

Corrective Evolution of Adaptable Process Models

Luciano Baresi¹, Annapaola Marconi², Marco Pistore², and Adina Sirbu²

¹ Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
bares@elet.polimi.it

² Fondazione Bruno Kessler (FBK), Trento, Italy
{marconi,pistore,sirbu}@fbk.eu

Abstract. The aim of corrective evolution is to update a process model by plugging in successful process instance adaptations, while ensuring that the resulting model satisfies the goal of the original model. Corrective evolution is necessary if adapting at runtime is costly, or if adaptations fail and should be avoided. Considering that a trace is a recording of executed activities, we identify three possibilities for correcting process models, depending on the traces on which the instance adaptation should be plugged in. A correction is *strict* if the adaptation should be plugged in on a precise trace, *relaxed* if on all traces, and *relaxed with conditions* if on a subset of all traces. Automated techniques for corrective evolution are necessary since changing models manually is difficult and there is a need to verify that the evolved model satisfies the goal. We provide automated solutions for two cases: when corrections are strict, and when they are either strict or relaxed. We evaluate the tradeoffs between strict and relaxed corrections using a real log.

1 Introduction

Modeling business processes is a complex task which can be simplified by allowing process instances to be structurally adapted at runtime, based on context (e.g., by adding or deleting activities). The process model then no longer needs to include a handling procedure for every exception that can occur. Instead, it only needs to include the assumptions under which a successful execution is guaranteed. If a design-time assumption is violated, an exception is triggered and the exception handling procedure matching the context is selected or constructed at runtime (e.g., [2,9,4]). However, if runtime structural adaptation is allowed, the process model may later need to be updated based on instance adaptations. For example, model updates are necessary if dealing with a frequent exception at runtime is too costly, or if an adaptation fails and should be avoided.

Evolving the process model based on instance adaptations or process variants is not new, and has already been addressed in, e.g., [7,10,12,19]. However, an issue insufficiently addressed in the previous work is how to evolve a process model and also ensure that the evolved model continues to achieve the goal of the original model. We refer to the problem of evolving a process model based on selected instance adaptations, such that the evolved model satisfies the goal of the original model, as *corrective evolution*.

We illustrate the need for corrective evolution on a car logistics scenario, in which cars arrive from manufacturers at a sea port and must be delivered to a retailer. The car handling process includes storing the car, waiting for a delivery order, performing treatments (e.g., painting), and delivering. The goal is to deliver the car to the retailer

in perfect condition. Cars may be damaged at any point. The repair can be performed immediately or can be postponed, depending on context (e.g., availability of resources). To consider at every step that the car can be damaged, and all the contexts in which the damage can occur, would complicate the process model significantly. Instead, we specify a constraint that the car should not be damaged. If this constraint is violated, the process instance will be adapted based on context. By analyzing the logs, we determine that if the car is damaged while being stored, and other cars are waiting to be repaired, the repair should be postponed. If the car is damaged a second time in the storage area, unless many cars are waiting, the repair should not be postponed anymore, such that the car is delivered on time. We therefore need to evolve the process model, such that the repair is postponed in the first situation, and performed immediately in the second. Moreover, the evolved model must satisfy the goal to deliver the car in perfect condition.

When plugging an instance adaptation at a certain point in the process model, we need to consider that there can be multiple paths to reach this point in the model. Each path is uniquely identified by a trace, i.e., a recording of the executed activities. For each adaptation there are three options, depending on which traces the adaptation should be plugged in. To the best of our knowledge, this distinction is not present in the literature.

- The first option, which we call a *strict correction*, is to plug in the adaptation only for the trace corresponding to the instance where the adaptation was used. The advantage is that the resulting model will contain only known behavior.

- The second option, or *relaxed correction*, is to plug in the adaptation on all the traces leading to the specified point. Although more behavior is introduced, the resulting model should be smaller than the model obtained with strict corrections. The reason is that applying strict corrections requires unfolding the model.

- Relaxed corrections are not always possible, since the resulting model must also satisfy the goal. The third option, *relaxed correction with conditions*, is to plug in the adaptation for a subset of the traces leading to the specified point.

In our scenario, we obtain different process models depending on the type of the two corrections. If both corrections are strict, the second adaptation is applied only for the trace when the car is damaged a second time, after having postponed the repair for the first damage. If the second adaptation is plugged in as a relaxed correction, it will be applied also for the traces where the car is damaged for the first time.

Automating corrective evolution is necessary since changing complex process models manually is difficult, and, unless all corrections are strict, there is a need to verify that the evolved model satisfies the goal. We provide automated solutions for two cases: when all corrections are strict, and when they are either strict or relaxed. These two cases do not require searching for the traces where to plug in the adaptations. The first case also does not require verification, since both process model and adaptations are assumed to satisfy the goal, when adaptations are applied only on corresponding traces.

The contributions of this paper are:

- we extend the existing work on process evolution by considering the problem of ensuring that the evolved model continues to achieve the goal of the original model.

- we identify three ways in which instance adaptations can be plugged into the model: strict, relaxed, and relaxed with conditions, and motivate their usage.

- we provide automated solutions for two special cases of corrective evolution.

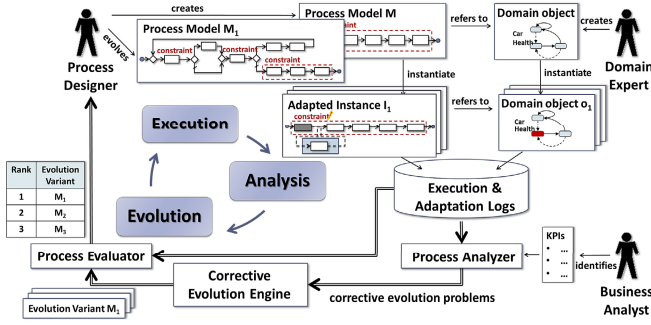


Fig. 1. Process-based application lifecycle

Section 2 presents concepts related to modeling, executing, and adapting a process-based application. We introduce the corrective evolution problem in Section 3, and solve the two cases in Section 4. We evaluate tradeoffs between strict and relaxed corrections in Section 5. Related work is discussed in Section 6, and future work in Section 7.

2 Background

We model the business logic using *process models* and the domain knowledge using *domain objects*. We relate the process to the domain by defining *goals* and process annotations based on domain objects. Fig. 1 shows the application lifecycle. In the *execution* phase, process models and domain objects are instantiated, and process instances are executed. Execution and adaptation events are recorded into logs, which are examined during *analysis*. Adaptations which are successful according to performance indicators may be selected to be included in the model, triggering *evolution*. During *evolution*, new process models are created and ranked according to, e.g., size or complexity.

2.1 Application Representation

Domain Objects. A domain object is a state transition system representing a property of an entity. States correspond to property values; value changes are transitions between states triggered by events. Controllable events are triggered by executing a process, while uncontrollable events can happen at any time and cannot be triggered directly.

Definition 1 (Domain Object). A domain object is a tuple $\langle L, L^0, \mathcal{E}, T \rangle$, where

- L is a finite set of states and $L^0 \subseteq L$ a set of initial states;
- \mathcal{E} is a set of events partitioned into sets: controllable \mathcal{E}_C , and uncontrollable \mathcal{E}_U ;
- $T \subseteq L \times \mathcal{E} \times L$ is a transition relation based on events.

Let O be a set of domain objects. We denote with P_S the set of propositions $s^S(o)$, where $o = \langle L, L^0, \mathcal{E}, T \rangle \in O$ and $s \in L$, and with $Bool(P_S)$ the set of boolean expressions over P_S . Similarly, P_E denotes the set of propositions $e^E(o)$, where $e \in \mathcal{E}$.

The domain objects in our scenario are shown in Fig. 2.

Goals. Goals specify desirable states to be reached and then continuously maintained during the execution of a process. We express goals in terms of domain objects.

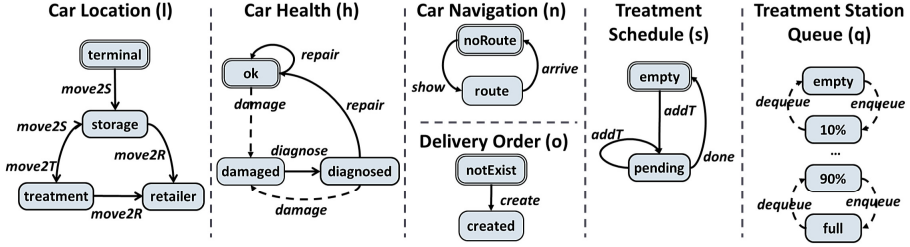


Fig. 2. Domain objects in the car logistics scenario

Definition 2 (Goal). Let O be a set of domain objects. A goal defined over O is a set of goal statements, where each goal statement is defined with the generic template $\psi_0 \implies (\psi_1 \succ \dots \succ \psi_n)$, where $\psi_i \equiv \top \mid s^S(o) \mid \psi_i \vee \psi_i \mid \psi_i \wedge \psi_i$ and $o \in O$.

A goal statement specifies that whenever the state in the left side occurs, the process should reach the state defined by the right side. If the left side is empty (\top), the state in the right side should be reached unconditionally. The states in the right side are ordered using a preference operator (\succ). In our scenario, the goal is to have the car delivered to the retailer in perfect condition: $\top \implies ok^S(h) \wedge retailer^S(l)$ (G_1)

Process Models. A process model is a directed graph for which the nodes are either activity nodes or control connectors, connected by control edges. Activity nodes are labeled with activities. Activities are atomic and can correspond to more than one node, i.e., duplicate nodes are allowed. We relate activities to the domain through preconditions and effects. The preconditions are boolean formulas over domain object states, while the effects correspond to controllable events. An activity can be executed only if the precondition holds, and executing the activity triggers the events in the effects.

Definition 3 (Activity). An activity is a tuple $\langle a, pre, eff \rangle$ defined over a set of domain objects O : a is the name, $pre \in Bool(P_S)$ the precondition, and $eff \subseteq P_E$ the effects.

We model the control flow using edges and the control connectors And/XorSplit, And/XorJoin. These constructs realize the patterns: sequence, parallel split, synchronization, exclusive choice, simple merge, and arbitrary loop, which form the core of any process modeling language. Our representation can therefore easily be mapped to, e.g., BPMN, WS-BPEL, or modeling languages with formal semantics.

Control edges connecting XorSplit nodes with other nodes can be annotated with conditions. A scope with constraint C is a sequence of activities with precondition C .

Definition 4 (Process Model). Let O be a set of domain objects and \mathcal{A} a set of activities defined over O . A process model M over O and \mathcal{A} is a tuple $\langle N, E, l, t, c \rangle$ where:

- N is a finite set of nodes partitioned into sets: N_A of activity nodes, and N_C of control connectors;
- $E \subseteq N \times N$ is a set of directed edges;
- $l : N_A \rightarrow \mathcal{A}$ is a function mapping activity nodes to activities;
- $t : N \rightarrow \{Start, End, Normal, XorSplit, XorJoin, AndSplit, AndJoin\}$ is a total function assigning a type to each node;

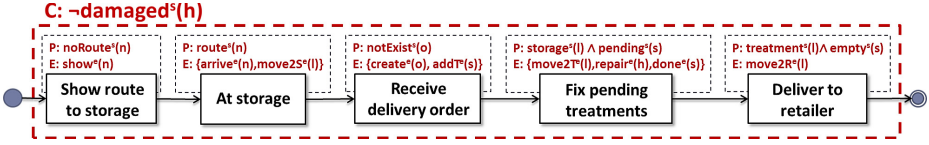


Fig. 3. Car process model

- $c : E \rightarrow \text{Bool}(P_S)$ is a branch condition function such that for all $e = (n_1, n_2) \in E$, if $c(e)$ is defined then $t(n_1) = \text{XorSplit}$.

Fig. 3 shows the process model in our scenario. For example, the precondition of *Receive delivery order* is that a delivery order should not exist; the effects are that the order is created and treatments are added to the schedule. $\neg\text{damaged}^S(h)$ is constraint, which means that each activity includes this formula in its precondition.

2.2 Execution

The *trace* of a process instance is the sequence of executed activities.

Definition 5 (Trace). Let $M = \langle N, E, l, t, c \rangle$ be a process model defined over O and \mathcal{A} . A trace π on M is a sequence of activities $\langle a_1, \dots, a_k \rangle$, $k \in \mathbb{N}$, such that $\forall i, 1 \leq i \leq k, a_i \in \mathcal{A}$, and $\exists n_i \in N, l(n_i) = a_i$, with $t(n_1) = \text{Start}$. For $i < k$, n_i and n_{i+1} are such that either $(n_i, n_{i+1}) \in E$, or $\exists n'_1, \dots, n'_j \in N_C, j \geq 1$, and $(n_i, n'_1), (n'_1, n'_2), \dots, (n'_j, n_{i+1}) \in E$. Activities can occur multiple times due to loops and duplicate nodes. A trace is complete if $t(n_k) = \text{End}$, and partial otherwise.

A *configuration* is a global state of the domain objects at a certain point during the execution of a process instance.

Definition 6 (Configuration). Let O be a set of domain objects. A configuration γ of O is a total function which maps each domain object $o \in O, o = \langle L, L^0, \mathcal{E}, T \rangle$ to a state in L . If γ maps every object o to an initial state in L^0 then γ is an initial configuration.

Since every object state is described by a proposition in P_S , a configuration γ corresponds to an interpretation I_γ of P_S . Slightly abusing the notation, we say that γ satisfies $b \in \text{Bool}(P_S)$ and write $\gamma \models b$, if $I_\gamma \models b$. γ' is directly reachable from γ if for every $o \in O$ for which $\gamma(o) \neq \gamma'(o)$, there exists a sequence of transitions in o from $\gamma(o)$ to $\gamma'(o)$ only on uncontrollable events. Activity $\langle a, \text{pre}, \text{eff} \rangle$ is applicable in γ if there exists γ_{pre} directly reachable from γ such that $\gamma_{\text{pre}} \models \text{pre}$. Then, γ_{eff} is reachable by applying a in γ if a is applicable in γ and by applying eff to γ_{pre} we obtain γ_{eff} .

Definition 7 (Execution). Let $M = \langle N, E, l, t, c \rangle$ be a process model defined over O and \mathcal{A} . An execution of M is an alternating sequence of configurations and activities represented as $\gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$, where:

- $a_1, \dots, a_k \in \mathcal{A}$, and $\langle a_1, \dots, a_k \rangle$ is a trace on M ,
- $\gamma_0, \dots, \gamma_k$ are configurations of O , with γ_0 an initial configuration,
- $\forall i, 1 < i \leq k$, if $n_{i-1}, n_i \in N$ are the nodes corresponding to a_{i-1} and a_i , and $a_i = \langle \text{name}_i, \text{pre}_i, \text{eff}_i \rangle$, then:

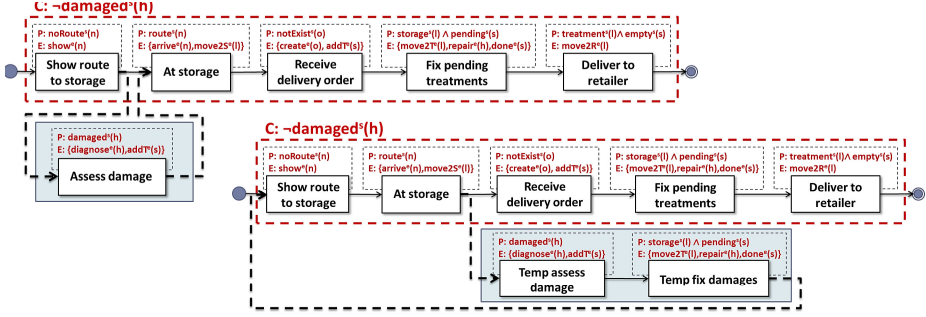


Fig. 4. Adaptation operations: (1) Schedule repair, (2) Repair temporarily

- if $(n_{i-1}, n_i) \in E$, then γ_i is reachable from γ_{i-1} by applying a_i ,
- otherwise, γ_i is reachable from γ_{i-1} by applying $a'_i = \langle \text{name}_i, \text{pre}_i \wedge \varphi, \text{eff}_i \rangle$, where φ is the conjunction of conditions between n_{i-1} and n_i .

An execution is complete if the trace $\langle a_1, \dots, a_k \rangle$ is complete, and partial otherwise.

We denote with $Exec(M)$ the set of complete executions that can be produced by model M . $Exec(M)$ can be infinite if M contains loops. M satisfies a goal statement $\psi_0 \implies (\psi_1 \succ \dots \succ \psi_n)$ if every complete execution of M is such that if a configuration satisfying ψ_0 is reached during the execution, then the last configuration satisfies at least one of ψ_1, \dots, ψ_n . M satisfies a goal G if it satisfies all the statements in G . The model in Fig. 3 satisfies goal G_1 . For every complete execution of M , every configuration reached satisfies \top , and the last configuration satisfies $ok^S(h) \wedge \text{retailer}^S(l)$.

2.3 Adaptation

Process instances can be adapted structurally while they are running: activities can be added, removed, re-executed. We start from the premise that adaptation is triggered by a constraint violation. Based on this premise, plugging an adaptation into the process model in the evolution phase results in an enhancement of the model, which allows to insert several adaptations at the same time. A second premise is that adaptation is performed to reach the goal. This ensures that there is at least one situation for which the adaptation can be plugged in the model such that the resulting model satisfies the goal. These two premises are more restrictive than that of existing process evolution approaches (e.g., [8, 10, 19]), where there are no domain-level restrictions on adaptations.

Given a process model M , an adaptation is an operation $adapt(M^a, from, to)$, where M^a is a process model, and $from, to$ are nodes in M . This operation allows to interrupt an execution of M after having completed $from$, execute M^a , and then resume from to . Adaptations are one-time changes, i.e., if $from$ is reached again, M^a is not re-executed. Without loss of generality, we consider that adaptations cannot contain nodes from the main model and jumping in a parallel branch is not possible. An equivalent adaptation satisfying these conditions can be constructed by duplicating nodes.

Definition 8 (Adaptation). Let $M = \langle N, E, l, t, c \rangle$ be a process model defined over O and \mathcal{A} . An adaptation of M is an operation $\Delta = adapt(M^a, from, to)$, such that:

$M^a = \langle N^a, E^a, l^a, t^a, c^a \rangle$ is a process model defined over O and \mathcal{A} , with $N \cap N^a = \emptyset$, $from, to \in N$, and to is not part of an AND-block.

An adaptation is applicable to a partial execution ω of M only if $from$ is the last completed node in ω , there is at least one execution of M^a starting from the last configuration in ω , and any resulting complete execution satisfies the goal of model M .

We represent adaptation as a single operation rather than using change patterns [18], to emphasize that it is a solution to an exception. If it were represented as multiple change operations, the meaning of an adaptation as an indivisible solution would be lost. Fig. 4 shows two adaptations of the model M from Fig. 3. *Schedule repair* is applicable to M on execution $\omega_1 = \gamma_0 \xrightarrow{\text{Show route to storage}} \gamma_1$. *Repair temporarily* is applicable to M on $\omega_2 = \gamma_0 \xrightarrow{\text{Show route to storage}} \gamma_1 \xrightarrow{\text{At storage}} \gamma_2$. Let $\omega_3 = \gamma_0 \xrightarrow{\text{Show route to storage}} \gamma_1 \xrightarrow{\text{Assess damage}} \gamma_2 \xrightarrow{\text{At storage}} \gamma_3$ be an execution of M adapted by *Schedule repair*. *Repair temporarily* is applicable to M also on ω_3 , which is the case when the constraint is violated a second time.

3 Corrective Evolution

To integrate an adaptation in a process model, we need to specify the point in the model and the condition under which it must be plugged in. We specify a plug-in point as a set of traces. Given a process model M and an adaptation $\Delta = \text{adapt}(M^a, from, to)$, the plug-in point must be such that each trace to this point ends with the activity corresponding to $from$. Given a plug-in point, the restrictions for the condition are:

- at least one configuration reachable at the plug-in point satisfies the condition;
- the next activities in M are not applicable in any plug-in point configuration which satisfies the condition;
- Δ is applicable to every plug-in point configuration which satisfies the condition.

Conditions which satisfy the first two restrictions are *deviating conditions*. A deviating condition represents plug-in point configurations for which the model does not specify how to proceed. A deviating condition which satisfies the third restriction is a *plug-in condition*. The combination of adaptation, plug-in point, and condition is a *correction*.

Definition 9 (Correction). *Let M be a process model defined over O and \mathcal{A} . A correction C is a tuple $\langle ct, \pi, \varphi, \Delta \rangle$ such that:*

- $ct \in \{\text{strict}, \text{relaxed}, \text{with-conditions}\}$ is the correction type;
- π is a partial trace on M ;
- φ is a boolean expression from $\text{Bool}(P_S)$;
- $\Delta = \text{adapt}(M^a, from, to)$ is an adaptation of M .

We say that C is applicable to M if:

- φ is a plug-in condition for applying Δ to M on π ,
- if $ct \neq \text{strict}$, then φ is a deviating condition for M and node $from$.

The point where Δ must be plugged in is determined by the type ct , the trace π , and the node $from$. For all correction types, Δ should be plugged in after $from$, under condition φ . If ct is strict, both φ and Δ should be applied only on π . If relaxed, they should be applied on all traces leading to $from$. Finally, if relaxed with conditions, they should be applied on at least one trace leading to $from$.

Let C be a correction applicable to M . We denote with $Exec(M, C)$ the set of complete executions that can be produced by M corrected by C . $Exec(M, C)$ adds new complete executions to $Exec(M)$. A new execution ω corresponds to a process instance for which a configuration γ satisfying φ is reached at the plug-in point. ω continues with an execution of M^a , and then resumes on M from node to . These new complete executions do not replace any complete executions of M , and $Exec(M) \subseteq Exec(M, C)$. We are interested in retrieving the process model corresponding to $Exec(M, C)$. It can be the case that a model which also satisfies the goal of M does not exist. However, a model M' such that $Exec(M') = Exec(M, C)$ can always be constructed.

Given a process model satisfying a goal, the corrective evolution problem is to apply a sequence of corrections to this model, such that the resulting model satisfies the goal.

Definition 10 (Problem). *Let M_0 be a process model and G a goal such that M_0 satisfies G . Let C_1, \dots, C_n be a sequence of corrections, such that $\forall i, 1 \leq i \leq n$:*

- $C_i = \langle ct_i, \pi_i, \varphi_i, \Delta_i \rangle$, $\Delta_i = adapt(M_i^a, from_i, to_i)$, and to_i is a node from M_0 ;
- C_i is applicable to M_{i-1} , and M_i is such that $Exec(M_i) = Exec(M_{i-1}, C_i)$.

The corrective evolution problem is to find a process model M_n which satisfies G .

Without generality loss, we assume that every adaptation returns the control to M_0 . If Δ_2 returns the control to Δ_1 , an equivalent Δ'_2 can be created by duplicating nodes.

We can now formalize the inputs to the corrective evolution problem in our scenario:

- process model M satisfying the goal G_1 from Section 2.1;
- correction $C_1 = \langle strict, \pi_1, \varphi_1, \Delta_1 \rangle$ where: $\pi_1 = \langle Show\ route\ to\ storage \rangle$, $\varphi_1 = damaged^s(h) \wedge (40\%^s(q) \vee \dots \vee full^s(q))$, Δ_1 is the *Schedule repair* in Section 2.3;
- correction $C_2 = \langle strict, \pi_2, \varphi_2, \Delta_2 \rangle$ where: $\pi_2 = \langle Show\ route\ to\ storage, Assess\ damage, At\ storage \rangle$, $\varphi_2 = damaged^s(h) \wedge (empty^s(q) \vee \dots \vee 70\%^s(q))$, and Δ_2 is the *Repair temporarily* in Section 2.3.

The solution M_n is an enhancement of M_0 , i.e., every complete execution of M_0 is also a complete execution of M_n . For every C_i , there are two possibilities. If C_i is applicable to M_0 , then M_n can replay all the executions that would result by applying C_i to M_0 . It can be the case that C_i is not applicable to M_0 , but it is applicable to M_0 corrected by C_{j_0}, \dots, C_{j_k} , $1 \leq j_0 < \dots < j_k < i$. This is the case when the adaptation must then be plugged in after other adaptations. Then, M_n can replay all the executions that result by applying C_i to M_0 corrected by C_{j_0}, \dots, C_{j_k} .

By formulating the problem as the application of n corrections, rather than only one, we are addressing a more general problem, in the sense that more solutions may be found. One reason is that we verify goal satisfaction only after applying all corrections. Moreover, by applying n corrections, the set of traces on which a relaxed correction (with conditions) can be applied changes depending on the other corrections.

4 Solution

In this section, we solve the case when all corrections are strict (*strict corrective evolution*), and the case when they are either strict or relaxed (*relaxed corrective evolution*).

In the first case, a solution can be constructed naively, by unfolding the process model up to the plug-in point, adding the adaptation, and duplicating the fragment starting

with *to*. Such a naive solution has many duplicated nodes. The challenge is to find a solution with as few duplicated nodes as possible. For this purpose, we encode the process model, domain objects, traces, conditions, and adaptations into state transition systems (STSs). We compute the parallel product of these STSs and obtain an STS which encodes all the executions of the corrected model. We use the correspondences created when encoding the inputs to translate this STS to a new process model.

In the second case, it is necessary not only to compose the process model and adaptations, but also to verify that the composition satisfies the goal. For this purpose, we devise a solution based on planning. We encode the inputs as STSs, encoding also the goal. We use the parallel product as a planning domain and create a planning goal from the process goal. We apply the approach in [1], and generate a controller for the domain to satisfy the planning goal. If a controller exists, we translate it to a new process model.

Since both solutions involve encoding the inputs as STSs, we first introduce some basic STS notions and the encoding of each input. We then present each solution.

An STS contains a set of states, some marked as initial and/or accepting. Each state is labeled with a set of properties. The STS moves to new states as a result of performing actions, which are either *input* (controllable) or *output* (not controllable).

Definition 11 (STS). *Let \mathcal{P} be a set of propositions and $Bool(\mathcal{P})$ the set of boolean expressions over \mathcal{P} . A state transition system is a tuple $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$, where:*

- \mathcal{S} is the set of states and $\mathcal{S}^0 \subseteq \mathcal{S}$ the set of initial states,
- \mathcal{I} and \mathcal{O} are the input and respectively output actions,
- $\mathcal{R} \subseteq \mathcal{S} \times Bool(\mathcal{P}) \times (\mathcal{I} \cup \mathcal{O}) \times \mathcal{S}$ is the transition relation,
- $\mathcal{S}^F \subseteq \mathcal{S}$ is the set of accepting states,
- $\mathcal{F} : \mathcal{S} \rightarrow 2^{\mathcal{P}}$ is the labeling function.

We write $s, \mathcal{F} \models b$ to denote that boolean expression b is satisfied in state s given \mathcal{F} . Transitions are guarded: (s, b, a, s') is possible in s only if $s, \mathcal{F} \models b$.

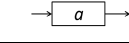
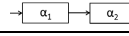

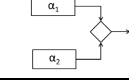
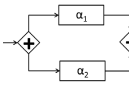
The parallel product of two STSs states that the two STSs move concurrently on common actions, and independently if there are no common actions.

4.1 Encoding into STSs

Encoding Process Models. For a process model M we construct an STS Σ_M by applying the rules in Table 1. α denotes a generic element, $s \xrightarrow{\alpha} s'$ the recursive translation of α . To differentiate input from output actions, names are prefixed with “?” , respectively “!”. For the And-block, Table 1 shows the case with two activities; the STS for a generic And-block allows every possible interleaving combination. Preconditions are copied as guards. Due to the guards, Σ_M does not move if run in isolation; the transitions become enabled in the product of Σ_M with the domain objects STSs. We label each state with a new proposition ($\mathcal{F}(s) = \{s(M)\}$), and mark initial and final states as accepting.

Encoding Adaptations. We define an STS Σ_{Δ_i} for each adaptation Δ_i . We first translate the model M_i^a . Each adaptation realizes a jump in M_0 , and, if applied on $\Delta_j, j < i$, also a reset jump in Δ_j . We encode these jumps using a new action $?resume_i$ and label the start of the jump with $point_i$, to mark the point where Δ_i must be applied. The initial transitions in Σ_{Δ_i} are guarded by $\varphi_i \wedge point_i \wedge trace_i$, where $trace_i$ will mark the end of π_i . If an adaptation is started, an adaptation (not necessarily the same) has to finish for the control to be given to the main process, and the control cannot be given to

Table 1. Encoding process model elements as STSs

Process model element		STS transitions
activity node $n \in N_A$, $l(n) = \langle a, pre, eff \rangle$		$(s_b, pre, ?a, s_e)$
sequence		$s_b \xrightarrow{\alpha_1} s', s' \xrightarrow{\alpha_2} s_e$
XORSplit		$(s_b, \top, ?xor, s_0)$ $(s_0, cond, !case_1, s_1), (s_0, \neg cond, !case_2, s_2)$ $s_1 \xrightarrow{\alpha_1} s_e, s_2 \xrightarrow{\alpha_2} s'_e$
XORJoin		$s_b \xrightarrow{\alpha_1} s_e, s'_b \xrightarrow{\alpha_2} s_e$
AND-block		$(s_b, \top, ?and, s_0)$ $(s_0, \top, !order_1, s_1), (s_0, \top, !order_2, s_2)$ $s_1 \xrightarrow{\alpha_1} s'_1, s'_1 \xrightarrow{\alpha_2} s_e$ $s_2 \xrightarrow{\alpha_2} s'_2, s'_2 \xrightarrow{\alpha_1} s_e$

a previous adaptation. We encode these properties using a semaphore. Σ_{sem} has a state s_0 corresponding to M_0 , and a state s_i for every Δ_i . If Σ_{Δ_i} moves, Σ_{sem} moves from any $s_j, j < i$, to s_i . From s_i it moves to s_0 on $?resume_i$. Each state in Σ_{sem} is labeled with a flag; these flags guard the transitions in $\Sigma_{M_0}, \Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_n}$.

Encoding Traces. We construct an STS Σ_{π_i} for each trace $\pi_i = \langle a_1, \dots, a_k \rangle$. Σ_{π_i} moves from state s_{j-1} to s_j on any action corresponding to a_j , and from s_{j-1} to s_{out} on actions corresponding to any other activity. Σ_{π_i} moves from s_k to s_{out} on any action corresponding to an activity. We label s_k with $trace_i$ to mark the end of π_i .

Encoding Conditions. Conditions can appear in corrections or as edge annotations. Let φ be a condition, and φ' the formula after replacing negative literals. We create an STS Σ_{φ} only if φ' contains a literal reachable through uncontrollable events. We add to Σ_{φ} two actions $!trigger_{\varphi}$ and $!no-trigger_{\varphi}$. The transitions on these actions are guarded, such that φ is triggered only before being evaluated. To simulate the uncontrollability of the events, both cases (when φ holds, and when it does not hold) will be considered.

Encoding Domain Objects. For each object $o = \langle L, L^0, \mathcal{E}, T \rangle$, we define an STS Σ_o with the same states and initial states, marking all states as accepting. We label states with the corresponding propositions, i.e., $\mathcal{F}(s) = \{s^s(o)\}$. For every transition $(s, e, s') \in T$, we consider all the activities a for which $e^e(o) \in eff$. Suppose a appears in M , and $?a$ is an action corresponding to a which can be executed in Σ_M from state s_j . We add to Σ_o a transition from s to s' on $?a$ guarded by $s_j(M)$ and flags. For every transition on an uncontrollable event $(s, e, s_u) \in T, e \in \mathcal{E}_U$, we consider the conditions φ containing $s_u^s(o)$, and add a transition on from s to s_u on $!trigger_{\varphi}$.

Encoding the Goal. We create an STS Σ_g for each statement $\psi_0 \Rightarrow (\psi_1 \succ \dots \succ \psi_k)$. For every ψ we introduce an action $!a_{\psi}$, guarding transitions on $!a_{\psi}$ with ψ . If $!a_{\psi_0}$ is triggered, Σ_g moves to a non-accepting state and waits for one of $!a_{\psi_1}, \dots, !a_{\psi_k}$ to be triggered, in which case it moves to an accepting state. If $!a_{\psi_0}$ is triggered again, Σ_g moves to the non-accepting state. The preference order is encoded as a requirement $\rho = (s_0, \dots, s_k), s_0$ being the initial state, s_1, \dots, s_k the states reached with $!a_{\psi_1}, \dots, !a_{\psi_k}$.

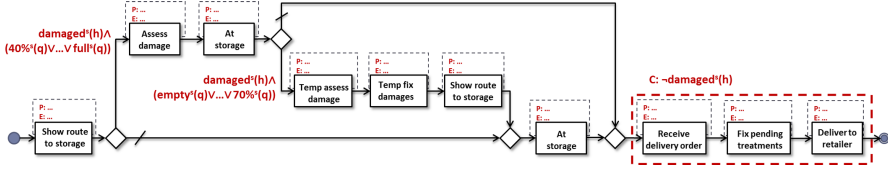


Fig. 5. Strict corrections: corrected process model

4.2 Strict Corrective Evolution

We encode the process model into an STS Σ_{M_0} , adaptations into $\Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_n}, \Sigma_{sem}$, partial traces into $\Sigma_{\pi_1}, \dots, \Sigma_{\pi_n}$, conditions into $\Sigma_{\varphi_1}, \dots, \Sigma_{\varphi_m}$, and domain objects into $\Sigma_{o_1}, \dots, \Sigma_{o_p}$, as described in Section 4.1. We then compute their parallel product: $\Sigma = \Sigma_{M_0} \parallel \Sigma_{\Delta_1} \parallel \dots \parallel \Sigma_{\Delta_n} \parallel \Sigma_{sem} \parallel \Sigma_{\pi_1} \parallel \dots \parallel \Sigma_{\pi_n} \parallel \Sigma_{\varphi_1} \parallel \dots \parallel \Sigma_{\varphi_m} \parallel \Sigma_{o_1} \parallel \dots \parallel \Sigma_{o_p}$

We simplify Σ by removing the transitions which can never fire, i.e., (s, b, a, s') for which $s, \mathcal{F} \not\models b$. We can then remove the guards and the labeling function \mathcal{F} . Σ is nondeterministic, due to the fact that the domain objects STSs can have multiple initial states and nondeterministic transitions. Moreover, if an object can be in more than one state at a certain point in the process, and these states are treated in the same way, Σ will contain many similar transitions. To transform Σ back to a process model, we must first convert it to a deterministic STS. For this, as well as to remove the redundant transitions, we minimize Σ . As criteria for STS equivalence we use complete trace equivalence, one of the weakest notions of behavioral equivalence [5]. The resulting minimal, deterministic STS Σ_{strict} is transformed to a process model.

Theorem 1. Assume a corrective evolution problem defined by a process model M_0 , a goal G , and a sequence of corrections C_1, \dots, C_n , such that $\forall i, 1 \leq i \leq n, C_i = \langle ct_i, \pi_i, \varphi_i, \Delta_i \rangle, ct_i = strict$. Let M_{strict} be the translation of Σ_{strict} . Then M_{strict} is a solution for the problem defined by M_0, G , and C_1, \dots, C_n .

The solution for the corrective evolution problem in our scenario is shown in Fig. 5. The corrected model has two new traces: one corresponding to the application of *Schedule repair*, and one to the application of both *Schedule repair* and *Repair temporarily*.

If M_0 and M_1^a, \dots, M_n^a do not contain parallelism, M_{strict} is the minimal solution to the problem, since the translation from the minimal Σ_{strict} to M_{strict} is direct. However, if any of M_0, M_1^a, \dots, M_n^a contain parallelism, this can be restored by applying post-processing techniques to M_{strict} .

With strict corrections, the original model is unfolded according to the partial traces. If an adaptation includes a backward jump, the activities in between will be duplicated in the new model. This redundancy can be removed by relaxing the corrections.

4.3 Relaxed Corrective Evolution

As in the previous case, we encode the inputs as STSs, this time encoding also the goal. If a correction C_i is relaxed, the trace is ignored and Σ_{π_i} contains one state labeled with $trace_i$. We compute the parallel product of all STSs and remove the transitions which can never fire, followed by the guards and the labeling function. The resulting STS Σ is our planning domain. We construct the planning goal ρ by combining the requirements

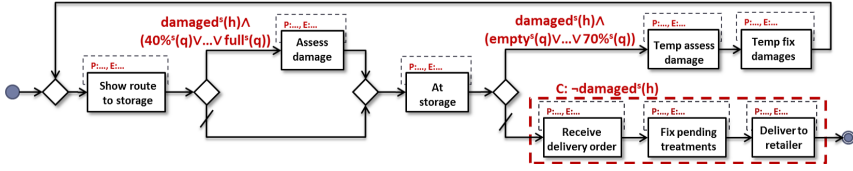


Fig. 6. Relaxed corrections: corrected process model

generated in Section 4.1. On the planning domain Σ and goal ρ we apply the technique in [1], which generates a controller Σ_c such that the controlled system satisfies the goal, i.e., $\Sigma_c \triangleright \Sigma \models \rho$. If Σ_c exists, it corresponds to a synthesis of the original model with the adaptations, which achieves the process goal. We minimize Σ_c and obtain $\Sigma_{relaxed}$, which we translate to a process model.

Theorem 2. Assume a corrective evolution problem defined by a process model M_0 , a goal G , and a sequence of corrections C_1, \dots, C_n , such that $\forall i, 1 \leq i \leq n, C_i = \langle ct_i, \pi_i, \varphi_i, \Delta_i \rangle, ct_i \in \{strict, relaxed\}$. Let $M_{relaxed}$ be the translation of $\Sigma_{relaxed}$. $M_{relaxed}$ is a solution for the problem defined by M_0, G , and C_1, \dots, C_n .

Fig. 6 shows the model obtained in our scenario if both C_1 and C_2 are relaxed. *Schedule repair* and *Repair temporarily* are applied when the car is damaged on the way to storage, respectively at storage, independent of how many damages already occurred.

5 Evaluation

We implemented the two solutions from Section 4 into a prototype tool. For strict corrective evolution, we used the NuSMV model checker (nusmv.fbk.eu). For relaxed corrective evolution, we used WSYNTH, a tool from the ASTRO toolset (astroproject.org).

We evaluated our approach using the event log from the 2012 BPI Challenge. This is a real-life log taken from a financial institute, containing 262.200 events in 13.087 traces. The traces correspond to a loan application process. As a first step, we obtained a rough process model by filtering the log to include the most frequent traces and events, and mining the filtered log. We designed our domain objects based on the log and descriptions, and used these objects to define our goal and annotate the activities appearing in the log. We computed the differences between the model and the traces in the log, and used the most frequent differences to generate strict and relaxed corrections. We then evaluated the tradeoffs between strict and relaxed corrections along three dimensions:

- *fitness* - how much of the behavior in the log is captured by the corrected models;
- *precision* - how much extra behavior is introduced in the corrected models;
- *structure* - how much the corrected models deviate structurally from the original.

To realize these comparisons, we used several metrics devised for evaluating process mining results, which are implemented in the ProM framework (www.promtools.org).

To measure fitness, we used the f metric from [14], implemented in ProM as *token-based fitness*. This is a fine-grained metric quantifying the extent to which the traces in the log can be replayed on the process model. We were interested not only to compare the two correction types, but also to evaluate the fitness of corrected models over time.

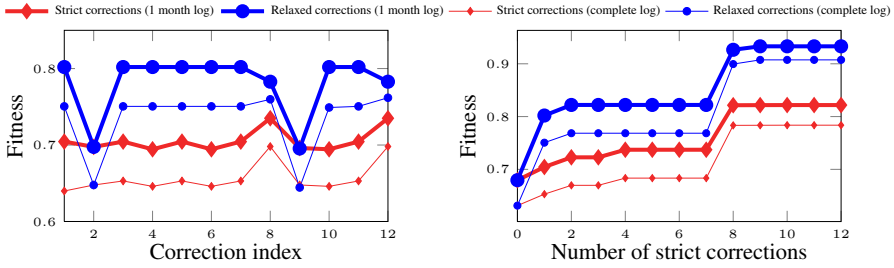


Fig. 7. Fitness: corrections applied (1) individually; (2) incrementally

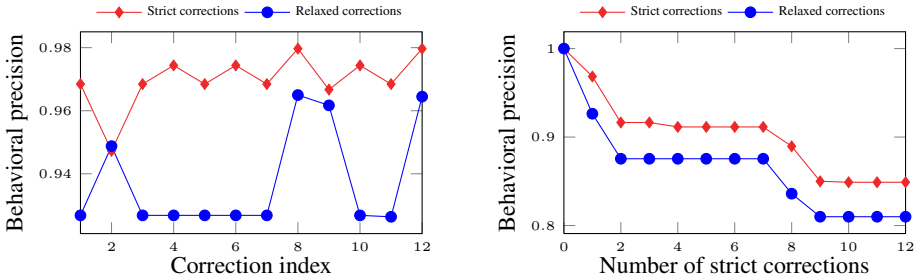


Fig. 8. Behavioral precision: corrections applied (1) individually; (2) incrementally

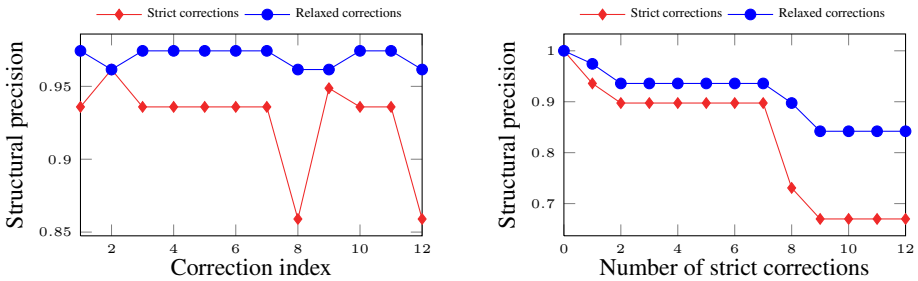


Fig. 9. Structural precision: corrections applied (1) individually; (2) incrementally

To simulate the passing of time, we used only a fragment of the entire log (roughly one sixth, corresponding to the first month) to generate our corrections. We measured the fitness of corrected models on the log fragment as well as on the entire log.

Fig. 7(1) shows the fitness of corrected models when corrections were applied individually, i.e., at each step we applied exactly one correction on the original model. Fig. 7(2) shows the fitness when corrections were applied incrementally. At step 0, we measured the fitness of the original model. Then, in the strict version, at step n we corrected the original model with n strict corrections. In the relaxed version, at step n we corrected the original model with $m \leq n$ relaxed corrections. The number of strict and relaxed corrections is not necessarily equal, since several strict corrections may correspond to the same relaxed correction. As can be seen in Fig. 7, the fitness increases

when corrections are applied, for both correction types. However, the fitness is higher for relaxed corrections, and remains higher also when tested against the entire log.

To measure changes in behavior, we used the *behavioral precision* B_P metric [11]. B_P quantifies how much extra behavior a process model allows with respect to a reference model and a log. B_P is lower when the deviation in behavior is higher. As for fitness, we measured B_P of corrected models when corrections are applied individually and incrementally (Fig. 8). The log used was the one-month log fragment. As expected, we observe that strict corrections introduce less behavior than relaxed corrections.

To measure the deviation in structure, we used the *structural precision* S_P metric [11], which assesses how many connections a process model has that are not in a reference model. Like B_P , S_P is lower when the deviation is higher. We measured S_P of corrected models when corrections are applied individually and incrementally (Fig. 9). We observe that relaxed corrections lead to smaller structural changes. An interesting case is that of corrections 8 and 12, which lead to the lowest values for the strict corrections in Fig. 9(1). For these two strict corrections, the trace passes through an And-block, and to apply the correction the model had to be unfolded up to the plug-in point. No such unfolding was necessary for the corresponding relaxed corrections.

We conclude that for this scenario there is a tradeoff between strict and relaxed corrections: relaxed corrections introduce more behavior, but lead to a higher fitness and less structural changes than strict corrections. In general, we expect relaxed corrections to introduce more behavior and less structural changes as soon as there is more than one trace to the plug-in point. Regarding fitness, a relaxed correction should be more effective if the adaptation is commonly applied at the given point, independent of trace.

6 Related Work

We focus on approaches which use the structural adaptations of process instances to support process evolution. If execution and adaptation logs are available, they can be analyzed to facilitate change reuse (e.g., [16,15]), support process diagnosis (e.g., [6]), and evolve the process model (e.g., [3,19]). Our approach belongs to the last category. In [16], process instances are grouped based on contextual properties, paths, and outcomes, to provide recommendations for improving the process model. For declarative processes, execution recommendations are generated in [15] based on past executions and optimization goals. In [6], process mining techniques are applied to change logs to provide an overview of when and why change was necessary. Also using process mining, [3] repairs a process model with respect to a log, such that the repaired model can replay the log and is similar to the original model. In [19], case-based reasoning is used to log instance changes, and derive suggestions for process model changes.

Alternatively, the result of structural adaptation can be represented as variants of a process model. Techniques for managing variants which use a single reference model to represent a set of variants (e.g., [7,8,10,13]) can also be used for process evolution. In [7] and [13], the reference model incorporates variation points, to distinguish the parts common to all variants from the variant-specific parts. The technique in [8] is used for resolving differences between variants. In [10], a heuristic search is employed to find a process model such that the distance between the model and variants is minimal.

The approaches in [19] and [10] are closest to our work, since they generate new process models based on an adaptation log, respectively process variants. Both approaches derive model changes from frequent instance adaptations. The context of the adaptations is considered in [19], but not in [10]. Further, the trace for which adaptations should be applied is considered implicitly in [10], and not considered in [19]. However, an adaptation is tightly coupled to the context *and* trace for which it is used, and may even be harmful if used for different contexts or traces. When evolving the model based on adaptations, the contexts and traces must be considered as well. If traces are ignored, we need to consider the overall goal of the process, to make sure that the adaptations introduced in the model are not harmful. The goal is not considered in [19], nor in [10]. The relation between context, traces, and goals has been considered in [16]. However, in [16] the aim is to recommend improvements to the process model, rather than to actually change it, and can be used as an analysis technique preceding corrective evolution.

Although goal compliance is insufficiently investigated for process evolution, there are many approaches which address the goal compliance of process models and their runtime adaptation, e.g., [2,9,4]. Also related is the problem of service composition, where a composite service is generated from service interfaces and goal specifications. Among service composition approaches, ASTRO [1] is particularly relevant, since we have used its powerful planning techniques to implement relaxed corrective evolution.

Finally, another relevant area is that of process model refactoring. Of the 11 techniques in [17], our work can be used for implementing RF11, *Pull Up Instance Change*.

7 Conclusions and Future Work

We presented a new approach for evolving process models based on instance adaptations. Our approach ensures that the evolved model achieves the goal of the original model. We identified three different ways for plugging adaptations into the model, and designed automated solutions for two special cases. Finally, we evaluated the tradeoffs between strict and relaxed corrections on a scenario built on a real log. In the future, we will design and evaluate solutions for the general case, when corrections can also be relaxed with conditions. The traces on which such a correction should be applied must be determined through search. The problem gets significantly more complex if more than one such correction should be applied, due to the combinatorial explosion. To deal with this complexity, we will devise heuristic techniques.

References

1. Bertoli, P., Kazhamiakin, R., Paolucci, M., Pistore, M., Raik, H., Wagner, M.: Control Flow Requirements for Automated Service Composition. In: Proc. ICWS 2009, pp. 17–24 (2009)
2. Bucchiarone, A., Marconi, A., Pistore, M., Raik, H.: Dynamic adaptation of fragment-based and context-aware business processes. In: Proc. ICWS 2012, pp. 33–41 (2012)
3. Fahland, D., van der Aalst, W.M.P.: Repairing process models to reflect reality. In: Barros, A., Gal, A., Kindler, E. (eds.) BPM 2012. LNCS, vol. 7481, pp. 229–245. Springer, Heidelberg (2012)

4. Friedrich, G., Fugini, M., Mussi, E., Pernici, B., Tagni, G.: Exception handling for repair in service-based processes. *IEEE Trans. Software Eng.* 36(2), 198–215 (2010)
5. van Glabbeek, R.J.: The linear time-branching time spectrum (extended abstract). In: Baeten, J.C.M., Klop, J.W. (eds.) *CONCUR 1990. LNCS*, vol. 458, pp. 278–297. Springer, Heidelberg (1990)
6. Guenther, C.W., Rinderle-Ma, S., Reichert, M., van der Aalst, W.M., Recker, J.: Using process mining to learn from process changes in evolutionary systems. *Int'l J. of Business Process Integration and Management* 3(1), 61–78 (2007)
7. Hallerbach, A., Bauer, T., Reichert, M.: Capturing variability in business process models: the provop approach. *Journal of Software Maintenance* 22(6-7), 519–546 (2010)
8. Küster, J.M., Gerth, C., Förster, A., Engels, G.: Detecting and resolving process model differences in the absence of a change log. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) *BPM 2008. LNCS*, vol. 5240, pp. 244–260. Springer, Heidelberg (2008)
9. de Leoni, M., Mecella, M., De Giacomo, G.: Highly dynamic adaptation in process management systems through execution monitoring. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) *BPM 2007. LNCS*, vol. 4714, pp. 182–197. Springer, Heidelberg (2007)
10. Li, C., Reichert, M., Wombacher, A.: Discovering reference models by mining process variants using a heuristic approach. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) *BPM 2009. LNCS*, vol. 5701, pp. 344–362. Springer, Heidelberg (2009)
11. de Medeiros, A.K.A., van der Aalst, W.M.P., Weijters, A.J.M.M.: Quantifying process equivalence based on observed behavior. *Data Knowl. Eng.* 64(1), 55–74 (2008)
12. Ploesser, K., Peleg, M., Soffer, P., Rosemann, M., Recker, J.: Learning from context to improve business processes. *BPTRends* 6(1), 1–7 (2009)
13. Reijers, H.A., Mans, R.S., van der Toorn, R.A.: Improved model management with aggregated business process models. *Data Knowl. Eng.* 68(2), 221–243 (2009)
14. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. *Inf. Syst.* 33(1), 64–95 (2008)
15. Schonenberg, H., Weber, B., van Dongen, B.F., van der Aalst, W.M.P.: Supporting flexible processes through recommendations based on history. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) *BPM 2008. LNCS*, vol. 5240, pp. 51–66. Springer, Heidelberg (2008)
16. Soffer, P., Ghattas, J., Peleg, M.: A goal-based approach for learning in business processes. In: *Intentional Perspectives on Information Systems Eng.*, pp. 239–256. Springer (2010)
17. Weber, B., Reichert, M.: Refactoring process models in large process repositories. In: Bellahsene, Z., Léonard, M. (eds.) *CAiSE 2008. LNCS*, vol. 5074, pp. 124–139. Springer, Heidelberg (2008)
18. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data Knowl. Eng.* 66(3), 438–466 (2008)
19. Weber, B., Reichert, M., Rinderle-Ma, S., Wild, W.: Providing integrated life cycle support in process-aware information systems. *Int. J. Cooperative Inf. Syst.* 18(1), 115–165 (2009)