Constraint-Based Workflow Models: Change Made Easy

M. Pesic[1], M.H. Schonenberg[2], N. Sidorova[2], and W.M.P. van der Aalst[2]

[1] Department of Technology Management
[2] Department of Mathematics and Computer Science,
Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands
{m.pesic, m.h.schonenberg, n.sidorova, w.m.p.v.d.aalst}@tue.nl

Abstract. The degree of flexibility of workflow management systems heavily influences the way business processes are executed. Constraint-based models are considered to be more flexible than traditional models because of their semantics: everything that does not violate constraints is allowed. Although constraint-based models are flexible, changes to process definitions might be needed to comply with evolving business domains and exceptional situations. Flexibility can be increased by run-time support for dynamic changes – transferring instances to a new model – and ad-hoc changes – changing the process definition for one instance. In this paper we propose a general framework for a constraint-based process modeling language and its implementation. Our approach supports both ad-hoc and dynamic change, and the transfer of instances can be done easier than in traditional approaches.

1 Introduction

When supporting business processes 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, workers want flexible processes that do not constrain them in their actions. This apparent paradox has limited the application of workflow management systems thus far since, as indicated by many authors, workflow management systems are too restrictive and have problems concerning dealing with change [3].

Many approaches to resolve the paradox have been proposed. Some of them try to avoid change, e.g. by generating implicit alternative paths [6, 8], or by differing the selection of the desired behavior [7]. Others allow for changing the model for a single instance and/or changing a process model while migrating all instances [9, 11, 19, 23]. The migration of process instances from one model to another introduces many interesting problems [3, 9, 19, 23]. For example, the "dynamic change bug" originally described in [11] shows that it may be impossible to put the process instance into a suitable state of the new model without skipping or repeatedly executing tasks.

In this paper we propose a solution that has some features of both approaches: we try to avoid the need for change and at the same time we provide full support for all kinds of change. To avoid the need for change we address the following problem: Traditional workflow languages force or stimulate the designer to *over-specify* things. For

R. Meersman and Z. Tari et al. (Eds.): OTM 2007, Part I, LNCS 4803, pp. 77–94, 2007.
© Springer-Verlag Berlin Heidelberg 2007

example, it is possible to model all kinds of choices in today's systems. It is, however, not possible to simply state that two activities should never occur together. Instead, the user is forced to provide a detailed strategy to implement this simple requirement, e.g. by introducing a decision task and deciding when and by whom this task is executed. We believe that replacing the imperative approach with a *declarative* one is essential for making workflow management more flexible. Therefore, we consider here a framework where workflows are defined by constraint models.

Avoiding over-specification makes processes more flexible (more execution paths are allowed) and allows avoiding costly changes. Change is sometimes still unavoidable because of exceptions (e.g., an important customer has a special request which requires the violation of a business rule) or changed circumstances (e.g., a new law enforcing to reverse the order of two activities). This paper explores *what change means in the context of constraint-based languages*. Here it is interesting to see whether similar problems as reported in [3, 11, 19] occur. Surprisingly, it turns out that both ad-hoc and evolutionary changes are rather easy to support. This explains the subtitle of this paper: "Change Made Easy".

The results we report here show that it is possible both (1) to avoid the need for unnecessary changes and restrictions (using a more declarative style) and (2) to provide support for changes at the instance level (ad-hoc change) and at the type level (evolutionary change). Moreover, it is also possible to easily differentiate between mandatory and optional constraints. A user is forbidden to violate a mandatory constraint or to change a model so that a mandatory constraint becomes violated. For optional constraints a warning is generated and the user may choose to violate it or not. Note that in each case model checking techniques can give good diagnostic information that helps the user to understand potential problems.

Our framework is supported by the *ConDec* language [16]. ConDec is a graphical declarative process modeling language supported by the *Declare* tool, which also supports related languages such as *DecSerFlow* [4]. The Declare workflow management system is open in the sense that it can support multiple constraint-based languages and each of the languages is extendible and can be changed without changing the engine. This is achieved by a flexible mechanism mapping graphical constraints onto LTL (Linear Temporal Logic) [14]. Note that the semantics are expressed in a temporal logic but the end user only sees the graphical notation when modeling. The Declare system fully supports the approach presented in this paper and the software can be downloaded from `http://is.tm.tue.nl/staff/mpesic/declare.htm`.

The remainder of the paper is organized as follows. The general framework of constraint modeling and changes are presented in Section 2. In Section 3 we describe ConDec, an implementation of a constraint modeling language, based on the general framework. ConDec is supported by the Declare tool. The support of change in ConDec is described in Section 4. Section 5 discusses related work, and finally Section 6 concludes the paper and gives directions for future work.

2 Constraint Models

Constraint models are suitable for supporting flexible processes that allow many different executions. Most theoretical process modeling languages, such as Petri Nets [17],

process algebras [15] and more applied business languages like BPMN, UML and EPCs [20] define direct causal relationships between activities in process models. Opposed to this, constraint-based languages are of a less procedural nature and use a more declarative style. Using constraints, the behavior is restricted. Unlike procedural languages constraints may be non-local, e.g., *"eventually A is followed by B"* and negative, e.g., *"either A or B can occur but not both"*.

Activities and constraints on activities are the key elements of a constraint-based model. We distinguish the universe of all activities $\mathcal{A}$ and the universe of all constraints $\mathcal{C}$. $\mathcal{A}^*$ denotes the set of all sequences over $\mathcal{A}$. We say that c is a constraint over $A \subseteq \mathcal{A}$, if it does not mention any activity $a \notin A$. A constraint is a boolean expression that evaluates to true or false for every trace $\sigma \in \mathcal{A}^*$. If a constraint c evaluates to true for a trace $\sigma \in \mathcal{A}^*$, then we say that σ satisfies c, denoted as $\sigma \vDash c$, otherwise we denote it as $\sigma \nvDash c$. In Section 3 we will show that such constraints can be expressed graphically and be mapped onto LTL.

Example 1 (Constraint). Let $A = \{curse, pray\}$ be a set of two activities and $c =$ *"Every curse activity is eventually followed by a pray activity"* be a constraint over A. Then $\langle curse \rangle \nvDash c$, $\langle pray, pray \rangle \vDash c$, $\langle curse, curse, pray \rangle \vDash c$ and $\langle curse, pray, curse \rangle \nvDash c$.

2.1 Preliminaries

First we introduce sequence concatenation ($\frown$), function overriding ($\oplus$) and reduction ($\ominus$), which are used in the remainder of the framework. Sequences can be concatenated into a new sequence.

Definition 1 (Sequence concatenation $\frown$). *Let σ, γ be sequences over $\mathcal{A}$ with $\sigma = a_1, a_2, a_3, \ldots, a_n (\forall a_i \in \mathcal{A})$ and $\gamma = b_1, b_2, b_3, \ldots, b_n (\forall b_i \in \mathcal{A})$. Then $\sigma \frown \gamma = a_1, a_2, a_3, \ldots, a_n, b_1, b_2, b_3, \ldots, b_n$.*

We use function overriding to add and remap elements of a function domain and we use function reduction to remove elements from a function domain. When a function f is undefined for an element a, we denote this as $f(a) = \perp$.

Definition 2 (Function overriding $\oplus$). *Let $f : A \to B$. Then $f \oplus (a,b) : A \cup \{a\} \to B \cup \{b\}$ such that $(f \oplus (a,b))(a) = b$ and $\forall x \in A \backslash \{a\} : (f \oplus (a,b))(x) = f(x)$.*

Definition 3 (Function reduction $\ominus$). *Let $f : A \to B$, $a \in A$, $b \in B$. Then $f \ominus (a,b) : A \backslash \{a\} \to B \backslash \{b\}$ such that $(f \ominus (a,b))(a) = \perp$ and $\forall x \in A \backslash \{a\} : (f \ominus (a,b))(x) = f(x)$.*

2.2 Constraint Workflows

Constraints define the boundaries within which activities can be executed. Besides activities and constraints on these activities, the constraint model also includes a mapping that defines whether constraints are optional (may be violated) or mandatory (may never be violated).

Definition 4 (Constraint Model cm). *A constraint model cm is defined as a triple $cm = (A, C, c_{type})$, where*

- *$A \subseteq \mathcal{A}$ is a set of activities in the model;*
- *$C \subseteq \mathcal{C}$ is a set of constraints where every element $c \in C$ is a constraint over A;*
- *$c_{type} : C \rightarrow \{mandatory, optional\}$ is a function that defines whether constraints are mandatory or optional.*

We use $\mathcal{U}_{CM}$ to denote the universe of all constraint models.

For convenience, we define an operation to remove optional constraints from a constraint model.

Definition 5 (Mandatory version of *cm***).** *Let $cm = (A, C, c_{type})$ be a constraint model, then $mand(A, C, c_{type}) = (A, C', c'_{type})$, where $C' = \{c \in C \mid c_{type}(c) = mandatory\}$ and $c'_{type} : C' \rightarrow$
$\{mandatory\}$.*

A constraint workflow contains several running instances, each related to a constraint model and a sequence of actions performed by the instance up to the current moment. The framework we develop here, should support changes of the constraint model by redefining restrictions on the behavior at run time. Moreover, we want to be able to change a constraint model for a cluster of instances, which could be e.g., all instances related to the handling of complaints at an insurance department. For this purpose, we introduce the notion of constraint model identifiers. This identifier is then mapped to one of the constraint models from the universe. Instances, in their turn, are mapped to a constraint model identifier.

Definition 6 (Constraint Workflow *wf***).** *A workflow specification based on constraint models is defined by the tuple $wf = (P_{id}, cm_{id}, Pmap, CMmap, trace)$, where*

- *P_{id} is a set of process identifiers (instances);*
- *cm_{id} is a set of constraint model identifiers;*
- *$Pmap : P_{id} \rightarrow cm_{id}$ is a function that maps instances to model identifiers;*
- *$CMmap : cm_{id} \rightarrow \mathcal{U}_{CM}$ is a function that maps model identifiers to constraint models;*
- *$trace : P_{id} \rightarrow \mathcal{A}^*$ is a function that maps instances to execution traces.*

We use $\mathcal{U}_{WF}$ to denote the universe of all constraint workflows wf.

Figure 1 depicts a mapping from instances of P_{id} to model identifiers in cm_{id} and a mapping from these model identifiers to constraint models in $\mathcal{U}_{CM}$. $CMmap(cm_{id})$ results in a constraint model *cm*. Note that not all constraint models in the universe $\mathcal{U}_{CM}$ need to

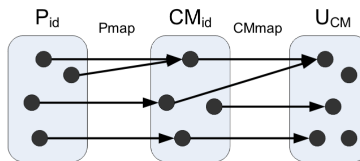


Fig. 1. Mappings

have a related model identifier in cm_{id}. Also observe that different process instances can be mapped onto the same constraint model identifier, i.e., the same constraint model. Even different constraint model identifiers might be mapped onto the same constraint model, which reflects situations in which the same constraint model is used in different contexts (e.g., two companies can develop two identical models).

Satisfaction of constraint sets depends on the execution trace.

Definition 7 (Satisfaction of constraint sets). *Let C be a set of constraints and $\sigma \in \mathcal{A}^*$ then $\sigma \models C \iff \forall c \in C : \sigma \models c$ and $\sigma \not\models C \iff \exists c \in C : \sigma \not\models c$.*

The purpose of constraints is, however, to define conditions that should hold on the completed traces of instances. During execution we can only evaluate prefixes of those traces, and a constraint violation on a prefix does not necessarily imply that the constraint will be violated on the completed trace. Therefore, we introduce an evaluation function that determines whether a constraint model *cm* is *satisfied*, *violated* or *temporarily violated*, i.e., although the trace currently violates the constraints, the constraints can still become satisfied in the future.

Definition 8 (Evaluation *eval*). *Let $cm = \in \mathcal{U}_{CM}$ be a constraint model where $cm = (A, C, c_{type})$ and $\sigma \in \mathcal{A}^*$ be a trace. Then the evaluation function eval is defined as*

$$eval(\sigma, (A, C, c_{type})) = \begin{cases} \textit{satisfied} & \textit{if } \sigma \models C; \\ \textit{temporarily violated} & \textit{if } (\sigma \not\models C) \wedge (\exists \gamma \in A^* : \sigma^\frown\gamma \models C); \\ \textit{violated} & \textit{otherwise}. \end{cases}$$

Example 2. (Satisfaction) Consider again constraint c, with $c =$ *"Every curse activity is eventually followed by a pray activity"*. Suppose trace $\sigma = \langle curse, curse\rangle$ at some moment during execution. Obviously $\sigma \not\models c$, but σ can be a prefix of a trace that could satisfy c during execution. For example $\langle curse, curse, pray\rangle \models c$.

When an instance executes actions, the trace of that instance is updated.

Definition 9 (Execution *exec*). *Let $wf \in \mathcal{U}_{WF}$ be a constraint workflow where $wf = (P_{id}, cm_{id}, Pmap, CMmap, trace)$. Let $p_{id} \in P_{id}$, $CMmap(Pmap(p_{id})) = cm$, $cm = (A, C, c_{type})$ and $a \in A$. Then the execution function exec is defined as*

$$exec(wf, a, p_{id}) = (P_{id}, cm_{id}, Pmap, CMmap, trace \oplus (p_{id}, trace(p_{id})^\frown\langle a\rangle)).$$

Furthermore we call an execution:

- ***normal*** *if no constraint is violated:* $eval(trace(p_{id})^\frown\langle a\rangle, cm) \neq violated$;
- ***deviating*** *if only optional constraints are violated:* $eval(trace(p_{id})^\frown\langle a\rangle, cm) = violated \wedge eval(trace(p_{id})^\frown\langle a\rangle, mand(cm)) \neq violated$;
- ***invalid*** *otherwise.*

If an instance is closed, the instance will be removed, together with its trace. Unlike in procedural language where an instance is closed automatically when some closing state is reached, instances of constraint-based models can have multiple states at which the instance could be closed. Therefore, many strategies can be used to close an instance of a constraint-based model. One example of a closing strategy would be to allow users to explicitly choose when to close an instance and to allow only "normal" closings.

Definition 10 (Closing an instance *close***).** *Let $wf = \in \mathcal{U}_{WF}$ be a constraint workflow where $wf = (P_{id}, cm_{id}, Pmap, CMmap, trace)$. Let $p_{id} \in P_{id}$, $Pmap(p_{id}) = cm_{id}$ and $CMmap(cm_{id}) = cm$. Then closing instance close is defined as*

$$close(wf, p_{id}) = (P_{id}', cm_{id}, Pmap', CMmap, trace'),$$

where
$P_{id}' = P_{id} \setminus \{p_{id}\}$, $Pmap' = Pmap \ominus (p_{id}, cm_{id})$ and $trace' = trace \ominus (p_{id}, trace(p_{id}))$.
We call closing of an instance:

- ***normal** if all constraints are satisfied: $eval(trace(p_{id}), cm) = satisfied$;*
- ***deviating** if all mandatory constraints, but not all optional constraints are satisfied: $eval(trace(p_{id}), cm) \neq satisfied \wedge eval(trace(p_{id}), mand(cm)) = satisfied$;*
- ***invalid** otherwise.*

Operations. We can easily add an instance to a workflow by extending the instance set and adding an empty trace for this instance to the trace mapping.

Definition 11 (Adding a process instance add_{PI}**).** *Let $wf \in \mathcal{U}_{WF}$ be a constraint workflow where $wf = (P_{id}, cm_{id}, Pmap, CMmap, trace)$. Let $p_{id} \notin P_{id}$, $cm_{id} \in cm_{id}$, then*

$$add_{PI}(wf, p_{id}, cm_{id}) = (P_{id} \cup \{p_{id}\}, cm_{id}, Pmap \oplus (p_{id}, cm_{id}), CMmap, trace \oplus (p_{id}, \langle\rangle)).$$

Constraint models from the universe can be added to the workflow by adding a constraint model identifier to the identifier set and linking the identifier to the required constraint model.

Definition 12 (Adding a constraint model add_{CMI}**).** *Let $wf \in \mathcal{U}_{WF}$ be a constraint workflow where $wf = (P_{id}, cm_{id}, Pmap, CMmap, trace)$. Let $cm_{id} \notin cm_{id}$, $cm \in \mathcal{U}_{CM}$, then*

$$add_{CMI}(wf, cm_{id}, cm) = (P_{id}, cm_{id} \cup \{cm_{id}\}, Pmap, CMmap \oplus (cm_{id}, cm), trace).$$

Verification. Once a constraint model has been defined, we can verify whether it contains dead activities and conflicts. We call an activity a dead activity, when all traces that contain this activity violate the constraints. We say that the model contains a conflict when there are no traces that could (eventually) satisfy the constraints.

Definition 13 (Dead activity). *Let $cm = (A, C, c_{type})$ and $a \in A$. Then a is a dead activity if $\forall \sigma \in \mathcal{A}^* : a \in \sigma \Rightarrow \sigma \not\models C$.*

Definition 14 (Conflict). *Let $cm = (A, C, c_{type})$, then there is a conflict in cm if $\forall \sigma \in \mathcal{A}^* : \sigma \not\models C$.*

Note that if there is a conflict in the model, then all activities of the model are dead activities.

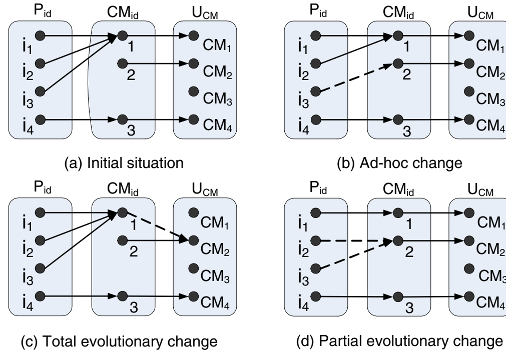


Fig. 2. Changes for constraint workflows

2.3 Change

Constraint models support both ad-hoc and evolutionary changes. Ad-hoc changes are typically needed to handle an exceptional situation for one case. An ad-hoc change for an instance is allowed, when the instance trace satisfies the new model. Evolutionary changes occur when there is a change in the process itself, e.g., by new laws or business strategies. Traces of all running instances of the corresponding process model should be evaluated and, if possible, the instances should be transferred (remapped) to the new model. When this is not possible, i.e., the trace violates the new constraints, we call this a *history violation*.

Figure 2 depicts a constraint workflow (without the *trace* mapping) and is used to illustrate all change types. In Figure 2(a) four instances are depicted, of which i_1, i_2, i_3 are instances of constraint model cm_1. Instance i_4 is an instance of cm_4. Constraint model cm_2 has no instances yet and cm_3 is not part of the constraint workflow. In the remainder of this section we will explain all change types, by describing changes performed on the original workflow, depicted in Figure 2(a). Dashed lines denote a remapping of *Pmap* or *CMmap* with respect to the original workflow.

Definition 15 (Ad-hoc change δ_{AH}). *Let $wf \in \mathcal{U}_{WF}$ be a constraint workflow where $wf = (P_{id}, cm_{id}, Pmap, CMmap, trace)$. Let $p_{id} \in P_{id}$, $cm_{id} \in cm_{id}$* [1], *$eval(trace(pid), CMmap(cm_{id})) \neq violated$, then*

$$\delta_{AH}(wf, p_{id}, cm_{id}) = (P_{id}, cm_{id}, Pmap \oplus (p_{id}, cm_{id}), CMmap, trace).$$

Figure 2(b) shows a possible ad-hoc change. Suppose we would like to perform an ad-hoc change on instance i_3. We want this instance to use constraint model cm_2 instead of cm_1. If the trace of instance i_3 does not violate the constraints of the new model cm_2, we can remap instance i_3 to identifier 2, which maps onto model cm_2.

Total evolutionary changes can only be performed when all instances satisfy the new model. If this is the case, all instances will be transfered to the new model.

[1] Note that if $cm_{id} \notin cm_{id}$, operation add_{CMI} could be executed first.

Definition 16 (Total evolutionary change $\delta_{E_{total}}$). *Let $wf \in \mathcal{U}_{WF}$ be a constraint workflow where $wf = (P_{id}, cm_{id}, Pmap, CMmap, trace)$. Let $cm_{id} \in cm_{id}$, $cm \in \mathcal{U}_{CM}$, $\forall p_{id} \in Pmap^{-1}(cm_{id}) : eval(trace(p_{id}), CMmap(cm_{id})) \neq violated$, then*

$$\delta_{E_{total}}(wf, cm_{id}, cm) = (P_{id}, cm_{id}, Pmap, CMmap \oplus (cm_{id}, cm), trace).$$

In Figure 2(c) we illustrate an example of a total evolutionary change. Suppose we would like to transfer all instances of constraint model cm_1 (instances mapped to identifier 1) to cm_2. When for all instances the current trace satisfies cm_2, we can remap identifier 1 to cm_2.

We also define partial evolutionary change, in which only instances that satisfy the new model are transferred to the new model. All other instances proceed their execution according to the old model.

Definition 17 (Partial evolutionary change $\delta_{E_{partial}}$). *Let $wf \in \mathcal{U}_{WF}$ be a constraint workflow where $wf = (P_{id}, cm_{id}, Pmap, CMmap, trace)$. Let $cm_1 \in cm_{id}$, $cm_2 \in cm_{id}$, then*

$$\delta_{E_{partial}}(wf, cm_1, cm_2) = (P_{id}, cm_{id}, Pmap', CMmap, trace),$$

where $Pmap' : P_{id} \rightarrow cm_{id}$, such that

$$Pmap'(p_{id}) = \begin{cases} cm_2 & , \forall p_{id} \in SatP_{id} \\ Pmap(p_{id}) & , \forall p_{id} \in P_{id} \backslash SatP_{id}. \end{cases}$$

and
$SatP_{id} = \{p_{id} \in P_{id} \mid Pmap(p_{id}) = cm_1 \wedge eval(trace(p_{id}), CMmap(cm_2)) \neq violated\}.$

An example of partial evolutionary change is given in Figure 2(d). Again, suppose we would like to transfer instances of constraint model cm_1 (instances mapped to identifier 1) to cm_2. Then all instances that satisfy cm_2 are remapped to an identifier that is related to cm_2. Note that for instance i_1 it is not possible to migrate. Therefore, it remains an instance of cm_1.

Change in imperative models is hindered by the fact that an equivalent new state must be found in the new model, which is not always possible [11]. For declarative models it is straightforward to transfer instances. Instances for which the current trace satisfies the constraints of the new model, are mapped onto the new model. Hence the "dynamic change bug" described in [11] does not apply. In the next section we will present an implementation based on this framework.

3 ConDec and Declare

In this section we briefly introduce ConDec [16], a constraint-based process modeling language, based on the framework we presented in Section 2. ConDec is supported by the Declare tool, see Section 3.2.

3.1 ConDec

ConDec uses *an open set of constraint templates* for the definition of relationships between activities. Each template has (1) a name, (2) a graphical representation and (3)

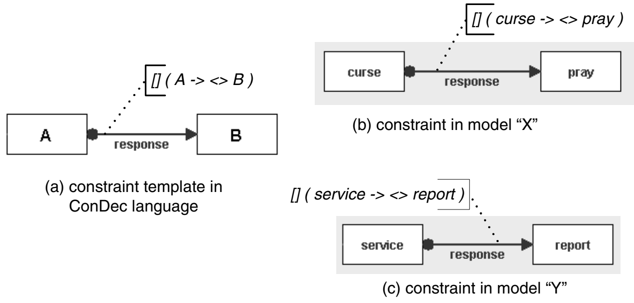


Fig. 3. Constraint template "response" and two "response" constraints

semantics given by a Linear Temporal Logic (LTL) formula on finite traces [13]. LTL is a temporal logic that, in addition to classical logical operators, uses several temporal operators: always ($\Box$), eventually ($\Diamond$), until ($\sqcup$) and next time ($\bigcirc$) [14]. LTL formulas can be added to the language by means of constraint templates. Constraint templates are parameterized graphical representations of LTL formulas. Templates can easily be added, removed and changed in ConDec.

Figure 3 shows a constraint template and its application to two models. For the sake of clarity, we have also added the corresponding LTL formulas to the picture. The template depicted in Figure 3(a) is the response template and it is defined as a single line with special symbols between some activities "A" and "B", i.e., "A" and "B" are parameters of the template. The semantics of the template are given by the formula $\Box(A \rightarrow \Diamond B)$: every execution of activity "A" should eventually be followed by at least one execution of activity "B". The response template can be used to create response constraints in various ConDec process models, by replacing template parameters with activities from the model. Figures 3(b) and 3(c) show parts of two ConDec models, each containing a response constraint between two activities.

Defining templates in this way enables adding various types of relations between activities in ConDec. More than twenty LTL-based constraint templates are described in [5]. The great benefit of constraint templates is that LTL formulas are hidden from the users, therefore they do not have to be LTL experts in order to understand underlying formulas.

Process Modeling in ConDec. ConDec models are suitable for supporting flexible processes with many deviations during the execution. As an example, consider the process for a car rental shop. The model of the car rental process is given in Figure 4. Initially, the client gets registered (activity "register client data"). The client will be charged (activity "charge") for the rental and (if applicable) all damage he caused. The "charge" activity will occur at least once, but the moment of charging is not fixed. If during the rental period a problem is identified (activity "identify problem"), then car will be checked (activity "check"). During the rental period the client can request repairs (activity "request") on which the car rental shop will repair the car if necessary (activity "service") and include the findings in the maintenance report of the car (activity "report") at a suitable moment. The client could request many repairs, or none at all and it is not known in advance when requests will be made.

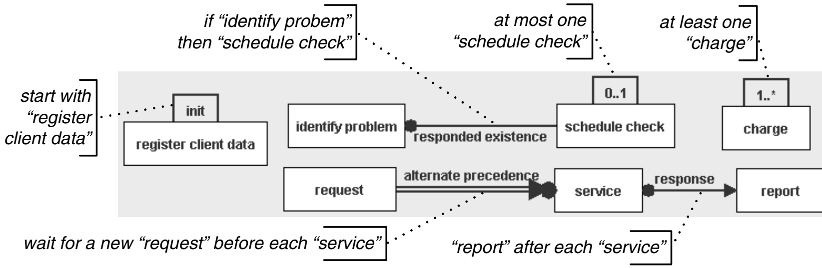
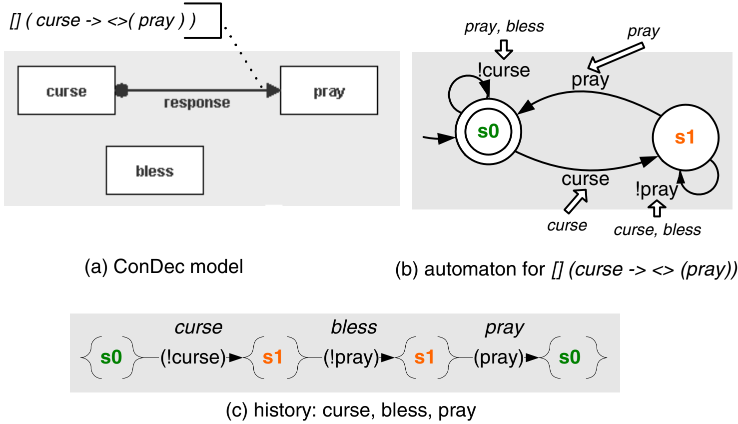


Fig. 4. Activities and constraints in car rental example

To model the the process of the car rental shop, we add several constraints on the execution of the activities in the shop. Every instance must start with registering the client (constraint "init"). The client will be charged for the rental and for all caused damage, so he will be charged at least once (constraint "1..*"). Every repair service on the car will only be done on request of the client, i.e., there has to be at least one occurrence of activity "request" between each two occurrences of activity "service". Note that other activities may be executed in between "request" and "service". Also, for every service, eventually a report must be generated (constraint "response"). The car must be checked when a problem is identified (constraint "responded existence"). However, in case of a car with a long period without checks, employees can decide to schedule check even if no serious problems were identified. At most one check will be performed during rental (constraint "0..1").

Process Execution. Execution of activities in a process instance creates a history trace for that instance (cf. Definition 9). The history of a process instance is a chronologically ordered list of events that occurred in the instance. During execution, the state of every constraint (cf. Definition 8) is depicted by a color: (1) green for satisfied, (2) orange for temporarily violated and (3) red for violated. We do not allow the execution of activities that would permanently violate mandatory constraints (depicted by solid lines). The user will be warned for violation of optional constraints (depicted by dashed lines), but he is free to choose to violate optional constraints. Closing an instance is only allowed when all mandatory constraints are satisfied. Again, warnings are given when an instance that is closed does not satisfy all optional constraints, but the user is free to close the instance anyway.

Constraint semantics (expressed in LTL formulas) are used for the automated execution of ConDec models. Every constraint (LTL formula) is translated into a finite automaton [13]. The constraint is satisfied when the automaton is in an accepting state. If the automaton is not in an accepting state and an accepting state is still reachable, the constraint is temporarily violated. The constraint is permanently violated when the automaton is not in an accepting state and an accepting state is not reachable. Also, one overall automaton (*mandatory automaton*) is generated for the conjunction of LTL formulas of all *mandatory* constraints in the model, and it is used to decide which activities can be executed without violating mandatory constraints (cf. Definition 9).



(a) ConDec model

(b) automaton for [] (curse -> <> (pray))

(c) history: curse, bless, pray

Fig. 5. Illustrative example - an instance with history ⟨curse, bless, pray⟩

For illustration purposes we use a simple ConDec model with three activities (“curse”, “pray” and “bless”) and only one constraint, as shown in Figure 5(a). The response constraint specifies that after every execution of the activity “curse” at least one execution of the activity “pray” has to follow (i.e., $\Box(curse \rightarrow \Diamond(pray))$).

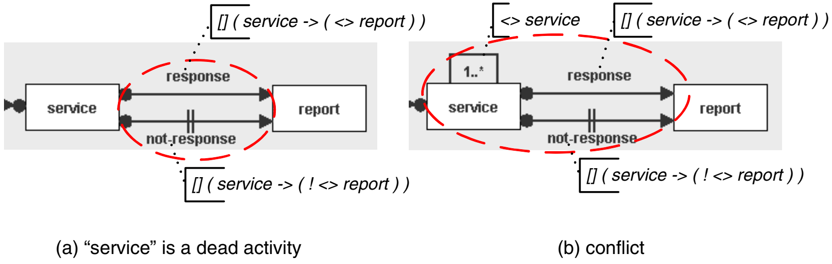
The automaton corresponding to the constraint in the ConDec model is shown in Figure 5(b). The automaton has two states $\{s0, s1\}$. The initial state is `s0`, which is also the accepting state of the automaton [2]. Executing an activity triggers a transition of the automaton.

Let us assume that an instance of the model presented in Figure 5(a) has history ⟨curse, bless, pray⟩. Figure 5(c) shows how this execution history determines the states of the automaton [3]. Initially, the automaton is in its accepting state `s0` (the response constraint is satisfied). Next, activity “curse” triggers a transition to state `s1`. State `s1` is not an accepting state, but an accepting state (`s0`) is reachable from it (the response constraint is temporary violated). Execution of activity “bless” triggers transition “`!pray`” and the automaton remains in state `s1`. Finally, activity “pray” transfers the automaton to accepting state `s0` (the constraint model is satisfied again).

Verification of ConDec models. For correct execution it is important that models do not contain errors. Errors can be discovered in a ConDec model using the mandatory automaton of that model. First, if there is no transition in the automaton that can be triggered by an activity then this activity is dead (cf. Definition 13). Second, if the automaton is empty (has no states and no transitions) then the model has a conflict (cf. Definition 14). It is possible to detect the smallest subset of constraints that causes the error by searching through the powerset of all mandatory constraints in the model. To

[2] Termination of instances is possible only if the automaton is in an accepting state (`s0` in our case).

[3] Note that Figure 5(b) shows a simplified deterministic automata for the response formula. The automata generated from LTL formulas are in general non-deterministic automata [13]. The standard determinization procedure [21] can be used to build a deterministic automaton.



(a) “service” is a dead activity (b) conflict

Fig. 6. Errors independent from history: dead activities and conflicts

achieve this, conjunction automata for subsets of constraints are created and analyzed. If an error is found in a subset, its supersets will be discarded during the search because all of them will contain the same error. This kind of verification can be performed on new models and during run-time changes (Section 4).

Activity “service” in the model given in Figure 6(a) is a dead activity due to the combination of the response and the not-response constraints, while other constraints in the model in Figure 4 do not contribute to this error. The response constraint expresses that every time “service” is executed, “register” has to be executed afterwards at least once. The not-response constraint specifies exactly the opposite, namely that activity “register” cannot be executed after activity “service”. As long as activity “service” is not executed in the model, both constraints are fulfilled. However, as soon as activity “service” is executed for the first time, it becomes impossible to fulfill both constraints. Therefore, we can not allow execution of activity “service” in any instance of this model.

Figure 6(b) shows a model with a conflicting combination of constraints. There is no possibility to satisfy: (1) the existence constraint (“1..*”) on “service”, (2) the response constraint on “service” and “report” and (3) the not-response constraint on “service” and “report”. Therefore, the combination of these three constraints causes a conflict.

3.2 Declare Tool

We developed the Declare tool for development and enactment of declarative process models. Declare can support various languages based on constraint templates as described in Section 3. Specifying languages in the tool is relatively easy – languages and constraint templates can be added, deleted, changed. Each language should include templates specific for a certain domain. For example, the DecSerFlow language [4] has been developed for web-service domain. This language is very similar to ConDec and it contains more than twenty constraint templates. Currently, Declare stores the semantics of constraint templates as LTL formulas, but it is implemented in a way that other formalization languages can be used. Templates are used in Declare to quickly define constraints in models: a template is first selected and activities form the model are assigned to the parameters of the template (cf. Figure 3). In this way, knowledge of the semantics formalization language (e.g., LTL) is not necessary for the development of models in Declare. Declare consists of three tools (see Figure 3.2): (1) the *Designer* is

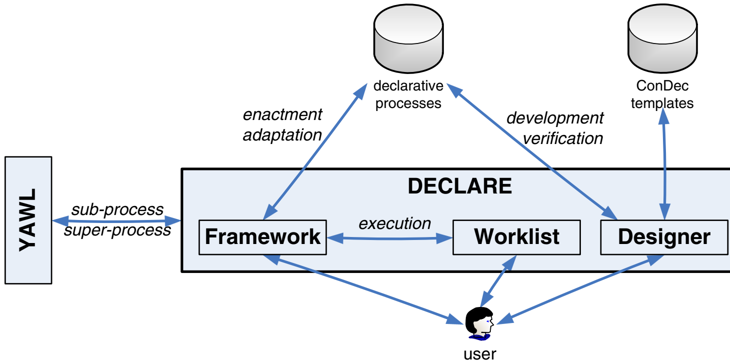


Fig. 7. Declare tool

used for the specification of languages (e.g. DecSerFlow, ConDec, etc.), specification of constraint templates, development of process models; (2) the *Framework* is the execution engine where process instances can be launched, run-time changes can be applied to instances, etc. and (3) the *Worklist* is a simple tool that each user uses to execute process instances. Declare works together with the YAWL workflow management system (`www.yawl-system.com`) [2]. On one hand, a DECLARE process can be implemented as a sub-process of a YAWL process. On the other hand, a YAWL process can be implemented as a sub-process of a DECLARE process. Therefore, it is possible to have workflows which are partly procedural and partly declarative. In the next section we show how Declare is extended to support changes.

4 Change in ConDec

Thanks to the usage of automata (cf. Section 3.1) it is fairly easy to change ConDec models for already running instances. ConDec supports both ad-hoc and evolutionary changes as defined in Section 2.3 (cf. Definitions 15, 16 and 17).

Procedure for change. The procedure for changing an instance during the execution is slightly different from the procedure for starting an instance. Figure 8 shows how the DECLARE tool performs both procedures. When an instance of a verified constraint model is started, an automaton is created for the mandatory constraints and is set to the initial state. After these steps, the instance starts executing with the constraint automaton in its initial state. During change, one additional step, called *instance verification* (cf. Definition 8), is needed to determine whether the history trace satisfies the constraints of the new (changed) model. Instance verification could reveal *history violations*. A history violation is a permanent violation of mandatory constraints of the new model. It occurs when the history (generated by the automaton for the old model) cannot be replayed by the new automaton (for the changed model). Change is only allowed in the absence of history violations. After change, the instance continues its execution according to the new model. The state of the new automaton is the state that was set by

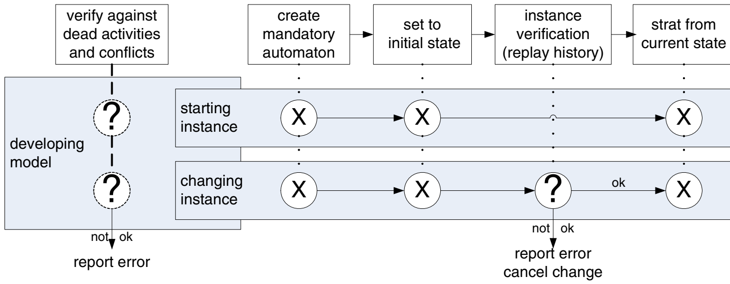


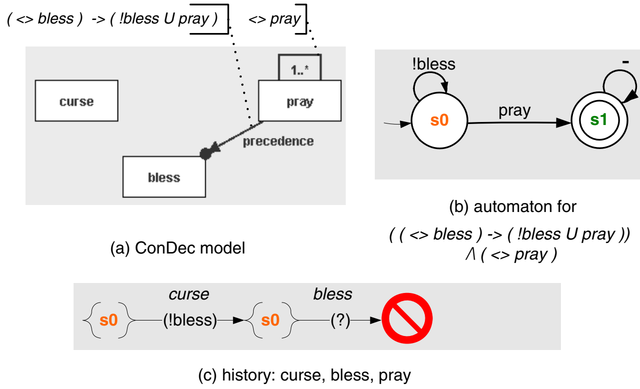
Fig. 8. Procedure for starting and changing instances

the history trace, i.e. history is "replayed" in the new model. In cases of history violation, the minimal subset of constraints causing the violation is detected using the same technique as we used for dead activities and conflict verification.

Change operations. Changes of ConDec models in the DECLARE tool can be achieved by: (1) adding constraints, (2) removing constraints, (3) adding activities and (4) removing activities. Using combinations of these four atomic change operations it is also possible to change constraint types from mandatory to optional and vice versa in running instances. For example, changing type of a constraint from optional into mandatory can be decomposed into two atomic actions: an optional constraint is removed and an mandatory constraint is added to the model.

The automaton used for execution of an instance is generated based on all mandatory constraints in the instance model. Therefore, this automaton will change only when adding/removing mandatory constraints. In cases of other changes (i.e., adding and removing activities and optional constraints), the execution automaton will remain the same like before the change. Due to this fact, history violations can only occur when adding mandatory constraints to the constraint model. When an activity is added (or removed) in the running instance, its execution automaton will remain the same, but the users will (or will not) be able to execute the activity in the future for the running instance (cf. Definition 9). When removing an activity involved in one or more constraints, one of the two strategies can be adopted: (1) the change operation is rejected and the activity cannot be removed or (2) the activity and all related constraints are removed from the model. Currently, the second strategy is implemented in Declare.

Figure 9(a) shows a ConDec model where a precedence constraint has been added to the model. The precedence constraint specifies that each "bless" activity can be executed only after at least one execution of activity "pray". Figure 9(b) shows the automaton for all constraints of the ConDec model. In the automaton the "bless" activity is only allowed after execution of the "pray" activity (state `s1`). Figure 9(c) shows a history violation for trace ⟨curse, bless, pray, bless⟩. The violation is caused by the fact that the automaton is unable to execute "bless" in `s0`. Therefore, adding the "precedence" constraint is not allowed for instances with this history trace. Figure 10 shows a screenshot of Declare reporting this history violation for the "precedence" constraint.



(a) ConDec model

(b) automaton for ((<> bless) -> (!bless U pray)) /\ (<> pray)

(c) history: curse, bless, pray

Fig. 9. "Precedence(pray,bless)" violates history ⟨curse, bless, pray⟩

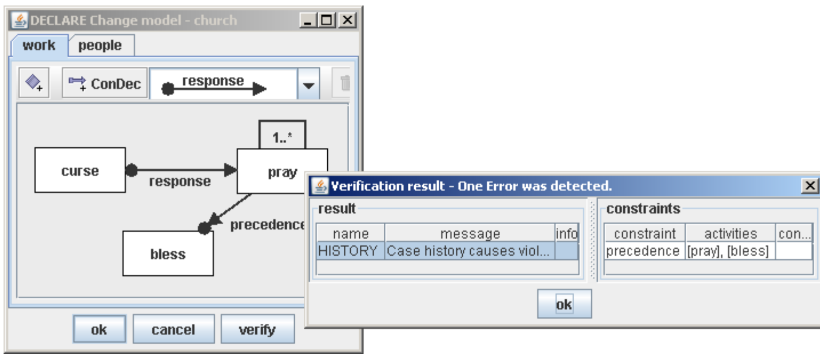


Fig. 10. Declare screenshot - precedence constraint violating history ⟨curse, bless, pray⟩

Note that there might be cases where a group of constraint causes a history violation. Declare searches for the smallest subset of constraints that causes an verification error (dead activity, conflict or history violation) by searching through the powerset of all mandatory constraints as described in Section 3.1.

Change types. The DECLARE tool supports both total and partial evolutionary change of ConDec models. For an ad-hoc change, instance verification is only performed for the relevant instance. In traditional languages migration of instances is complicated and not always possible. The complexity of this operation stems from the fact that for the current state of an instance, an appropriate state in the new model has to be found and this is not always possible (cf. the "dynamic change bug" described in [11] and the many problems described in [19]). In declarative languages, such as ConDec, it is not necessary to find such a state. DECLARE only investigates the state of the new mandatory constraint automaton for the history trace(s), to detect whether migration

is possible. For evolutionary changes, DECLARE performs instance verification on all instances of the old constraint model. Instances that do not cause history violation are migrated to the new model. Other instances continue execution with the old model.

5 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, workers want lots of flexibly and to feel unconstrained in their actions [1,3,4,6,7,8,9,11,16,18,19,23]. 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.

See [3] for a taxonomy of change and [12] for an introduction to the different types of workflow processes. The case handling concept is advocated as a way to avoid restricting users in their actions [6]. This is achieved by a range of mechanisms that allow for implicit deviations that are rather harmless. In [8] completely different techniques are used, but also the core idea is that implicit paths are generated to allow for more flexibility. In [7] 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. Many papers look at problems related to ad-hoc and/or evolutionary change [1,9,11,18,19,23]. The problem of the dynamic change bug was introduced in [11]. In [1] this problem is addressed by calculating so-called change regions based on the structure of the process. A particular correctness property is described in [23] and the problem of instance migration is also investigated in [9]. In the context of the ADEPT system the problem of workflow change has been investigated in detail (including data analysis) [18, 19].

It is also interesting to mention some commercial workflow management systems 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 [6]; 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 ConDec [16] and DecSerFlow [4] where a more declarative style of modeling is advocated. In those papers, the problem of change is not addressed, i.e., the goal is to avoid change. Despite various approaches to declarative (and constraint-based) workflow specification [10, 22], as far as we know, this paper is the first paper that investigates the possibility of allowing ad-hoc and evolutionary changes in a constraint-based language.

6 Conclusions

This paper presented a new and comprehensive approach towards supporting change in constraint-based workflow models. This approach combines the advantages of having a declarative style of modeling and allowing ad-hoc and evolutionary changes. On

the one hand, we try to avoid over-specification by using a declarative style of modeling rather than the typical procedural styles used in today's workflow management systems. On the other hand, we acknowledge the fact that sometimes change is unavoidable and provide extensive support for this. The results presented in this paper show that it is relatively easy to support ad-hoc and evolutionary changes in constraint-based workflow models.

In this paper, we presented a general approach and also showed a concrete application of the ideas using the ConDec language [16]. Moreover, the whole approach is supported by the Declare system. The reader is invited to download the tool from `http://is.tm.tue.nl/staff/mpesic/declare.htm`. Declare works together with the YAWL workflow management system [2] (`www.yawl-system.com`) that also allows for flexibility through so-called worklets [7]. This enables developing workflows which are partly procedural and partly declarative while using all kinds of flexibility mechanisms. Future work will aim at experimenting with interesting mixtures of these mechanisms, e.g., to provide guidelines on when to use particular types of flexibility.

References

1. van der Aalst, W.M.P.: Exterminating the Dynamic Change Bug: A Concrete Approach to Support Workflow Change. Information Systems Frontiers 3(3), 297–317 (2001)
2. van der Aalst, W.M.P., Aldred, L., Dumas, M., ter Hofstede, A.H.M.: Design and Implementation of the YAWL System. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 142–159. Springer, Heidelberg (2004)
3. van der Aalst, W.M.P., Jablonski, S.: Dealing with Workflow Change: Identification of Issues and Solutions. International Journal of Computer Systems, Science, and Engineering 15(5), 267–276 (2000)
4. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer, Heidelberg (2006)
5. van der Aalst, W.M.P., Pesic, M.: Specifying, discovering, and monitoring service flows: Making web services process-aware. BPM Center Report BPM-06-09, BPM Center (2006), `http://www.BPMcenter.org`
6. van der Aalst, W.M.P., Weske, M., Grünbauer, D.: Case Handling: A New Paradigm for Business Process Support. Data and Knowledge Engineering 53(2), 129–162 (2005)
7. Adams, M., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows. In: Meersman, R., Tari, Z. (eds.) On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE. LNCS, vol. 4275, pp. 291–308. Springer, Heidelberg (2006)
8. Agostini, A., De Michelis, G.: Improving Flexibility of Workflow Management Systems. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) Business Process Management. LNCS, vol. 1806, pp. 218–234. Springer, Heidelberg (2000)
9. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow Evolution. Data and Knowledge Engineering 24(3), 211–238 (1998)
10. Dourish, P., Holmes, J., MacLean, A., Marqvardsen, P., Zbyslaw, A.: Freeflow: mediating between representation and action in workflow systems. In: CSCW 1996. Proceedings of the 1996 ACM conference on Computer supported cooperative work, pp. 190–198. ACM Press, New York, NY, USA (1996)

11. Ellis, C.A., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Comstock, N., Ellis, C., Kling, R., Mylopoulos, J., Kaplan, S. (eds.) ACM SIGOIS. Proceedings of the Conference on Organizational Computing Systems, Milpitas, California, pp. 10–21. ACM Press, New York (1995)
12. Georgakopoulos, D., Hornick, M., Sheth, A.: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Distributed and Parallel Databases 3, 119–153 (1995)
13. Giannakopoulou, D., Havelund, K.: Automata-based verification of temporal properties on running programs. In: ASE 2001. Proceedings of the 16th IEEE international conference on Automated software engineering, p. 412. IEEE Computer Society Press, Washington, DC, USA (2001)
14. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts and London, UK (1999)
15. Milner, R.: Communicating and Mobile Systems: The Pi-Calculus. Cambridge University Press, Cambridge, UK (1999)
16. Pesic, M., van der Aalst, W.M.P.: A Declarative Approach for Flexible Business Processes. In: Eder, J., Dustdar, S. (eds.) Business Process Management Workshops. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
17. Petri, C.A.: Kommunikation mit Automaten. PhD thesis, Fakultät für Mathematik und Physik, Technische Hochschule Darmstadt, Darmstadt, Germany (1962)
18. Reichert, M., Dadam, P.: ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. Journal of Intelligent Information Systems 10(2), 93–129 (1998)
19. Rinderle, S., Reichert, M., Dadam, P.: Correctness Criteria For Dynamic Changes in Workflow Systems: A Survey. Data and Knowledge Engineering 50(1), 9–34 (2004)
20. Scheer, A.-W.: ARIS: business process modeling, 2nd edn. Springer, Berlin (1998)
21. Sudkamp, T.-A.: Languages and machines: an introduction to the theory of computer science, 2nd edn. Addison-Wesley Pub., Reading (1997)
22. Wainer, J., de Lima Bezerra, F.: Groupware: Design, Implementation, and Use. In: Favela, J., Decouchant, D. (eds.) CRIWG 2003. LNCS, vol. 2806, pp. 151–158. Springer, Heidelberg (2003)
23. Weske, M.: Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System. In: Sprague, R. (ed.) HICSS-34. Proceedings of the Thirty-Fourth Annual Hawaii International Conference on System Science, IEEE Computer Society Press, Los Alamitos, California (2001)