defined using an "inside out" style of modeling where routing is modeled explicitly. The black thick boundary in Fig. 8c represents the "outside in" style of modeling supported by *Declare* and the use of optional constraints. Comparison of the two approaches in Fig. 8 suggests that a declarative approach indeed allows for more flexibility.

5.1 Defer ("decide to decide later")

Declare facilitates flexibility by the possibility to easily defer choices to run time. Consider for example the *not-coexistence* constraint in Fig. 5. This constraint specifies the mutual exclusion of the two activities *cast* and *fixation*. We only specify that if *cast* is done, then *fixation* cannot occur and vice versa. Hence, with little effort, many choices are deferred to run time. For example, *cast* and *fixation* may

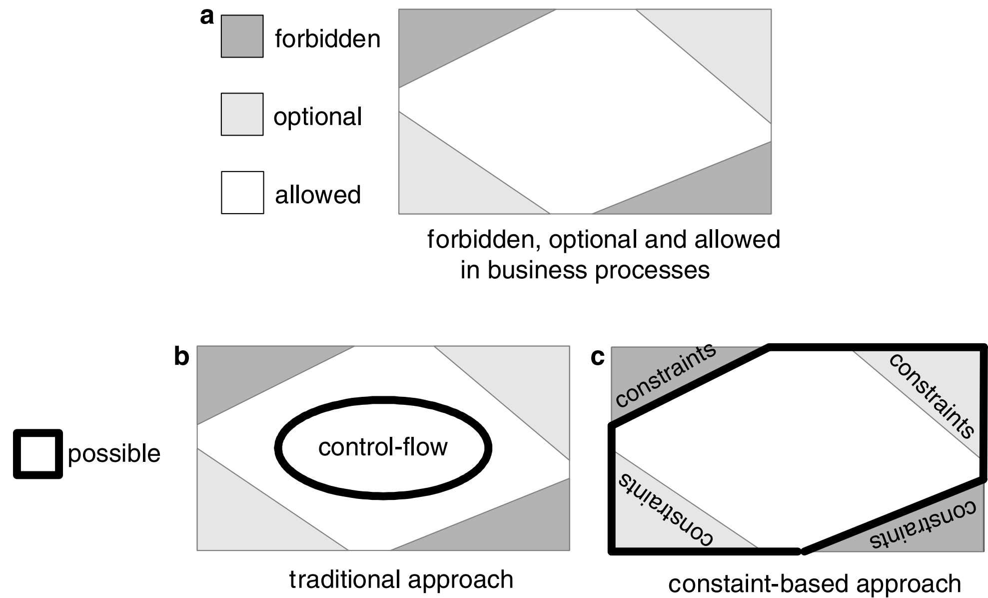

Fig. 8 Mandatory constraints restrict the set of possible behaviors while optional constraints further guide the user. Since traditional approaches explicitly specify the possible behaviors, declarative language tend to be more flexible [33]

both occur multiple times or not at all. The only requirement is that for the same instance they do no both happen. Note that there is no explicit choice between activities *cast* and *fixation*, i.e., there is no activity inserted to make decisions on the number of times each of these activities should be executed. Specifying such a constraint in an imperative language typically results in an "over specification" of the desired behavior, i.e., the user has to explicitly incorporate all execution paths and identify explicit decision points determined at design time.

Figure 5 shows several other constraints that would be difficult to capture in traditional languages without unnecessarily restricting the user at run time. For example, in Fig. 5 a specialist can administer *medication* at any moment. The decision *how often* and *when* this can be done is deferred to run time and not included into the design. The only thing that is specified is that *medication* is an activity the specialist can execute in the fracture process and that there are no constraints on this activity.

These examples show that *Declare* definitely supports the flexibility type "decide to decide later" described in Sect. 2.

5.2 Change ("decide to change model")

Declare allows for instance change at run time to support unforeseen situations or changed circumstances. The change may refer to a single instance (ad-hoc change) or all instances of a given process (evolutionary change). To illustrate the functionality provided by *Declare*, we consider Fig. 5. Suppose that new guidelines would forbid specialists to prescribe the usage of *sling* without the execution of *X-ray*. This can be achieved by adding an additional branch to activity *sling* on the *precedence* constraint as shown in Fig. 9. This branch can be added to the instance in an ad-hoc manner, i.e., during the execution of the instance. In addition, it is possible to request that the change should be applied to all instances (evolutionary change). This implies that all running instances of the model are migrated (if possible) from the model in Fig. 5 to the model in Fig. 9. Moreover, all new instances would start in the new model. All of this is possible in *Declare* illustrating that a declarative approach is able to address many of the problems described in literature [2, 22, 40].

Declare applies the change only to instances that do not become *violated* because of the migration to the new or adapted process, For example, assume that the ad-hoc change presented in Fig. 9 is applied to an instance of the model from Fig. 5 for which activities *examination*, *sling*, and *medication* have been executed. Because the added branch specifies that activity *sling* can be executed only after

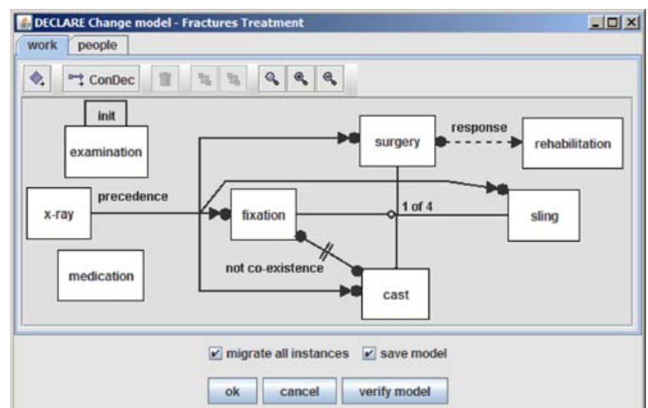

Fig. 9 The initial model is changed by extending the *precedence* constraint. This constraint now also includes activity *sling*

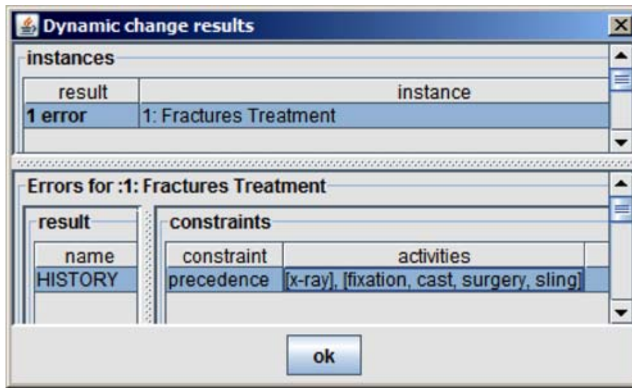

Fig. 10 Report for the ad-hoc change in Fig. 9 showing that an instance cannot be migrated because of a violation of the new *precedence* constraint

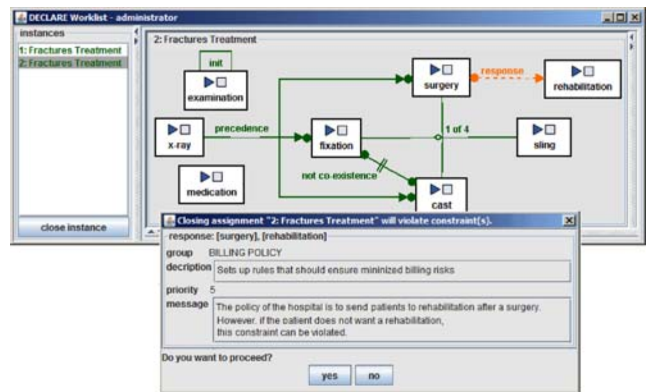

Fig. 11 A warning regarding the possible violation of the optional *response* constraint

activity *X-ray* was executed, applying this change would lead to a violation because activity *sling* is already executed before activity *X-ray*. The instance automaton (cf. Sect. 4) is used to check whether an ad-hoc change is applicable for an instance. The change is applicable if and only if the instance history can be "replayed" on the automaton generated for the new model. If the new model is not consistent with the instance history, then the automata generated for subsets of constraints can be used to determine which minimal combination of constraints in the new model prevents the ad-hoc change. Indeed, Fig. 10 shows that the *precedence* constraint does not allow the change presented in Fig. 9.

5.3 Deviate ("decide to ignore model")

As mentioned earlier, *Declare* allows for optional constraints. In Fig. 5 the *response* constraint is optional as indicated by the dashed line. Figure 6 showed the definition of this constraint indicating the warning that will be given if a users tries to violate this constraint. Figure 8c shows the basic idea behind such optional constraints, the user can deviate into the light-gray part of the figure and thus "decide to ignore model". One can think of optional constraints as guidelines or "soft" constraints. *Declare* allows the designer to set a warning level to indicate the severity of such a deviation. This way *Declare* also provides provides for the third type of flexibility identified in Sect. 2.

Figure 11 shows a warning shown to a user who is about to violate the optional *response* constraint. This constraint indicates that if *surgery* has been executed, then *rehabilitation* has to be executed afterwards. It is not enforced because when the patient is young and mobile and the severity of fracture is minor this is not strictly necessary. The instance presented in Fig. 11 corresponds to a patient for which activity *surgery* is executed but activity *rehabilitation* is not executed yet. Therefore, the *response* constraint is *temporarily violated* (shown using the color orange). If the user would try to close this instance at this moment, this would permanently violate the *response* constraint. If this would be a mandatory constraint, closing the instance would not be possible. However, since the constraint is optional, the user gets an informative warning about the possible violation. Note that the warning shown in Fig. 11 is indeed the warning defined in Fig. 6.

6 Support by *Declare*

This section presents the support provided by *Declare* using the topics identified in Sect. 3. First, we discuss the design-time support offered by *Declare*. Then, we elaborate on *Declare*'s run-time support.

6.1 Design-time support

6.1.1 Verification

Constraints in *Declare* models can interfere in subtle ways. This can cause *two types of errors*. First, it might be that an activity in the model can never be executed. We refer to such an activity as a *dead activity*. Second, it might be that constraints are conflicting, i.e., it is not possible to execute a model in such a way that all mandatory constraints are *satisfied*. We refer to this kind of error as a *conflict*. Each dead activity and conflict is caused by a certain combination of mandatory constraints in the model (i.e., the so-called *cause of error*). Through verification, *Declare* can automatically detect the existence of dead tacks and conflicts while explicitly indicating the combination of mandatory constraints that cause each error. Consider, for example, the model shown in Fig. 12. This model contains three errors. First, *fixation* is dead because of the *non co-existence* constraint and the *1..** constraint on activity *cast*, i.e., *cast* should be performed at least once and therefore *fixation* can never occur. Sec-

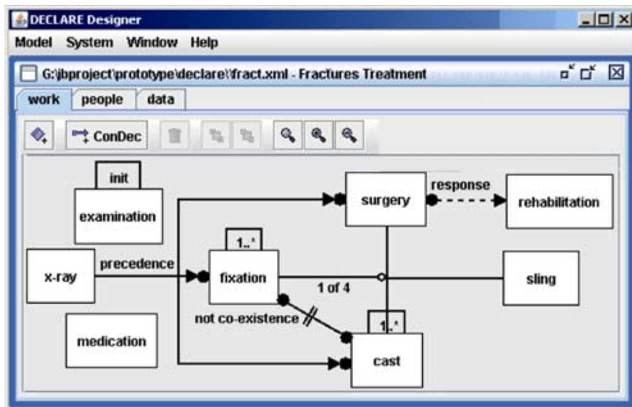

Fig. 12 A model with a conflict

a

Verification result - 3 errors were detected.

result

name	
DEAD ACTIVITY	[cast]
CONFLICT	
DEAD ACTIVITY	[fixation]

constraints

constraint	activities	condition
not co-existence	[fixation], [cast]	
1..*	[fixation]	

ok

activity *cast* is dead

b

Verification result - 3 errors were detected.

result

name	
DEAD ACTIVITY	[cast]
CONFLICT	
DEAD ACTIVITY	[fixation]

constraints

constraint	activities	condition
not co-existence	[fixation], [cast]	
1..*	[fixation]	
1..*	[cast]	

ok

a conflict

c

Verification result - 3 errors were detected.

result

name	
DEAD ACTIVITY	[cast]
CONFLICT	
DEAD ACTIVITY	[fixation]

constraints

constraint	activities	condition
not co-existence	[fixation], [cast]	
1..*	[cast]	

ok

activity *fixation* is dead

Fig. 13 (**a**) Activity *cast* is dead; (**b**) a conflict; (**c**) activity *fixation* is dead. Verification result for the model in Fig. 12

ond, *cast* is dead because of the *non co-existence* constraint and the *1..** constraint on activity *fixation*. Third, there is a conflict because the combination of the *non co-existence* constraint, the *1..** constraint on activity *fixation*, and the *1..** constraint on activity *cast* leads to a contradiction.

The three errors in the Fig. 12 and the combination of constraints that cause them can be detected using the automata generated from LTL specifications of constraints. In order to find the cause of an error, an automaton is generated for the conjunction of all subsets of mandatory constraints. If an error is found in a minimal subset of constraints, then the error is reported and the root cause is given as a subset of constraints. Because transitions of an automaton enable the execution of activities, a dead activity is detected as an activity that cannot be triggered by any transition of the automaton. Because a model exhibiting a conflict can never be executed such that all related constraints are satisfied, this error is detected by checking whether the corresponding automaton is empty, i.e., if it has no states and no transitions, then the model has a conflict. *Declare* uses this automata-based method to detect errors and the corresponding root causes. Figure 13 shows the verification report of *Declare* for the model presented in Fig. 12. Indeed, the three errors and combinations of constraints that cause them are detected.

Note that dead activities and conflicts correspond to syntactical errors (cf. Sect. 3.1). Using LTL it is also easy to check for semantical errors. As long as the (un)desired property can be expressed in terms of LTL, classical model checking techniques can be used to find the error. Note that this is not (yet) implemented in *Declare*. However, syntactical errors are verified both at design time and run time (see also Sect. 6.2).

6.1.2 Performance analysis

Declarative models are less suitable for performance analysis. The reason is that many execution paths are possible, i.e., there is a tendency to defer decisions to run time. Hence, arbitrary runs of *Declare* may not be representative for actual user behavior. In a procedural language, the users are driven by the process model and it is easy to define the behavior of users as their degree of freedom is limited. This implies that *Declare* could only support performance analysis if a lot of information is added about the expected behavior of users. Therefore, no support for performance analysis at design time is added.

6.2 Run-time support

6.2.1 Enforcing correct execution

As explained in Sect. 4, *Declare* constructs an instance automaton based on the conjunction of all mandatory constraints. This automaton forces the users to stay within the boundaries of the model. By adding lots of restrictive constraints, the behavior of the *Declare* engine becomes similar to that of a classical workflow engine. However, it is also possible to have just a few mandatory constrains and thus only force the users with respect to these constraints. The run-time view of *Declare* shows the activities that are enabled and indicates the status of each constraint (cf. Fig. 7).

An activity is not enabled if its execution would permanently violate a mandatory constraint.

6.2.2 Recommending effective execution

The link between the process mining tool *ProM* [4] and *Declare* enables the guidance of users through recommendations [44]. *Declare* records all events taking place, e.g., starting or completing an activity. These recorded events are shared with *ProM*. *ProM* is able to extract knowledge from *Declare*'s event logs using process mining techniques. For example, as shown in [44], *ProM* can find the paths that minimize costs, flow time, response time, resource utilization, etc. Based on this *ProM* provides a so-called *recommendation service*. For each running instance, *Declare* continuously asks the recommendation service for suggestions, i.e., all the enabled activities are ranked based on historic information. Note that users do not have to follow recommendations; they merely serve as an advice to support the user. Related to the *recommendation service* is the *prediction service* of *ProM* that predicts when a case is finished. This information may also be used when selecting an enabled activity.

6.2.3 Monitoring process instances

The monitoring of instances is important to manage the process and to make users aware of the context of their work. Many systems use work distribution mechanisms that lead to "context tunneling", i.e., the user only sees one particular work-item in isolation without seeing the bigger picture. This is the reason that *Declare* shows the process model augmented with additional information. Using colors it is shown which constraints are satisfied and which activities are enabled (cf. Fig. 7). As explained in Sect. 4 multiple automata are used to determine the state of an instance and states of all constraints. *Declare* relies on *ProM* for monitoring processes at a higher level [4]. In the future we also hope to use the work-item visualization of YAWL [29]. This visualization allows for the definition of several "maps" on which work-items may be projected. We believe that the more flexible processes are, the more important it is to visualize "work" while using contextual information.

6.2.4 Learning from processes

Declare can export historic event information (e.g., starting and completing activities) by creating so-called MXML logs while executing instances and these can be loaded into *ProM*. *ProM* provides numerous plug-ins to extract knowledge from these *Declare* logs [4]. For example, *ProM* can extract process models based on the work that has actually been done or construct social networks based on the interactions between people. Moreover, *Declare* templates and constraints can be exported into the so-called LTL format readable by *ProM*. *ProM*'s LTL checker can then verify properties specified in LTL with respect to some event log [4].

A-posteriori analysis of instances of the model presented in Fig. 5 can provide information that can help to improve the model. For example, it is possible to check which instances violated the optional constraint in Fig. 5. Moreover, additional properties not specified in the model can be checked. Consider, for example, the verification of instances processed in the past against the following two properties: (1) activity *surgery* was executed in the instance, and (2) activity *rehabilitation* was executed after activity *surgery* in the instance. On the one hand, verification could show that the first property (1) holds in 80% of the instances processed in the previous year, while it holds in only 40% of the instances processed in the year before. This can be an indication that the hospital should hire more surgeons. On the other hand, verification could show that the second property (2) does not hold in 90% of the instances, i.e., the medical staff violated the optional constraint from the model shown in Fig. 5 in 90% of the instances. This result may indicate that this constraint should either be removed from the model or made mandatory.

In [28] it is shown that *Declare* models can also be discovered using inductive logic programming. This makes it possible to discover the primary constraints by just observing the process. The discovered models can be loaded into *Declare*.

6.2.5 Enforcing correct changes

As shown in Sects. 4 and 5, it is possible to change processes. For an ad-hoc change only one instance needs to be migrated, while for an evolutionary change in principle all running instances need to be migrated. In Fig. 9 it is shown that after making a choice the new model can be verified and that instances can be migrated. Verification of the new model is done in the same way as described in Sect. 6.1.1. This way it can be enforced that the new model is syntactically correct, i.e., free of deadlocks, etc. Moreover, using "replay" it is enforced that only those instances are migrated that actually fit into the new model. Hence, *Declare* ensures syntactical correctness even when processes are changing.

7 Limitations

In this paper, we demonstrated that declarative workflow languages can assist in balancing between flexibility and support. As a proof of concept, the *Declare* system was presented. However, the current approach also has some short-

comings. These are characterized by the following three limitations.

The first limitation is that a constraint-based approach is not very suitable for processes that are of a strict procedural nature. This becomes clear when looking at Fig. 8. If the desired control-flow is highly procedural, it is easier to simply describe what should happen rather than describing the constraints that should be satisfied. As shown in Fig. 8, the starting point of our constraint-based approach is that anything is allowed unless not explicitly forbidden. In traditional approaches everything is forbidden unless explicitly specified. The later is more suitable if the desired behavior is highly restricted.

The second limitation is that declarative workflow specifications may be less readable if many (interacting) constraints are added. A well-known problem of rule-based systems is the cognitive load on the designer if rules interact in various ways. Similar problems may be encountered if our constraint-based approach is used. However, these problems are (partly) addressed by our graphical language and our verification techniques. Unlike rule-based systems, we provide a powerful graphical notation which immediately shows possible interactions between constraints. Moreover, our verification techniques can be used to easily detect conflicting constraints (cf. Sect. 6.1.1).

An alternative way to tackle some of the problems associated to the first two limitations is to use composition and mixtures of various languages. In [3, 33] we show that flexibility can be provided as a "service", i.e., using service-orientation various languages can be mixed. For example, we have integrated YAWL, Worklets, and *Declare* using such an approach. Moreover, composition can be used to define a hierarchy of simpler models ("divide and conquer").

The third limitation is the efficiency of the current implementation. The current *Declare* engine has problems dealing with large specifications. This is due to the complexity of the model-checking techniques used. So far we have followed a rather straightforward implementation strategy and a much faster implementation is possible, e.g., by a better encoding of the automaton and by cashing results. Moreover, the underlying constraint language can be restricted to not use the full power of LTL. By limiting the expressiveness more efficient implementations come into reach. We expect that, based on focussed research and development efforts, highly performing implementations are possible.

8 Related work

Many researchers have been trying to provide ways of avoiding the apparent paradox where, on the one hand, there is the desire to control the process and to avoid incorrect or undesirable executions of the processes, and, on the other hand, users want lots of flexibly and to feel unconstrained in their actions [2, 5, 6, 8, 11, 12, 16, 22, 38, 40, 48].

See [5, 9, 23, 26, 33, 39, 42, 43, 47] for various taxonomies/classifications of workflow flexibility. The taxonomy in this paper is based on [43].

It is impossible to provide a complete overview of related work. Therefore, we refer to only some of the most related papers in this area. In [47] 18 "change patterns" (e.g. inset/delete process fragments) and 7 "change support features" (e.g., correctness enforcement and instance migration) are identified. The case handling concept is advocated as a way to avoid restricting users in their actions [8]. This is achieved by a range of mechanisms that allow for implicit deviations that are rather harmless. In [12] completely different techniques are used, but also the core idea is that implicit paths are generated to allow for more flexibility. In [9, 11] pockets of flexibility are identified that are specified/selected later in the process, i.e., there is some form of "late binding" at run time. A similar approach was proposed in [42] where the process of defining a change is integrated in the process itself. Many papers look at problems related to ad-hoc and/or evolutionary change [2, 16, 22, 38, 40, 48]. The problem of the dynamic change bug was introduced in [22]. In [2] this problem is addressed by calculating so-called change regions based on the structure of the process. A particular correctness property is described in [48] and the problem of instance migration is also investigated in [16]. In the context of the ADEPT system the problem of workflow change has been investigated in detail (including data analysis) [38–40]. In [41] a holistic approach is given which combines the ADEPT framework for process change with cased-based reasoning technology to learn from change. Case based reasoning was also proposed in [32] to reuse past experiences with change. Here a suspension mechanism was proposed that allows the designer to modify suspended parts of the workflow while other parts continue to be executed.

Another popular stream of research is applying rule-based or constraint-based process modeling languages [20, 25, 46] that are able to offer multiple execution alternatives and, therefore, can enhance flexibility by design. In [25], Glance et al. use process grammars for definition of rules involving activities and documents. Process models are executed via execution of rules that trigger each other. The Freeflow prototype presented in [20] uses constraints for building declarative process models. Freeflow constraints represent dependencies between states (e.g., inactive, active, disabled, enabled, etc.) of different activities, i.e., an activity can enter a specific state only if another activity is in a certain state. Some approaches consider process models based on dependencies between events involving activities [18, 46]. For example, the constraint-based language presented in [46] uses rules involving (1) preconditions that must hold

before an activity can be executed, (2) postconditions that must hold after an activity is executed and (3) additional conditions that must hold in general before or after an activity is executed. A similar idea was presented in [27] by Joeris, who proposes flexible workflow enactment based on event-condition-action (ECA) rules. In [30], a temporal constraint network is proposed for business process execution. The authors use thirteen temporal intervals defined by Allen [13] (e.g., *before*, *meets*, *during*, *overlaps*, *starts*, *finishes*, *after*, etc.) to define selection constraints (which define activities in a process) and scheduling constraints (which define when these activities should be executed). Several approaches propose using *intertask dependencies* for specification of the process models. In [14, 15], Attie et al. propose using Computational Tree Logic (CTL) for the specification of intertask dependencies amongst different unique events (e.g., commit dependency, abort dependency, conditional existence dependency, etc.). Dependencies are transformed into automata, which are used by a central scheduler to decide if particular events are accepted, delayed or rejected. In [36, 37], Raposo et al. propose a larger set of basic interdependencies and propose modeling their coordination using Petri nets.

It is also interesting to mention some commercial WFMSs in this context. Historically, InConcert of Xerox and Ensemble of FileNet were systems among the first commercial systems to address the problem of change. Both supported ad-hoc changes in a rather restrictive setting. Several systems have been extended with some form of late binding. For example, the Staffware workflow system allows for the dynamic selection of subprocesses at run time. Probably the most flexible commercial system is FLOWer of Pallas Athena [8]; this system supports a variety of case handling mechanisms to enable flexibility at run time while avoiding changes of the model.

This paper is based on the earlier work on *Declare* and related languages such as ConDec and DecSerFlow [6, 33–35, 45]. In [3] it is explained how *Declare* can be combined with other (not constraint-based) approaches.

9 Conclusion

In this paper, we advocated the use of declarative language as a means to balance between flexibility and support. The ideas have been implemented which resulted in *Declare*, an open source workflow management system. *Declare* provides a wide range of flexibility mechanisms: *defer* (decide to decide later), *change* (decide to change model), and *deviate* (decide to ignore model). Both ad-hoc and evolutionary change are supported and models can be verified. As far as we know, there is no other workflow management system that supports such a wide range of flexibility mechanisms. In this paper, we also elaborated a bit on the link with process mining. The connection between *ProM* and *Declare* enables innovate means of analysis, e.g., users may get recommendations based on historic information.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

References

1. van der Aalst WMP (1998) The application of Petri nets to workflow management. J Circ Syst Comput 8(1):21–66
2. van der Aalst WMP (2001) Exterminating the dynamic change bug: a concrete approach to support workflow change. Inform Syst Front 3(3):297–317
3. van der Aalst WMP, Adams M, ter Hofstede AHM, Pesic M, Schonenberg H (2008) Flexibility as a service. BPM Center Report BPM-08-09, BPMcenter.org
4. van der Aalst WMP, van Dongen BF, Günther CW, Mans RS, Alves de Medeiros AK, Rozinat A, Rubin V, Song M, Verbeek HMW, Weijters AJMM (2007) ProM 4.0: Comprehensive support for real process analysis. In: Kleijn J, Yakovlev A (eds) Application and theory of Petri nets and other models of concurrency (ICATPN 2007), vol 4546 of Lecture Notes in Computer Science, pp 484–494. Springer-Verlag, Berlin
5. van der Aalst WMP, Jablonski S (2000) Dealing with workflow change: identification of issues and solutions. Int J Comput Syst Sci Eng 15(5):267–276
6. van der Aalst WMP, Pesic M (2006) DecSerFlow: Towards a truly declarative service flow language. In: Bravetti M, Nunez M, Zavattaro G (eds) International Conference on Web Services and Formal Methods (WS-FM 2006), vol 4184 of Lecture Notes in Computer Science, pp 1–23, Springer-Verlag, Berlin
7. van der Aalst WMP, Reijers HA, Weijters AJMM, van Dongen BF, Alves de Medeiros AK, Song M, Verbeek HMW (2007) Business process mining: an industrial application. Inform Syst 32(5):713–732
8. van der Aalst WMP, Weske M, Grünbauer D (2005) Case handling: a new paradigm for business process support. Data Know Eng 53(2):129–162
9. Adams M (2007) Facilitating dynamic flexibility and exception handling for workflows. Phd thesis, Queensland University of Technology, Brisbane
10. Adams M, ter Hofstede AHM, van der Aalst WMP, Edmond D (2007) Dynamic, extensible and context-aware exception handling for workflows. In: Curbera F, Leymann F, Weske M (eds) Proceedings of the OTM Conference on Cooperative information Systems (CoopIS 2007), vol 4803 of Lecture Notes in Computer Science, pp 95–112. Springer-Verlag, Berlin
11. Adams M, ter Hofstede AHM, Edmond D, van der Aalst WMP (2006) Worklets: A service-oriented implementation of dynamic flexibility in workflows. In: Meersman R, Tari Z et al (eds) On the Move to Meaningful Internet Systems 2006, OTM Confederated International Conferences, 14th International Conference on Cooperative Information Systems (CoopIS 2006), vol 4275 of Lecture Notes in Computer Science, pp 291–308. Springer-Verlag, Berlin
12. Agostini A, De Michelis G (2000) Improving flexibility of workflow management systems. In: van der Aalst WMP, Desel J, Oberweis A (eds) Business process management: Models, techniques,

and empirical studies, vol 1806 of Lecture Notes in Computer Science, pp 218–234. Springer-Verlag, Berlin
13. Allen JF (1983) Maintaining knowledge about temporal intervals. Commun ACM 26(11):832–843
14. Attie PC, Singh MP, Emerson EA, Sheth A, Rusinkiewicz M (1996) Scheduling workflows by enforcing intertask dependencies. Distrib Syst Eng J 3(4):222–238
15. Attie PC, Singh MP, Sheth A, Rusinkiewicz M (1993) Specifying and enforcing intertask dependencies. In: 19th International Conference on Very Large Data Bases (VLDB), pp 134–145, Dublin, Ireland, August 24–27, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
16. Casati F, Ceri S, Pernici B, Pozzi G (1998) Workflow evolution. Data Knowl Eng 24(3):211–238
17. Clarke EM, Grumberg O, Peled DA (1999) Model checking. The MIT Press, Cambridge, London
18. Decker G, Grosskopf A, Barros A (2007) A graphical notation for modeling complex events in business processes. In: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), pp 27–36, IEEE Computer Society
19. Declare (2008) http://declare.sf.net. Accessed March 1, 2009
20. Dourish P, Holmes J, MacLean A, Marqvardsen P, Zbyslaw A (1996) Freeflow: Mediating between representation and action in workflow systems. In: Proceedings of the CM Conference on Computer Supported Cooperative Work (CSCW '96), pp 190–198. ACM Press, New York
21. Dumas M, van der Aalst WMP, ter Hofstede AHM (2005) Process-aware information systems: Bridging people and software through process technology. Wiley & Sons, Hoboken
22. Ellis CA, Keddara K, Rozenberg G (1995) Dynamic change within workflow systems. In: Comstock N, Ellis C, Kling R, Mylopoulos J, Kaplan S (eds) Proceedings of the Conference on Organizational Computing Systems, pp 10–21, Milpitas, California, ACM SIGOIS, ACM Press, New York
23. Georgakopoulos D, Hornick M, Sheth A (1995) An overview of workflow management: from process modeling to workflow automation infrastructure. Distrib Parall Datab 3:119–153
24. Giannakopoulou D, Havelund K (2001) Automata-based verification of temporal properties on running programs. In: ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering, p 412, Washington, DC, IEEE Computer Society
25. Glance N, Pagani D, Pareschi R (1996) Generalised process structure grammars (GPSG) for flexible representations of work. In: Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW'96), pp 190–198, ACM Press, New York
26. Heinl P, Horn S, Jablonski S, Neeb J, Stein K, Teschke M (1999) A comprehensive approach to flexibility in workflow management systems. In: Georgakopoulos G, Prinz W, Wolf AL (eds) Work Activities Coordination and Collaboration (WACC'99), pp 79–88, San Francisco, February, ACM press
27. Joeris G (2000) Decentralized and flexible workflow enactment based on task coordination agents. In: Wangler B, Bergman L (eds) Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE'00), vol 1789 of Lecture Notes in Computer Science, pp 41–62, Stockholm, Sweden, Springer-Verlag, Berlin
28. Lamma E, Mello P, Montali M, Riguzzi F, Storari S (2007) Inducing declarative logic-based models from labeled traces. In: Alonso G, Dadam P, Rosemann M (eds) International Conference on Business Process Management (BPM 2007), vol 4714 of Lecture Notes in Computer Science, pp 344–359. Springer-Verlag, Berlin
29. de Leoni M, van der Aalst WMP, ter Hofstede AHM (2008) Visual support for work assignment in process-aware information systems. In: Dumas M, Reichert M, Shan MC (eds) International Conference on Business Process Management (BPM 2008), vol 5240 of Lecture Notes in Computer Science, pp 67–83. Springer-Verlag, Berlin
30. Lu R, Sadiq S, Padmanabhan V, Governatori G (2006) Using a temporal constraint network for business process execution. In: Proceedings of the 17th Australasian Database Conference (ADC '06), pp 157–166, Darlinghurst, Australia, Australian Computer Society, Inc
31. Ly LT, Rinderle S, Dadam P (2006) Semantic correctness in adaptive process management systems. In: Business process management, vol 4102 of Lecture Notes in Computer Science, pp 193–208. Springer-Verlag, Berlin
32. Minor M, Schmalen D, Koldehoff A, Bergmann R (2007) Structural adaptation of workflows supported by a suspension mechanism and by case-based reasoning. In: Proceedings of WETICE 2007, pp 370–375
33. Pesic M (2008) Constraint-based workflow management systems: Shifting control to users. Phd thesis, Eindhoven University of Technology, Eindhoven
34. Pesic M, Schonenberg H, van der Aalst WMP (2007) DECLARE: Full support for loosely-structured processes. In: Spies M, Blake MB (eds) Proceedings of the Eleventh IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), pp 287–298. IEEE Computer Society
35. Pesic M, Schonenberg MH, Sidorova N, van der Aalst WMP (2007) Constraint-based workflow models: Change made easy. In: Curbera F, Leymann F, Weske M (eds) Proceedings of the OTM Conference on Cooperative information Systems (CoopIS 2007), vol 4803 of Lecture Notes in Computer Science, pp 77–94. Springer-Verlag, Berlin
36. Raposo AB, Fuks H (2002) Defining task interdependencies and coordination mechanisms for collaborative systems. In: Blay-Fornarino M, Pinna-Dery AM, Schmidt K, Zaratè P (eds) Cooperative systems design, vol 74 of Frontiers in Artificial Intelligence and Applications, pp 88–103, Amsterdam, The Netherlands, IOS Press
37. Raposo AB, Magalhaes LP, Ricarte ILM, Fuks H (2001) Coordination of collaborative activities: A framework for the definition of tasks interdependencies. In: Proceedings of the 7th International Workshop on Groupware (CRIWG), pp 170–179, IEEE Computer Society
38. Reichert M, Dadam P (1998) ADEPTflex: supporting dynamic changes of workflow without loosing control. J Intell Inform Syst 10(2):93–129
39. Rinderle S, Reichert M, Dadam P (2003) Evaluation of correctness criteria for dynamic workflow changes. In: van der Aalst WMP, ter Hofstede AHM, Weske M (eds) International Conference on Business Process Management (BPM 2003), vol 2678 of Lecture Notes in Computer Science, pp 41–57. Springer-Verlag, Berlin
40. Rinderle S, Reichert M, Dadam P (2004) Correctness criteria for dynamic changes in workflow systems: a survey. Data Knowl Eng 50(1):9–34
41. Rinderle S, Weber B, Reichert M, Wild W (2005) Integrating process learning and process evolution: A semantics based approach. In: van der Aalst WMP, ter Hofstede AHM, Weske M (eds) International Conference on Business Process Management (BPM 2005), vol 2678 of Lecture Notes in Computer Science, pp 252–267. Springer-Verlag, Berlin
42. Sadiq S, Sadiq W, Orlowska M (2001) Pockets of flexibility in workflow specification. In: Proceedings of the 20th International Conference on Conceptual Modeling (ER 2001), vol 2224 of Lecture Notes in Computer Science, pp 513–526. Springer-Verlag, Berlin
43. Schonenberg H, Mans R, Russell N, Mulyar N, van der Aalst WMP (2008) Process flexibility: A survey of contemporary approaches.

In: Dietz J, Albani A, Barjis J (eds) Advances in enterprise engineering I, vol 10 of Lecture Notes in Business Information Processing, pp 16–30. Springer-Verlag, Berlin

44. Schonenberg H, Weber B, van Dongen BF, van der Aalst WMP (2008) Supporting flexible processes through recommendations based on history. In: Dumas M, Reichert M, Shan MC (eds) International Conference on Business Process Management (BPM 2008), vol 5240 of Lecture Notes in Computer Science, pp 51–66, Springer-Verlag, Berlin
45. Schonenberg MH, Mans RS, Russell NC, Mulyar NA, van der Aalst WMP (2007) Towards a taxonomy of process flexibility (extended version). BPM Center Report BPM-07-11, BPMcenter.org
46. Wainer J, de Lima Bezerra F (2003) Constraint-based flexible workflows. In: Proceedings of the 9th International Workshop on Groupware: Design, Implementation, and Use (CRIWG 2003), vol 2806, pp 151–158 Springer-Verlag, Berlin
47. Weber B, Reichert M, Rinderle-Ma S (2008) Change patterns and change support features: Enhancing flexibility in process-aware information systems. Data Knowl Eng 66(3):438–466
48. Weske M (2001) Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In: Sprague R (ed) Proceedings of the Thirty-Fourth Annual Hawaii International Conference on System Science (HICSS-34). IEEE Computer Society Press, Los Alamitos

Wil van der Aalst is a full professor of Information Systems at the Technische Universiteit Eindhoven and an adjunct professor at Queensland University of Technology. His research interests include workflow management, process mining, Petri nets, business process management, process modeling, and process analysis. Many of his papers are highly cited (he has an H-index of 55 according to Google Scholar) and his ideas have influenced researchers, software developers, and standardization committees working on process support. For more information about his work visit: www.workflowpatterns.com, www.workflowcourse.com, www.processmining.org, www.yawl-system.com, www.wvdaalst.com.

Maja Pesic studied information systems at University of Belgrade, Serbia. She received a Ph.D. degree in 2008 at Eindhoven University of Technology, The Netherlands. Her research in the field of workflow management systems aims at increasing the flexibility of these systems. Currently, she is working as a researcher (Postdoc) at the department of Mathematics and Computer Science at the Eindhoven University of Technology, The Netherlands.

Helen Schonenberg is currently working as Ph.D. student in the group of professor Wil van der Aalst at Eindhoven University of Technology, The Netherlands. Her research focus is on supporting the execution of flexible processes by means of recommendations and predictions, using process mining techniques. Schonenberg received her Master of Science in Computer Science (Ingenieur) in 2006 from the University of Twente, The Netherlands.