Available online at www.sciencedirect.com
ScienceDirect
Data & Knowledge Engineering 64 (2008) 3–23
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Integration and verification of semantic constraints in adaptive process management systems

Linh Thao Ly *, Stefanie Rinderle, Peter Dadam

Ulm University, Faculty of Engineering and Computer Science, Institute of Databases and Information Systems, James-Franck-Ring, 89069 Ulm, Germany

Accepted 12 June 2007
Available online 23 June 2007

Abstract

Adaptivity in process management systems is key to their successful applicability in practice. Approaches have been already developed to ensure system correctness after arbitrary process changes at the syntactical level (e.g., avoiding inconsistencies such as deadlocks or missing input parameters after a process change). However, errors may be still caused at the semantical level (e.g., violation of business rules). Therefore, the integration and verification of domain knowledge will flag a milestone in the development of adaptive process management technology. In this paper, we introduce a framework for defining semantic constraints over processes in such a way that they can express real-world domain knowledge on the one hand and are still manageable concerning the effort for maintenance and semantic process verification on the other hand. This can be used to detect semantic conflicts (e.g., drug incompatibilities) when modeling process templates, applying ad hoc changes at process instance level, and propagating process template modifications to already running process instances, even if they have been already individually modified themselves; i.e., we present techniques to ensure semantic correctness for single and concurrent changes which are, in addition, minimal regarding the set of semantic constraints to be checked. Together with further optimizations of the semantic checks based on certain process meta model properties this allows for efficiently verifying processes. Altogether, the framework presented in this paper provides the basis for process management systems which are adaptive and semantic-aware at the same time.

Keywords: Semantic correctness; Semantic process verification; Semantic constraints; Adaptive process management systems

1. Introduction

Due to steadily changing conditions at the global market, companies are forced to frequently adapt their business processes [1–4]. Therefore, adaptivity is the key factor for the successful application of process management technology in practice. Generally, process changes can take place at two levels – instance level (e.g., adapting a particular process instance to the needs of a customer) and process template level (e.g., adapting

* Corresponding author. Tel.: +49 731 50 24194; fax: +49 731 50 24134.
E-mail addresses: thao.ly@uni-ulm.de (L.T. Ly), stefanie.rinderle@uni-ulm.de (S. Rinderle), peter.dadam@uni-ulm.de (P. Dadam).

0169-023X/$ - see front matter
doi:10.1016/j.datak.2007.06.007

the process template due to legal changes) [5,6]. Thus, it is crucial for an adaptive process management system (PMS) to support both kinds of changes. However, it is still not sufficient to support process template and instance changes in an isolated manner. An adaptive PMS must also allow for the *interplay* between process template and instance changes [7]. A framework for the support of process template and instance changes as well as for their interplay (i.e., the support of change propagation to already individually modified instances) has been developed [3,8]. Within this framework the *syntactical* correctness of the process is always preserved after arbitrary process changes. For example, it is automatically checked by the PMS whether process changes will lead to structural errors, such as deadlock-causing cycles, not properly supplied input parameters, or to inconsistent instance states.

Ensuring syntactical correctness, however, is often not sufficient. In particular, for processes undergoing frequent changes performed by various staff members, mechanisms to ensure the semantic correctness of the processes become necessary. For this purpose, mechanisms to integrate semantic domain knowledge into adaptive PMS are required. In this paper, we introduce a framework for supporting semantic knowledge integration and semantic process verification in the context of changes at process instance and at process template level as well as for their interplay.

1.1. Problem description and challenges

In many application domains, it is not possible to foresee and thus model all possible exceptions and necessary deviations at buildtime. Thus, frequent ad hoc modifications on process instances, for instance adding a special treatment for a particular patient, are the normal case in these domains. It is, therefore, an important requirement to support ad hoc changes in an user-friendly way such that even non-expert users (e.g., a nurse at a hospital workstation) are able to perform ad hoc process instance deviations if necessary. However, performing ad hoc modifications of process instances can also be a source of semantic errors. Consider, for example, process instance I reflecting the treatment process for patient Smith as depicted in Fig. 1. Assume that, due to a suddenly arising headache, drug Aspirin is administered to patient Smith. This is achieved by inserting task `Administer Aspirin` into instance I in an ad hoc manner by, for example, a nurse at her workstation. However, in this treatment process, the drug Marcumar, which is not compatible to Aspirin, is already administered some tasks ahead (*semantic conflict*). Even if the process change is syntactically correct, it is not semantically.

In particular, if a process instance is often modified in an ad hoc manner (e.g., `Administer Marcumar` was previously inserted as an ad hoc modification) or if changes at process template level and at process instance level are merged (i.e., when propagating process template changes to individually modified process instances), it is likely that such semantic conflicts occur and that they remain undetected even by process experts. This leads to a contradictory situation: On the one hand, a lot of time and effort is spent on modeling structurally and semantically correct process templates at buildtime. On the other hand, however, due to performing semantically conflicting process changes, semantic errors creep in at runtime possibly affecting the success of the process.

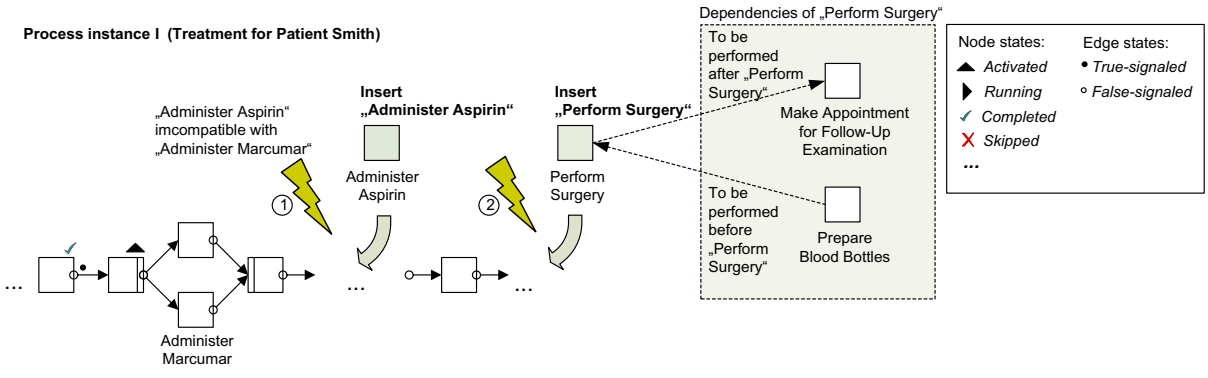

Fig. 1. Semantic conflicts after process changes due to drug incompatibility and dependencies between tasks.

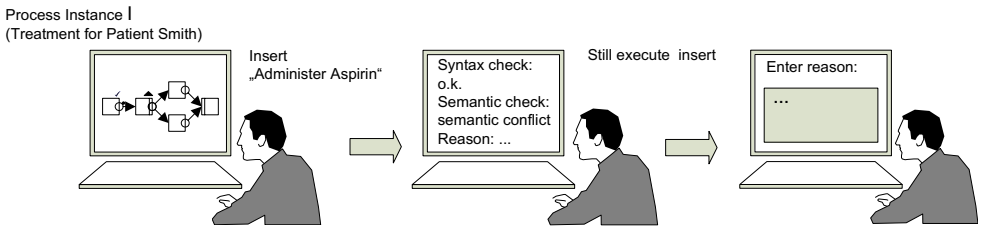

Fig. 2. Interaction scenario when a semantic conflict occurs.

If the PMS was aware of the incompatibility of the two tasks in the above mentioned example (cf. Fig. 1), it could prevent arising semantic conflicts by, for example, warn the user accordingly. In Fig. 2, the user (a doctor with the necessary authorization) still administers Aspirin despite the semantic conflict, but he has to document the reason for overriding the semantic constraint. Generally, documentation and traceability of violations of guidelines and best practices are highly relevant for many application domains (e.g., the medical or the automotive domain) since it becomes possible to trace back semantic conflicts which might cause, e.g., failures in the production.

In order to ensure the semantic correctness of processes – even across various changes – adequate mechanisms are required. Domain knowledge – e.g., knowledge about incompatible tasks – needs to be integrated within the process change framework in order to enable adaptive PMS to be *semantic-aware*. In this context, many challenging questions arise:

- How to formalize and integrate domain knowledge (e.g., business rules [9], medical guidelines [10,11], etc.) within an adaptive PMS?
- How to define a notion of semantic correctness for process templates and process instances and support its efficient verification?
- How to determine whether changes on process templates, process instances, and the propagation of template changes to running process instances are semantically correct?
- How to integrate semantic verification into the interaction scenarios of an adaptive PMS?

Integrating domain knowledge within adaptive PMS and ensuring semantic correctness of running processes at any time will mark a milestone in the practical application of such systems.

1.2. Contribution

In this paper, we present a framework for integrating domain knowledge into adaptive PMS and for performing semantic process verification in an efficient way. First of all, we provide a formalization for semantic constraints imposed on business processes. In particular, we introduce two fundamental kinds of semantic constraints (mutual exclusion constraints and dependency constraints) which serve as a basis for the following considerations. Both kinds of constraints have been identified as highly relevant for different application domains (i.e., the medical or the insurance domain). However, this basic set of semantic constraints can be easily extended as we will show in future work. Based on the notion of semantic constraints, a general criterion for the semantic correctness of business processes (independent from the underlying process meta model) is provided. Most of these results have been presented in [12]. In this paper, we refine the semantic correctness criterion in order to account for state-related differences between process templates and process instances. Furthermore, we extend our fundamental work by showing how semantic correctness of processes can be verified, ranging from verifying semantic correctness for process templates at buildtime to verifying semantic correctness of process changes. For this, we exploit the semantics of the applied change operations, for example when applying single change operations (e.g., ad hoc changes of process instances), or when applying concurrent changes (e.g., propagating process template changes to biased process instances). Afterwards, we discuss different possibilities to realize verification of semantic process correctness in an efficient manner. One way based on exploiting certain process meta model properties is discussed in more detail. Finally, we show how the semantic constraints can be organized within a domain repository.

In Section 2, we present considerations on integrating semantics in adaptive PMS and motivate our approach. In Section 3, a framework for the definition of semantic constraints and the notion of semantic correctness are introduced. In Section 4, we show how the semantic correctness of process templates and the semantic correctness of process changes can be verified. The migration of running instances to a new template is dealt with in Section 5. In Section 6, we show the application of the correctness criterion when, for example, a block-structured process model is used. In Section 7, considerations on the integration of semantic verification in adaptive PMS as well as a framework for organizing semantic constraints are presented. Related work is discussed in Section 8. Finally, Section 9 concludes with a summary and an outlook on future research.

2. On integrating semantics in adaptive PMS

As motivated in Section 1, it is desirable to integrate (semantic) domain knowledge into the PMS in order to enable semantic-aware process management technology. Basically, it is possible to integrate even very complex domain knowledge into adaptive PMS. By connecting the PMS with a knowledge-based system or an expert system (e.g. [13]), for instance, domain knowledge maintained in the external system can be used to avoid semantic conflicts. However, two aspects influence the possibilities of integrating domain knowledge into adaptive PMS. First, it is an important question how and by whom the knowledge base is maintained. The more domain knowledge, and in particular the more complex the knowledge, the greater is the effort to keep the knowledge base up-to-date. Thus, there is a risk that the knowledge, according to which the semantic checks are performed, is outdated. In fact, this might be even more dangerous than not performing semantic checks at all. Users might rely on the semantic checks to ensure the semantic correctness of the process not knowing that the knowledge base is outdated. As a consequence, it seems reasonable to only integrate such domain knowledge which is really important and which will really be kept up-to-date. Second, the goal of integrating domain knowledge is to enable the PMS to also perform correctness checks at the semantic level. However, the effort to perform these semantic checks must not lead to a bottleneck, especially when changes on process templates are propagated to a multitude of running and possibly ad hoc modified instances (for some application domains, e.g., large hospitals, several thousands of instances may be active at the same time).

The two aspects mentioned above should be kept in mind when thinking about how to integrate domain knowledge into adaptive PMS. Therefore, we introduce two fundamental kinds of semantic constraints which can be imposed on processes: mutual exclusion constraints and dependency constraints. The reason is that these kinds of constraints are very common in practice as we know from preliminary case studies (e.g., [14]). Exclusion and dependency constraints refer to tasks and impose certain conditions on how these tasks can be used in the process. By enabling the PMS to be aware of these constraints, many semantic errors, for example the ones depicted in Fig. 1, can be avoided. Furthermore, the introduced kinds of constraints are still manageable regarding the effort for maintenance and semantic verification. We take these two kinds of constraints as a starting basis in order to find out, how semantic constraints and semantic verification can be incorporated into the mechanisms of an adaptive PMS. In future work, we want to extend our approach step by step in order to further investigate on the right balance between expressiveness of constraints, effort for maintenance, and analyzability.

3. Semantic correctness for business processes

In this paper, we introduce two fundamental kinds of semantic constraints: mutual exclusion constraints and dependency constraints. Mutual exclusion constraints express that two tasks are not compatible and should not be executed together (e.g., administering two incompatible drugs). Mutual exclusion constraints are symmetric. Dependency constraints express that a task is dependent of another task, i.e., these tasks have to occur together in the process. In Fig. 1, for instance, task `Perform Surgery` is added to the process. However, in the treatment process the task `Prepare Blood Bottles` needs to be performed before and `Make Appointment for Follow-Up Examination` needs to be performed after `Perform Surgery`. These semantic dependencies of `Perform Surgery` cause a semantic conflict, when only `Perform Surgery` is inserted into the process.

At buildtime, a set of semantic constraints can be assigned to a process template which the template itself as well as the process instances running according to this template have to comply with. Basically, it is possible to extend or restrict the set of inherited semantic constraints for certain process instances. However, an extended or restricted constraint set at instance level has to be considered when checking semantic correctness accordingly. In the remainder of this paper we assume that process instances inherit the set of semantic constraints from their referenced process template. Furthermore, we assume the uniqueness of tasks in a process (i.e., each task may occur only once in a business process). In future work, we will extend our considerations to deal with other kinds of processes as well. In the following definition the structure of a *semantic constraint* is determined.

Definition 1 (*Semantic constraint*). Let $\mathscr{A}$ be a set of tasks.[1] A semantic constraint $c = (type, source, target, position, userDefined)$ whereas

- $type \in \{Exclusion, Dependency\}$,
- $source, target \in \mathscr{A}$, $source \neq target$,
- $position \in \{pre, post, notSpecified\}$,
- *userDefined* is a user-defined parameter.

The parameter *type* denotes whether the semantic constraint is a mutual exclusion constraint or a dependency constraint. The second parameter *source* denotes the source task the constraint refers to while *target* denotes the target task related to the source task. Parameter *position* specifies in which order the source and target task are to be related to each other within the process (e.g., the surgery depends on the preparation of blood bottles and the bottles have to be prepared before the surgery (*pre*)). According to Definition 1, there are six possible constraint types. The last parameter *userDefined* can be used for several purposes, e.g., for additionally describing the constraint or to indicate the importance of the constraint. In the latter case we can express if a constraint is merely a recommendation or if its violation leads to severe problems in the sequel. By exploiting such information the PMS is able to create an appropriate feedback for the user in case of violations. As an example, the constraint mentioned above would look like this:

(`Dependency`, `Perform surgery`, `Prepare blood bottles`, `pre`, `Blood bottles need to be prepared for the patient and stored in the surgery room before the surgery can take place`)

Based on the notion of semantic constraints a general criterion for semantic correctness is defined in the following. Basically, semantic correctness needs information on how tasks can be used within in a process and in which ordering relations they occur. All this information is captured within so called *execution traces*. Therefore, we take execution traces as a basis for our formal criterion for semantic correctness. In addition, defining the semantics of the constraints based on execution traces allows for a meta model independent understanding of constraints. In Definition 2, we define the notion of execution traces and some useful functions over them.

Definition 2 (*Execution trace*). Let $\mathscr{A}$ be a set of tasks which can be used to specify a process template S. An execution trace $\sigma := \langle e_1, \ldots, e_k \rangle$ over S contains events $e_i = \mathrm{End(t)}$[2], $t \in \mathscr{A}$. The set of all possible execution traces over S is denoted as Q_S.

A process instance I is defined as a tuple $I := (S, \bar{\sigma})$ where S denotes the process template which captures the structure of I[3] and execution trace $\bar{\sigma}$ captures the current execution state of I. Then Q_I denotes the set of all possible execution traces over process instance I with $Q_I := \{\sigma \in Q_S | \exists\ \tilde{\sigma}$ with $\sigma = \bar{\sigma}\tilde{\sigma}\}$.

[1] Within the ADEPT framework, for example, $\mathscr{A}$ refers to the task repository containing all relevant tasks in the context of a certain process type T.

[2] We abstract from start events in the traces.

[3] Since a process instance I can be individually modified, the structure of I does not necessarily correspond to the structure of the template I was started on.

Useful functions over execution traces are:

- Function *traceTasks*(σ) returns the set of all tasks constituting the execution trace σ. Formally: *traceTasks*: $Q_{[S|I]} \mapsto 2^{\mathscr{A}}$ with $traceTasks(\langle e_1, \ldots, e_k \rangle) := \{t \in \mathscr{A} | \exists e_i \text{ with } e_i = End(t) \text{ where } i = 1, \ldots, k\}$.
- Function *possibleTasks*($[S|I]$) returns all tasks which are executable for a process template S or a process instance I. Formally: $possibleTasks([S|I]) := \bigcup_{i=1}^{n} traceTasks(\sigma_i)$ for $Q_{[S|I]} = \{\sigma_1, \ldots, \sigma_n\}$.
- Function *traceSucc*(t, σ) returns the set of all direct and indirect successors of the task t in σ. Formally: *traceSucc*: $\mathscr{A} \times Q_{[S|I]} \mapsto 2^{\mathscr{A}}$ with $traceSucc(t, \langle e_1, \ldots, e_k \rangle) := \{t' \in \mathscr{A} | \exists e_i, e_j \text{ with } e_i = End(t') \wedge e_j = End(t) \wedge j < i, \text{ where } i,j = 1, \ldots, k\}$.
- Function *tracePred*(t, σ) returns the set of all direct and indirect predecessors of the task t in σ. Formally: *tracePred*: $\mathscr{A} \times Q_{[S|I]} \mapsto 2^{\mathscr{A}}$ with $tracePred(t, \langle e_1, \ldots, e_k \rangle) := \{t' \in \mathscr{A} | \exists e_i, e_j \text{ with } e_i = End(t') \wedge e_j = End(t) \wedge j > i, \text{ where } i,j = 1, \ldots, k\}$.

In Fig. 3, the execution traces of example process templates and process instances are depicted. For process template S_1, two execution traces can be generated. This is because, for all instances of S_1, either `B` or `C` will be executed (due to the XOR-split) resulting in two different kinds of instances of S_1. Based on the current execution trace of I_1 (which is empty) the same execution traces as over S_1 can be generated. In contrast, the set of possible execution traces of instance I_2 differs from the set of possible execution traces over S_1, due to the current execution trace of I_2. In I_2, task `B` has been executed whereas task `C` has been skipped. Hence, `B` necessarily occurs in all possible execution traces over I_2 whereas `C` does not occur in any of them.

Based on the of notion execution traces, we can define a trace-based satisfaction criterion for semantic constraints which we first want to motivate by means of some examples. Consider again the process template S_1 in Fig. 3. Constraint `c1` is violated when `C` and `E` are executed together, which is the case with trace `A C D E F H`. Constraint `c2` is violated if `C` is executed but not `G`. This is the case with both the possible traces over S_1. Constraint `c3` is violated if `H` is executed without `B` being executed previously. Since `B` is situated in an XOR-branch, this situation occurs if `B` is skipped and `C` is executed instead (reflected by trace `A C D E F H`). In order to avoid all possible semantic conflicts, the semantic correctness has to be ensured for all possible executions (reflected by all possible execution traces) of a process template or a process instance. We refer to a semantic constraint as being violated over a process if there are possible execution traces of the process violating the constraint (cf. Definition 3). According to the set of possible execution traces over S_1, the constraints `c1`, `c2`, and `c3` are not satisfied over S_1 (i.e., violated). For process instance I_1, the very same applies. For process instance I_2, however, none of the semantic constraints are violated according to the only possible execution trace over I_2. Since task `C` is skipped, `c1` as well as `c2` can never be violated. Also constraint `c3` cannot be violated since `B` is already executed.

These examples show that constraints might be satisfied over a process instance, but not over the corresponding template. This is because, generally, the traces which can be generated over a process instance at a certain execution state form a subset of the traces, which can be generated over its process template. The

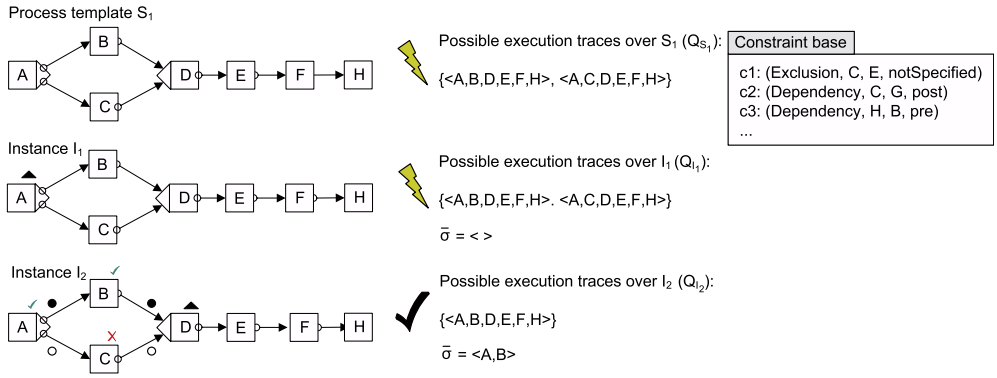

Fig. 3. Impact of execution states on the satisfaction of semantic constraints.

examples also show that the satisfaction of a constraint over a process template is constant over time (in case no conflicting changes are made (cf. Section 4)), whereas the satisfaction of a constraint over a process instance may vary due to its ongoing execution. All these aspects are accounted for in the following definition of a satisfaction criterion for semantic constraints.

Definition 3 (Satisfaction criterion for semantic constraints). Let $\mathscr{A}$ be a set of tasks which can be used to specify process template S and let I be a process instance derived from S. Let further Q_S (Q_I) be the set of possible execution traces over S (I) and let a_1, $a_2 \in \mathscr{A}$ be two tasks, $a_1 \neq a_2$. Then semantic constraint $c = (type, source, target, position, userDefined)$ with $source = a_1$ and $target = a_2$ is `satisfied` over S (I) – formally: $satisfied(c, [S|I]) = \texttt{True}$ – iff one of the following conditions holds:

- $type \in \{Exclusion, Dependency\}$ and $a_1 \notin possibleTasks([S|I])$
- $type = Exclusion$, $position = pre$ and $\forall$ execution traces $\sigma \in Q_{[S|I]}$: $a_1 \in traceTasks(\sigma) \Rightarrow a_2 \notin tracePred(a_1, \sigma)$
- $type = Exclusion$, $position = post$ and $\forall$ execution traces $\sigma \in Q_{[S|I]}$: $a_1 \in traceTasks(\sigma) \Rightarrow a_2 \notin traceSucc(a_1, \sigma)$
- $type = Exclusion$, $position = notSpecified$ and $\forall$ execution traces $\sigma \in Q_{[S|I]}$: $a_1 \in traceTasks(\sigma) \Rightarrow a_2 \notin traceSucc(a_1, \sigma)$ and $a_2 \notin tracePred(a_1, \sigma)$
- $type = Dependency$, $position = pre$ and $\forall$ execution traces $\sigma \in Q_{[S|I]}$: $a_1 \in traceTasks(\sigma) \Rightarrow a_2 \in tracePred(a_1, \sigma)$
- $type = Dependency$, $position = post$ and $\forall$ execution traces $\sigma \in Q_{[S|I]}$: $a_1 \in traceTasks(\sigma) \Rightarrow a_2 \in traceSucc(a_1, \sigma)$
- $type = Dependency$, $position = notSpecified$ and $\forall$ execution traces $\sigma \in Q_{[S|I]}$: $a_1 \in traceTasks(\sigma) \Rightarrow (a_2 \in tracePred(a_1, \sigma)$ or $a_2 \in traceSucc(a_1, \sigma))$

Otherwise, c is `violated` over P (I), formally: $satisfied(c, [S|I]) = \texttt{False}$.

Based on the satisfaction criterion for semantic constraints, a semantic correctness criterion for process templates and process instances can be defined.

Definition 4 (*Semantic correctness of process template/instance*). Let S be a process template and $I = (S, \bar{\sigma})$ be a process instance on S. Let further C_S be the set of all semantic constraints imposed on S. Then, S (I) is semantically correct $\leftrightarrow \forall c \in C_S$: $satisfied(c, [S|I]) = \texttt{True}$.

Definitions 1–4 build the formal basis for semantic process verification and semantic change verification. How the verification is conducted (in particular, how to avoid the calculation of all possible execution traces over a process which has exponential complexity) is discussed in the following.

4. Semantic verification of process templates and process instance modifications

Verifying the semantic correctness of a process template (instance) based on Definition 4 might be expensive depending on the number of semantic constraints imposed on the process template (instance) on the one hand and the number of possible execution traces on the other hand. The verification effort, however, can be decreased by restricting the set of semantic constraints to be checked as well as by avoiding a complete calculation and analysis of possible execution traces over the process template (instance). In Section 4.1, we present considerations on how to minimize the amount of constraints to be verified for a process template. In Section 4.2, we show how to identify potentially violated constraints when ad hoc process adaptations are carried out.

4.1. On optimizing semantic verification of process templates

Basically, if a process template S specified over task set $\mathscr{A}$ is built by applying process changes to an "empty" template the PMS might perform a semantic check each time a change operation is applied and check whether the semantic correctness of the template is still preserved after the change or not (cf. Section 4.2). It is, however, also important for an adaptive PMS to support the verification of a completely modeled process

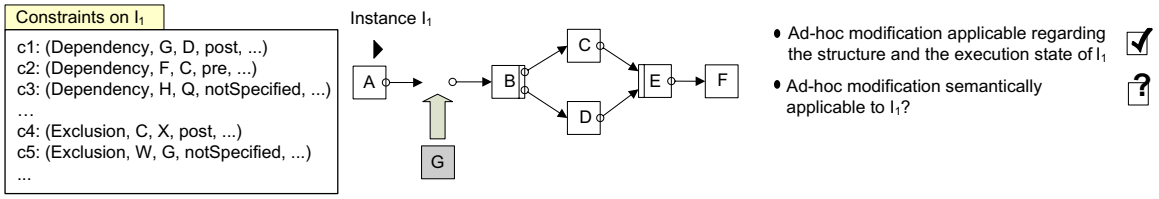

Fig. 4. Minimizing the set of potentially violated constraints for instance changes.

template S. This is, for instance, necessary when S was modeled with a tool not supporting stepwise semantic verification and S is imported into and executed by the PMS. In this case, it is necessary to verify whether the constraints imposed on S are satisfied or not. However, constraints, whose source task is not included in S, are always satisfied over S by definition (cf. Definition 3) and, thus, do not have to be verified. Altogether, the set of constraints to be verified for the template S and its constraint base C_S is given by C_S^v with:

$$C_S^v = \{c = (\ldots, source, target, \ldots) \in C_S | source \in possibleTasks(S)\}$$

4.2. On checking semantic correctness after process instance changes

In our framework, an ad hoc process change is considered *semantically applicable* to a process instance I, if its application still preserves the semantic correctness of I. The naive way of verifying the semantic correctness of an instance I after a change is to verify the complete set of constraints C_I^v obtained by applying the considerations presented in Section 4.1. As we will show, this effort can be reduced by exploiting the semantics of the applied change operations (e.g., which task has been inserted at which position). Thus, depending on which change operation is requested, only a subset of constraints imposed on instance I needs to be verified. Consider for example the process instance I_1 in Fig. 4. In order to verify whether the insertion of `G` is semantically applicable, we can exploit the fact that I_1 was semantically correct before the modification (i.e., all constraints imposed on I_1 are satisfied before the change). Thus, there is no need to reverify each constraint. Instead, only constraints which might be violated by the change operations are relevant. In Fig. 4, constraint `c2`, `c3`, and `c4` do not affect the task to be inserted (`G`). Hence, they cannot be violated by the insertion of `G`.

Table 1 gives an overview of change operations, their effects, and the set of semantic constraints to be verified in order to verify the semantic applicability of the respective change operation. In the following, the results presented in the table are explained in more detail.[4]

When **inserting** a task a into process instance I, all semantic constraints over I having a as source parameter need to be verified since they might be violated. For example, if a is dependent of or is incompatible to another task, this is expressed by the semantic constraints associated with a (i.e., constraints having a as source parameter). For process instance I_1 in Fig. 4, this applies to constraint `c1`. Since mutual exclusion constraints are symmetric, also mutual exclusion constraints with a as target parameter might be violated by the insertion of a. In our example, this applies to constraint `c5`.

All other constraints are sure to be satisfied. Dependency constraints not having a as source parameter cannot be violated by the addition of a (e.g., `c2` and `c3` in Fig. 4). Also exclusion constraints not affecting a cannot be violated either (e.g., `c4` in Fig. 4). The reason is that the process is supposed to be semantically correct before applying the insertion. Hence, all constraints imposed on the process not related to a are supposed to be satisfied over I. These constraints, therefore, can be excluded from satisfaction checks.

Based on information on the process instance, the set of potentially violated exclusion constraints can be further minimized: Only those with source parameter in *possibleTasks*(I) (i.e., source is included in I and is not skipped) with target parameter corresponding to the inserted task a can be violated. That is because all exclusion constraints, whose source parameter are not included in the process or will not come to execution, are satisfied by definition (cf. Definition 3). As a result, in Fig. 4, only `c1` needs to be verified in order to find out whether the insert operation is semantically applicable.

[4] In this paper, we restrict our considerations to the most common change operations: *Insert*, `Delete`, and `Move`.

Table 1
Change operations, their effects, and the set of semantic constraints to be verified

Operation applied to instance I	Effect	Constraints to be verified (C_I^v)
Insert(I,a,pos)	Inserts task a into instance I at position pos	$C_I^v = \{c = (type, source, target, \ldots) \in C_I \mid (source = a) \vee (type = Exclusion \wedge source \in possibleTasks(I) \wedge target = a)\}$
Delete(I,a)	Deletes task a from instance I	$C_I^v = \{c = (type, source, target, \ldots) \in C_I \mid (type = Dependency \wedge source \in possibleTasks(I) \wedge target = a)\}$
Move(I,a,pos)	Moves task a to the position pos in instance I	C_I^v corresponds to the union of the constraint sets to be verified when performing Delete(I, a) and Insert(I, a, pos)

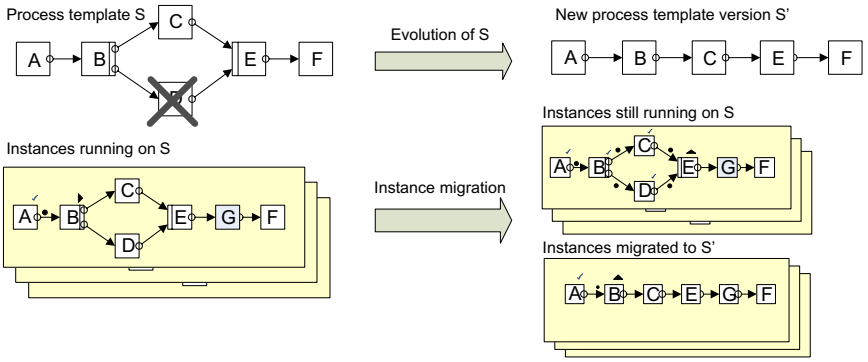

Fig. 5. Propagating process template changes to running process instances.

When **deleting** a task a from process I, all semantic constraints over I with a as source parameter are satisfied by definition. Similar to the insertion of tasks, all constraints for which a occurs as target parameter are potentially interesting for correctness checks. However, mutual exclusion constraints with a as target parameter cannot be violated by the deletion of a. Only dependency constraints with a as target parameter and whose source parameter is included in I and is not skipped (i.e., in $possibleTasks(I)$) might be violated by the deletion operation and, therefore, need to be verified.

Moving a task a from its original position within process I to a new position pos can be understood as being equivalent of deleting a and inserting a at pos afterwards.[5] Consequently, all constraints which might be violated after applying deletion and insertion operations need to be verified.

Generally, these considerations can also be applied in order to verify changes at template level. Note that changes might be applicable to a process instance which are not semantically applicable to the corresponding template. Due to the current execution trace of the instance, semantic constraints might be satisfied over the instance after a process change while not beeing satisfied over the corresponding template after the same process change (cf. Section 3).

5. On checking semantic correctness for process template evolution

In addition to ad hoc changes at the instance level, adaptive PMS must also support modifications of process templates. This becomes necessary, for instance, when changes in laws and policies need to be adopted in the business process (process optimization). In those cases, it is not only necessary to be able to modify the process template but also to propagate template changes to instances already running according to the old template (cf. Fig. 5). As mentioned, for semantically verifying the template change, the considerations made in Section 4.2 can be applied. The question of whether and how running instances can be migrated to the new template version is more challenging.

[5] In conjunction with data flow aspects, moving is not always equivalent to deleting and inserting. However, this assumption can be used to derive statements about possible semantic conflicts here.

In order to be migrated to the new template version, it is important that instances are not only syntactically but also *semantically compliant* to the new template. In current literature, the problem of testing *syntactical compliance* of instances has already been tackled. In [8,15], a change framework for efficient syntactical instance migration is introduced. Using this framework it can be checked, for example, whether migrating an instance to the new template would cause structural (e.g., deadlocks-causing cycles or not correctly supplied input parameters) or state-related inconsistencies. This has been accomplished for arbitrary running process instances. Since process instances can be individually modified via ad hoc changes this is not a trivial task. The problem of testing semantic compliance, to our best knowledge, is novel in the context of adaptive PMS.

In general, a process instance I is semantically compliant with a new process template version S', if the template changes are semantically applicable to I as an ad hoc change. However, trying to apply the template change to each running process instance is not feasible in practice since there might be thousands of instances to be checked. In this paper, we present more efficient mechanisms to find out, whether instances can be migrated to the new template without violating any semantic constraints. For this purpose, first of all, it is determined which instances can be migrated to the new template version in a syntactically correct manner. Then, the semantic checks are only applied to the set of syntactically compliant instances. For a better understanding, some background information on the syntactical instance migration framework is provided in the next section.

5.1. Background information: syntactical instance migration

In the framework presented in [8,15], instances to be checked for syntactical compliance are classified according to the relation between their individual changes and the template changes (cf. Fig. 6). The reason for doing this is that the migration strategy to be applied (i.e., whether the instance is compliant with the template changes and if so which instance adaptions become necessary) is based on the degree of overlap between instance and template changes [8]. Our basic idea is to check whether the classification for syntactical compliance is useful for checking semantic compliance as well. This would be very beneficial since the classification has to be accomplished in any case and could be used for semantic verification at no extra costs.

As shown in Fig. 6, instances can be divided into two main classes: *unbiased* instances and *biased* instances. Unbiased instances are instances, which have not been individually modified so far (cf. Fig. 7). Biased instances have already been individually modified. Depending on the degree of overlap between instance and template changes, they can be further divided into subclasses: instances with *disjoint bias* and instances with *overlapping bias*.

Informally, instance and template changes are disjoint if they affect different areas of the underlying process structure. In Fig. 7, instance I_6 belongs to this class. Instances with *overlapping bias* comprise instances which have changes partly or fully overlapping with the changes of the template. There are three subclasses to this class: *equivalent bias*, *subsumption equivalent bias*, and *partially equivalent bias* (again, this distinction is necessary for finding adequate migration strategies afterwards).

In the following, let ΔI be the instance changes on a process instance I and ΔS be the template changes on process template S.

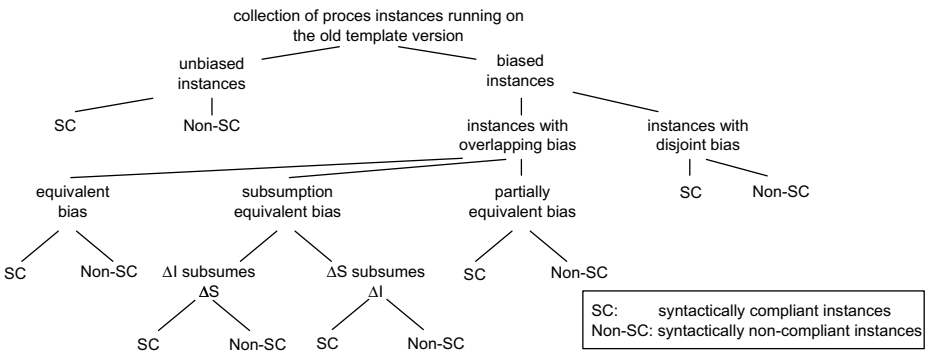

Fig. 6. Classification of instances according to the syntactical migration framework.

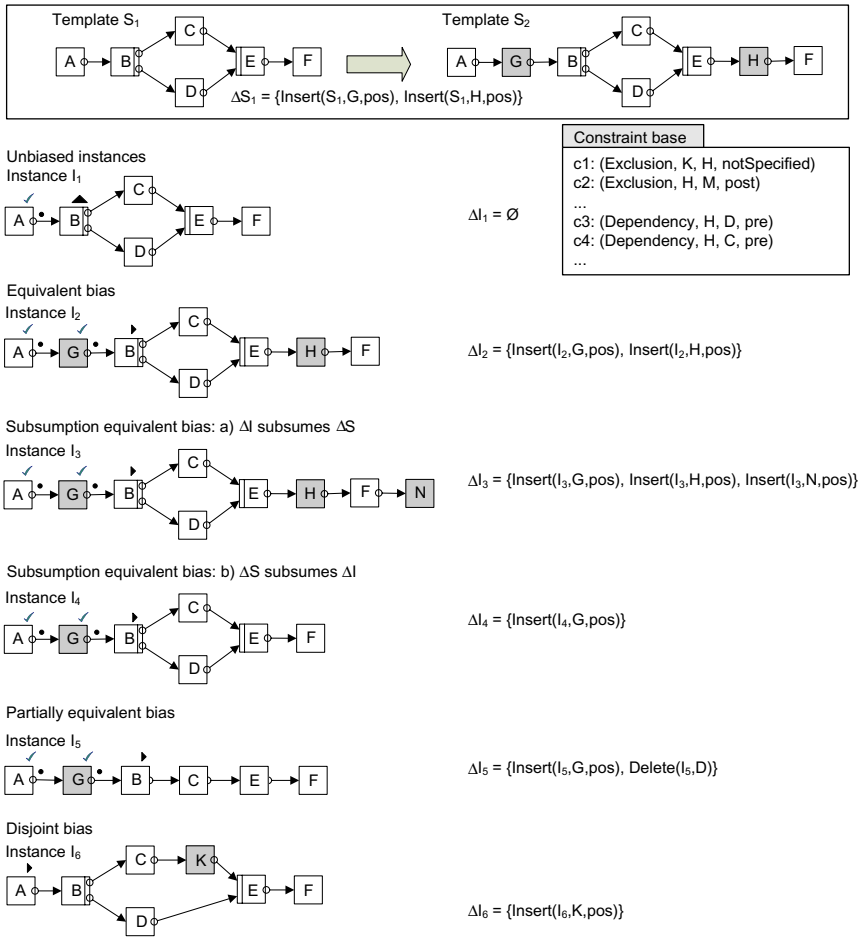

Fig. 7. Process template and instance changes with different degrees of overlap.

Instances with *equivalent bias* have the same modifications as the template (cf. instance I_2 in Fig. 7). Instances with subsumption equivalent bias can be further divided into two classes: ΔI *subsumes* ΔS and ΔS *subsumes* ΔI. For instance I_3 in Fig. 7, for example, the instance changes comprise the template changes plus a delete operation. Thus, I_3 belongs to the class ΔI *subsumes* ΔS whereas instance I_4 belongs to the class ΔS *subsumes* ΔI. Instances with partially equivalent bias have changes in common with the new template version. However, there are also changes on the template which do not correspond to the instance changes and vice versa. In Fig. 7, instance I_5 belongs to this class.

Based on the classification presented in Fig. 6 it can be checked whether the instances are syntactically compliant or syntactically non-compliant with a modified process templated. Checking for semantic compliance of instances which are not syntactically compliant with the new template does not make sense. Therefore, in this paper, we focus on efficiently checking the semantic compliance of syntactically compliant instances. As already mentioned, the classification depicted in Fig. 6 can be used as input for semantic migration tests at no extra costs. In Section 5.3, we show how this classification can be employed in order to reduce semantic verification efforts.

5.2. The basic scenario: semantic migration of unbiased process instances

In case a template change ΔS is semantically correct on S, it will also be semantically correct when being applied to *unbiased* instances. This is because unbiased instances only differ from the template in point of their execution states. Thus, changes semantically applicable on a process template are a subset of changes semantically applicable on the template's unbiased instances (cf. Section 3). As a consequence, unbiased instances are

semantically compliant to the new process template version and can be migrated without any further checks. In Fig. 7, for example, instance I_1 can be migrated to S_2 without any additional semantic checks.

5.3. The advanced scenario: concurrent process changes

Contrary to unbiased instances, applying template changes to *biased* instances may lead to semantic conflicts between process template and process instance changes (*concurrent changes*). Fig. 8 gives an example of how merging the changes by migrating instance I_6 to the new template version S_2 causes a semantic conflict. At the template level, task G and H are inserted resulting in the new template S_2. Instance I_6 was individually modified at runtime by inserting task K. Propagating template changes to I_6, however, causes a semantic conflict. This is because, the insertion of H is not semantically applicable to I_6, since K is not compatible to H. Imagine, for instance, that, at template level, Aspirin is administered while at instance level, Marcumar is administered. Even though the resulting instance would be syntactically correct, I_6 cannot be migrated to S_2 in a semantically correct way.

In Section 5.3.1, we show how the amount of instances to be verified can be further reduced in order to reduce the effort for verifying biased instances. Then, in Section 5.3.2, we present considerations on how to minimize the amount of constraints to be checked depending on the semantics of the change operations made at template and instance level.

5.3.1. Identifying semantically critical instances

As mentioned before, it only makes sense to migrate biased instances, which are syntactically compliant with the new template version. Depending on the ad hoc changes applied to the instance so far, however, it is not necessary to check all syntactically compliant instances for semantic compliance. Based on the classification of the syntactically compliant instances as presented in Section 5.1, we provide different semantic migration strategies in order to reduce semantic verification efforts.

Equivalent bias: Since template and instance changes are equivalent, instances of this class are syntactically identical to the new template version (cf. instance I_2 in Fig. 7). Thus, no semantic conflicts can ever occur when migrating these instances to the new template. As a consequence, it is not necessary to carry out any semantic migration checks for these instances. Instead, they can be migrated without any semantic checks.

Subsumption equivalent bias: This class is further divided into two subclasses:

ΔI subsumes ΔS: Instances of this class can be semantically migrated to a new template version S' without any further semantic checks. This is because all template changes are already included (i.e., anticipated) in the set of changes on the process instance. For instance I_3 in Fig. 9, for example, task N is inserted additionally to the changes made at template level. The changes I has in common with S' cannot lead to a semantic conflict in case of a migration (this is comparable to instances in the class equivalent changes). Furthermore, I is supposed to be semantically correct due to semantic verification of the ad hoc changes on I. Hence, also the additional changes on I (e.g., the insertion of N into I_3 in Fig. 9), cannot semantically conflict with the template changes (i.e., the remaining changes on I). Thus, instances of this class can also be migrated without any semantic checks (cf. Fig. 9).

ΔS subsumes ΔI: Since ΔS subsumes ΔI, migrating I to S' means to propagate those template changes to I, which are not included in ΔI. For I_4 in Fig. 9, this would mean to propagate the insertion of H to I_4. Since

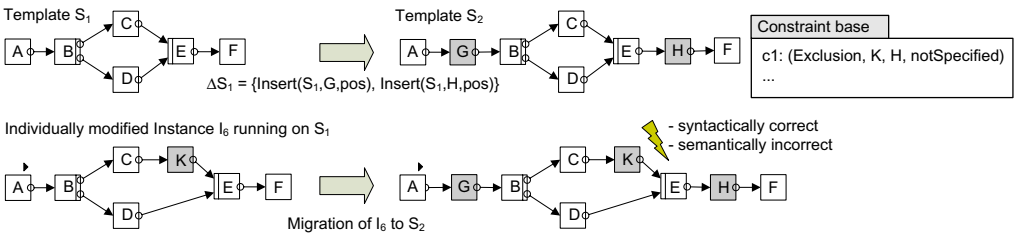

Fig. 8. Semantic conflict due to concurrent changes at template and instance level.

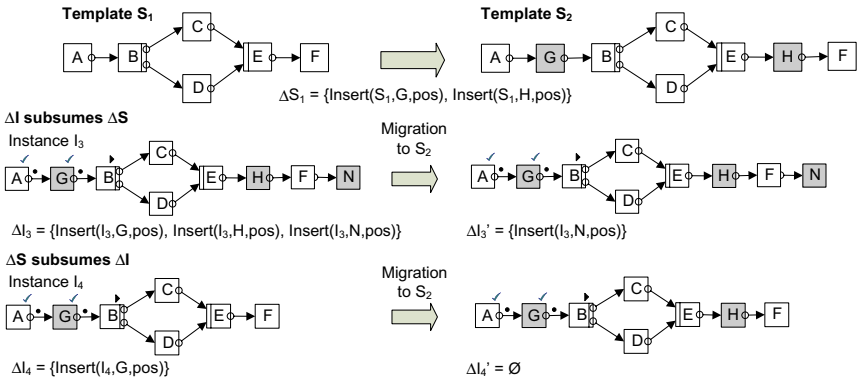

Fig. 9. Migration of instances from the class subsumption equivalent changes.

all instance changes ΔI are already anticipated by ΔS, a propagation cannot lead to a semantic conflict between the change operations. After a migration, I is structurally identical to S' (cf. Fig. 9). As a consequence, instances of this class can also be migrated without any semantic checks.

Disjoint bias $\Delta S \cap \Delta I = \emptyset$: Contrary to the instances considered so far, migrating instances of this class to a new template version S' may cause semantic conflicts. Since the instance changes and the template changes are disjoint, semantic conflicts may be caused by the interplay of these changes (e.g., insertion of two incompatible tasks at template and at instance level). Consequently, it has to be checked whether instances of this class are semantically compliant to S'. This is true, if ΔS is semantically applicable to I as an ad hoc change (cf. Section 5.3.2). **Partially equivalent bias $\Delta S \cap \Delta I \neq \emptyset$**: Similar to instances with disjoint changes, instances whose changes are partially equivalent to the template changes may be semantically incorrect if migrated to the new template version. This is because the changes made on S but not on I (i.e., $\Delta S \backslash \Delta I$) may be conflicting with changes made on I but not on S (i.e., $\Delta I \backslash \Delta S$) when being propagated to I. In Fig. 7, this applies to the insertion of `H` into I_5. This would cause a semantic conflict, since `H` needs `D` to be executed previously (constraint `c3`) but `D` has already been deleted from I_5. Thus, instances with partially equivalent bias have to be verified. However, it is not necessary to check whether ΔS is semantically applicable to I as a whole (as for instances with disjoint changes). Since changes which ΔS and ΔI have in common are semantically applicable on I, these changes are compatible to other ad hoc changes made on I. In our example, the insertion of `G` has already been applied to I_5. Thus, this change operation cannot conflict with other changes on I_5. Consequently, instances of this class are semantically compliant with S', if $\Delta S \backslash \Delta I$ is semantically applicable to I as an ad hoc change (e.g., deletion of `D` for instance I_5 in Fig. 7). For the calculation of $\Delta S \backslash \Delta I$, we refer to [8].

Using the considerations presented above unnecessary semantic migration checks can be avoided. As summed up by Table 2, only instances of the classes *partially equivalent bias* and *disjoint bias* have to be checked for semantic compliance. For instances of the class *partially equivalent bias*, the verification effort can be reduced by minimizing the set of changes to be checked.

In the next section, we show how to minimize the set of constraints to be verified for instances of both these classes.

Table 2
Instance classes and changes to be semantically verified

Instance class	Changes to be semantically verified
Unbiased	$\emptyset$
Equivalent bias	$\emptyset$
ΔI subsumes ΔS	$\emptyset$
ΔS subsumes ΔI	$\emptyset$
Partially equivalent bias	ΔS
Disjoint bias	$\Delta S \backslash \Delta I$

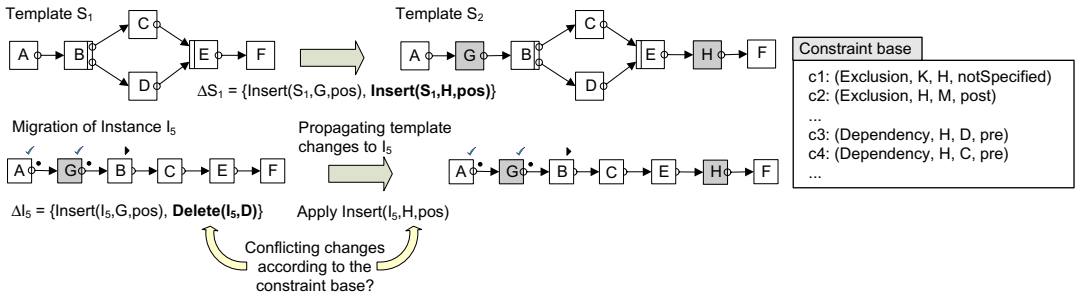

Fig. 10. Identifying potentially violated constraints for propagating template changes to instances.

5.3.2. Identifying potentially violated constraints

In this section, we present how the checks for semantic compliance necessary for instances with disjoint or partially equivalent changes can be optimized. In particular, an instance I of the class *disjoint bias* (*partially equivalent bias*) is semantically compliant if the templates changes ΔS ($\Delta S \backslash \Delta I$) are semantically applicable to I as an ad hoc change, respectively. For applying this compliance criterion, the considerations for ad hoc changes in order to identify potentially violated constraints presented in Section 4.2 can be employed. Take, for example, instance I_5 from Fig. 10. Instance I_5 belongs to the class *partially equivalent bias*. As a result, it is semantically compliant with S_2, if the insertion of `H` is semantically applicable to I_5. Applying the considerations for identifying critical constraints for ad hoc changes from Section 4.2, the constraints `c2`, `c3`, and `c4` are identified as potentially violated. However, different from verifying the applicability of ad hoc changes, more semantic information is available when verifying the semantic compliance of instances with a new template version. It is known that the template change is semantically applicable to the template and the instance change is semantically applicable to the instance. This information can be exploited to further narrow down the set of constraints potentially violated by only considering constraints which might be violated by the *interplay*, i.e. the merge, of both the template and instance changes.

In Table 3, for each combination of change operations at instance and template level the corresponding constraint types which might be violated by the interplay of both the changes are listed. If, for example, task a is inserted into instance I while task b is inserted into template S resulting in S', only exclusion constraints affecting both of the inserted tasks can be violated when the I is migrated to S'.

In Fig. 10, only constraints need to be verified which can be violated by the application of both the changes `Insert(I5,H,pos)` and `Delete(I5,D)`. According to Table 3, only a constraint of type `(Dependency,H,D,2...)` can possibly be violated. Only constraint `c3` corresponds to this type and, therefore, has to be verified. In our example, `c3` would be violated by the change propagation. Hence, I_5 cannot be semantically migrated to S_2.

In Section 5, we narrowed down the set of instances to be semantically verified based on the degree of overlap between instance and template changes. Furthermore, we showed how to minimize the set of constraints to be verified by exploiting the semantics of the change operations. In the next section we consider how to optimize the verification of these constraints.

Table 3
Potentially violated constraint types depending on the applied change operations at template and instance level

	Insert(I,a,pos)	Delete(I,a)	Move(I,a,pos)
Insert(S,b,pos)	(Exclusion,$b,a,\ldots$) (Exclusion,$b,a,\ldots$)	(Dependency,$b,a,\ldots$)	(Exclusion,$b,a,\ldots$) (Exclusion,$a,b,\ldots$) (Dependency,$b,a,\ldots$)
Delete(S,b)	(Exclusion,$a,b,\ldots$)		(Dependency,$a,b,\ldots$)
Move(S,b,pos)	(Exclusion,$b,a,\ldots$) (Exclusion,$a,b,\ldots$) (Dependency,$a,b,\ldots$)	(Dependency,$b,a,\ldots$)	(Dependency,$b,a,\ldots$) (Dependency,$a,b,\ldots$) (Exclusion,$b,a,\ldots$) (Exclusion,$a,b,\ldots$)

6. On optimizing verification strategies based on process meta model properties

The trace-based satisfaction criterion for semantic constraints introduced in Section 3 is generic and can be applied to any process meta model (e.g., Petri Nets [1] or BPEL4WS Nets [16]). For verifying the criterion, reachability analysis can be applied (i.e., by calculating all possible execution traces and checking them for certain order relations between tasks according to the semantic constraints) which might be very costly. Therefore, we want to investigate different methods to verify the satisfaction of constraints which are less expensive. In this paper, we present an approach which makes use of certain properties of the underlying process meta model, namely block-structuring (e.g., WSM Nets [3]). However, we intend to also develop model-independent methods in future work.

6.1. Background information

This section summarizes background information on WSM Nets [17,15] as process description formalism in order to present an optimized verification method for semantic correctness.

A *process template* is represented by a WSM Net which defines the *process tasks* as well as the *control* and *data flow* between them. When using WSM Nets the control flow template can be represented by attributed, serial–parallel graphs. In order to synchronize tasks from parallel paths additional links can be used [18]. In this paper we abstract from cyclic structures within the process meta model in order to provide a fundament for an optimized semantic correctness verification. Further on, a WSM Net comprises a set of *data elements* and a set of *data edges*. A data edge links a task with a data element and either represent a read access of this task or a write access. The total set of data edges constitutes the data flow template.

Definition 5 (*WSM Net*). A tuple $S = (N, D, NT, CtrlEdges, SyncEdges, DataEdges, BC)$ is called a WSM Net, if the following holds:

- N is a set of process tasks and D a set of process data elements
- $NT: N \mapsto \{\texttt{StartFlow}, \texttt{EndFlow}, \texttt{Task}, \texttt{AndSplit}, \texttt{AndJoin}, \texttt{XOrSplit}, \texttt{XOrJoin}, \texttt{StartLoop}, \texttt{EndLoop}\}$ NT assigns to each node of the process template a respective node type.
- $CtrlEdges \subset N \times N$ is a precedence relation definining the valid order of tasks (notation: $n_{src} \rightarrow n_{dst} \equiv (n_{src}, n_{dst}) \in CtrlEdges$)
- $SyncEdges \subset N \times N$ is a precedence relation between tasks of parallel branches
- $DataEdges \subseteq N \times D \times \{\text{read, write}\}$ is a set of read/write data links between tasks and data elements
- $BC: N \mapsto Conds(D)$ where $Conds(D)$ denotes the set of all valid transition conditions on data elements from D. $BC(n)$ is undefined for nodes n with $NT(n) \neq \texttt{XOrSplit}$.

Which constraints have to hold such that a process template S is well-structured is summarized in [18,8] (e.g., absence of deadlock-causing cycles and correctly supplied input parameters). In the context of this paper, the block-structuring property is important, i.e., for all tasks of node type `AndSplit` (`XOrSplit`) there is a unique task of node type `AndJoin` (`XOrJoin`) and blocks (sequences as well as parallel and alternatives branchings) can be nested but must not overlap.

6.2. On exploiting process meta model properties

For each type of semantic constraint, we derived structural and state-related conditions on WSM Nets which ensure the trace-based constraint satisfaction criterion presented in Section 3. Using these meta model specific conditions the satisfaction of semantic constraints and, thus, the semantic correctness of a process can be verified in an optimized way. Due to space restrictions, however, we abstain from presenting all conditions in this paper. Instead, we show how meta model specific conditions can be derived for one example constraint type. First, we focus on process templates. Later, we extend our considerations to process instances. Consider the following semantic constraint over the treatment process from Section 1:

$c_1 : (\texttt{Exclusion}, \texttt{AdministerAspirin}, \texttt{AdministerMarcumar}, \texttt{notSpecified}, \ldots)$

In general, an exclusion constraint is always satisfied if either the source or the target task is not included in the set of tasks constituting the template. Due to the absence of at least one of the two tasks, no execution of the template is possible which violates the constraint. The only way target and source task can both occur in a process template without violating c_1 is on different branches of an XOr-block. In this case, the tasks can only be executed exclusively which ensures the semantic correctness of all of the template's possible executions. In Fig. 11, c_1 is not satisfied over process template S_1, since there are possible execution traces over S_1 for which both of the incompatible tasks are executed.

From this example we conclude the following conditions for the satisfaction of exclusion constraints of type $c_{ex} = (Exclusion, source, target, notSpecified, \ldots)$ over block-structured process templates:

A semantic constraint c_{ex} imposed on a process template S represented by a WSM Net $(N, D, NT, \ldots)$ is satisfied if and only if one of the following conditions holds.

- $source \notin N$ (i.e. *source* does not occur in S);
- $target \notin N$ (i.e. *target* does not occur in S);
- $MinBlock(S, source, target) = (blockStart, blockEnd)$ with $blockStart = \texttt{XOrSplit}$, where *MinBlock* denotes a function which returns the start and end task of the minimal block surrounding the given tasks [19].

In [12], we presented the structural satisfaction conditions for dependency constraints and proved that they ensure the trace-based satisfaction criterion.

In general, the structural satisfaction criterion for process templates can also be applied to process instances. However, due to the current execution state of the process instances, this is too restrictive. The reason is that there are situations in which the semantic constraint is satisfied over a process instance (due to its execution states) but not over the corresponding process template (cf. Section 3). Take for example instance I_1 in Fig. 11. Constraint $\texttt{c}_1$ is satisfied over I_1 but not over S. This is because for I_1, task `Administer Aspirin` has already been skipped. Thus, for all possible execution traces of I_1, `Administer Aspirin` and `Administer Marcumar` will not occur together, independently from whether `Administer Marcumar` will be executed later on or not. Since exclusion constraints are symmetric, the same applies if `Administer Marcumar` is skipped. Thus, for process instances the structural satisfaction criterion has to be complemented by additional state-related satisfaction criterion in order to account for these situations.

A semantic constraint $c_{ex} = (Exclusion, source, target, notSpecified, \ldots)$ over a process instance $I = (S, \bar{\sigma})$ with S capturing the structure of I is satisfied if and only if one of the following conditions holds:

- c_{ex} is satisfied over S (structural condition)
- $ExState(source) = \texttt{SKIPPED}$, where *ExState* denotes a function which returns the execution state marking for the corresponding node
- $ExState(target) = \texttt{SKIPPED}$

The structural/state-related satisfaction conditions on block-structured process meta models derived can be verified very efficiently. In particular, special properties of the meta models, for instance references to the corresponding split/join node of a block and topological sorting [19], can be exploited by the PMS in order to find out whether the respective constraint is satisfied or not.

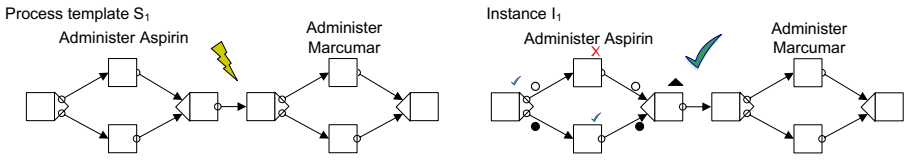

Fig. 11. Satisfaction of exclusion constraints over process templates and instances.

7. Architectural considerations

In this section, we present our ideas on integrating semantic verification into the verification mechanisms of an adaptive PMS. In addition, we present our ideas on organizing semantic constraints in order to facilitate constraint reuse.

7.1. Architectural integration

As pointed out previously, semantic verification is coupled with structural verification. In Fig. 12, an example interaction for verifying an ad hoc change is depicted. At first, an ad hoc change is initiated be the user. Since ad hoc changes are often performed by end-users, the workflow client needs to provide adequate features. The change request is then processed to the process engine. Internally, a module is required to coordinate the verification process. First, the structural verification is triggered for the change request. In case the change is not syntactically applicable, the change can be denied immediately. In our example, the change does not cause any syntactical errors. Thus, semantic verification is triggered. Depending on the results, the coordinator can allow or deny the change and provide an appropriate feedback.

7.2. Organization of constraints

In our approach, a set of semantic constraints is assigned to a process. However, several processes may share constraints. In this section, we present a framework for organizing semantic constraints for their easy reuse.

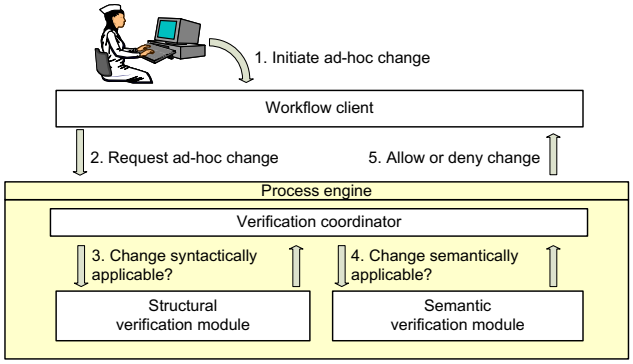

Fig. 12. Interaction scenario for verifying ad hoc changes.

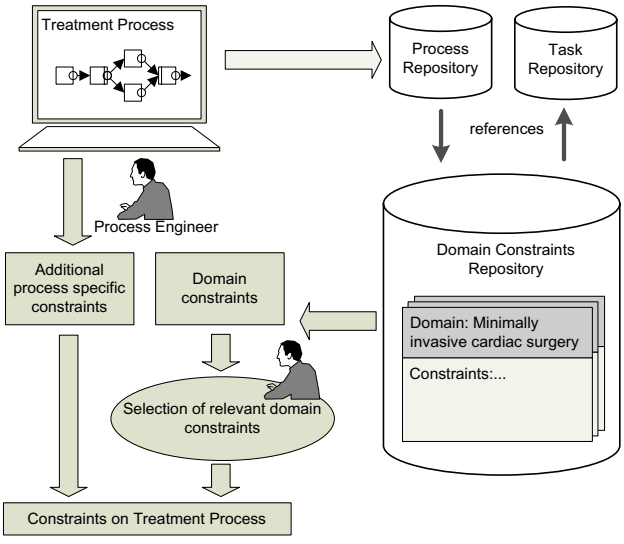

Fig. 13. Organization of constraints in a domain constraints repository.

The three main components of the framework are the *domain repository*, the *process repository*, and the *task repository* (cf. Fig. 13). Semantic constraints are organized in the domain repository. In particular, constraints are assigned to domains (e.g., domain *Minimally invasive cardiac surgery*). Thus, a domain contains a set of constraints that are typical of this domain. The constraints presented in this paper refer to tasks which are organized in a task repository. For future work, we also plan to introduce constraints that refer to other abstraction levels, for instance abstraction levels in the task repository. Process templates are organized in a process repository. Each process template is assigned a domain of the domain repository. Thus, it is possible to assign a default set of domain constraints to a process. However, processes that are assigned to the same domain might still have different semantic constraints that are not captured in the domain. Therefore, for each process template, the process designer can specify additional semantic constraints for the process or leave out some unnecessary domain constraints. Thus, it is possible to tailor an individual set of constraints for each process template.

8. Related work

The discussion ranges from approaches which we consider as being most related to ours to related work which can be seen as orthogonal to our approach.

Rule/Constraint-Based Process Specification: In [20–22], approaches dealing with the specification of inter-task interdepencies are introduced. One of the main concerns in [21,22] is to provide a scheduler which schedules the tasks (and their events like commit or abort) according to the specified dependencies. A workflow is considered a set of dependencies between tasks which are specified using different formalisms, e.g., computational tree logic (CTL) [21]. Although focussing on inter-workflow dependencies, the basic ideas of the approach presented in [23] are quite similar to the approaches mentioned before. One of the main differences is that the approach in [23] uses an extension of regular expressions for expressing legal executions of workflows.

For verifying the runtime execution, a state automaton based approach is employed. Due to their transactional background, [20–23] focus on scheduling upcoming event requests (e.g., the start of a task) according to predefined dependencies whereas in our approach, violations of dependencies shall be detected as early as possible (i.e., in advance). Furthermore, also the aspect of process change is not addressed. In [24], an approach to declaratively specify process models using a graphical notation which can be mapped to formulas in Linear Temporal Logic (LTL) is presented. In order to enact the process model, the LTL formulas can be synthesized into automatons.

Other approaches on constraint-based process specifications have been introduced, for example using Concurrent Transaction Logic (CRT) [25] or path constraints [26]. These approaches aim at providing a formalism which allows for defining local and global dependencies (constraints) between tasks and also for verification of the specified constraints. However, the verification is only considered for the static case (comparable to semantically verifying a process template). In [27–29] workflows are modeled based on rules or constraints. However, these approaches do not aim at verifying semantic constraints.

Adaptive/flexible PMS: Current approaches on adaptive PMS mainly focus on structural aspects (e.g., [3,4,8,30]) or have a different notion of semantic correctness (e.g., [31,1]).

Some approaches (e.g., [32,33]) aim at enabling more flexible process executions. The basic idea is to allow the execution of a partly defined process template (core process). At runtime, once the flexible parts of the process (predefined as a pocket of flexibility which is comparable to an unspecified subprocess) is reached, a process expert can complete the specification of the subprocess. The subprocess is then verified based on constraints (e.g., composition constraints), which are used to restrict the composition possibilities. Although these approaches rather aim at allowing for more flexibility they are very inspiring for our work, especially the constraints they introduced. However, the verification of the subprocess can be compared to verifying a process template. Verification of real ad hoc changes at runtime are not addressed.

In [34,35], an approach for ensuring the integrity of process instances based on rule-based process adaptation is introduced. Rules are applied, when certain conditions (e.g., too high blood pressure) apply. Using the ADEPT framework [18] the process is adapted in an ad hoc manner according to the rule triggered. This

approach is not directly applicable to process verification but is interesting for preserving the semantic integrity of process instances at runtime.

Integration of heterogenous sources: Many related approaches focus on integrating heterogenous resources (e.g., [36–38]) and semantic web service composition (e.g., [39,40]). In particular, tasks and their parameters are often described using ontologies. When a process is composed, the PMS can check, whether the tasks and their parameters semantically fit together. However, these approaches do not consider semantic constraints over processes.

Knowledge-based systems: Besides, there are also approaches which we consider as being orthogonal to our work. Interesting in this context are knowledge-based systems [13]. In such systems, expert knowledge can be deposited. Recently, knowledge-based systems became more popular in the form of Business Rule Management Systems (BRMS), e.g., commercially available systems like ILOG JRules [41]. BRMS provide convenient interfaces for managing and deploying *business rules* (e.g., "if credit risk >50%, then refuse loan"). By employing techniques known from knowledge-based systems, the rules can be evaluated in the Business Rule Engine (BRE) of the BRMS. In the context of PMS, a BRE is primarily used for decision making (e.g., which outgoing paths of a task to follow). This is done by predefining decision making points in the process and business objects (e.g., a customer's credit risk) used for passing data from the PMS to the BRE and vice versa. At runtime, the BRE is then invoked and can operate over the predefined business objects [9,41]. Approaches concerning the integration of organizational memory information systems (OMIS) into process management (e.g., [42,43]) can also be considered orthogonal to our work. In particular, [42,43] deal with providing background information (e.g., former experience with a particular customer) based on context information (e.g., the customer) in order to support users when executing knowledge-intense tasks.

A Posteriori Verification: In [44] an approach for verifying properties of past processes by verifying execution logs is introduced. This approach can help detecting constraint violations. However, being an a posteriori verification approach this approach is orthogonal to our work. These two approaches can complement each other.

9. Summary and outlook

Our objective is to enable adaptive PMS to perform semantic verification. However, semantic verification must not be conducted in an isolated manner. In fact, semantic verification must be incorporated into typical adaptive PMS interaction scenarios such as ad hoc changes and template evolution. In this paper, we introduced a framework for the integration of domain knowledge within an adaptive PMS by using semantic constraints. Based on these constraints, a generic criterion for semantic correctness of processes has been provided. We have shown how this criterion can be generally ensured. Furthermore, we have addressed the issue of verifying semantic correctness of process changes and verifying the semantic compliance of individually modified process instances with a modified template. Exemplarily for block-structured process meta models, we have shown how semantic process verification can be realized in an efficient manner. Finally, an architecture for the integration of semantic constraints within an adaptive PMS has been presented.

Using our approach, all semantic conflicts caused by violation of dependency and mutual exclusion constraints can be avoided. However, the expressiveness of the presented constraints is limited. Therefore, in future work we will extend our framework, e.g. by introducing context restrictions on constraints concerning their validity (e.g., time or location) or by introducing constraints on other levels of granularity than the task level (e.g., data). Furthermore, we want to develop further methods to efficiently verify semantic correctness within an adaptive PMS. For example, we want to analyze how the information referred to by semantic constraints can be organized (e.g., within an ontology) in order to decrease evaluation effort. All considerations are to be implemented within the adaptive PMS ADEPT2 (www.aristaflow.com).

References

[1] W. van der Aalst, T. Basten, Inheritance of workflows: an approach to tackling problems related to change, Theor. Comput. Sci. 270 (1–2) (2002) 125–203.
[2] F. Casati, S. Ceri, B. Pernici, G. Pozzi, Workflow evolution, DKE 24 (3) (1998) 211–238.

[3] S. Rinderle, M. Reichert, P. Dadam, Flexible support of team processes by adaptive workflow systems, DPD 16 (1) (2004) 91–116.
[4] M. Weske, Formal foundation and conceptual design of dynamic adaptations in a workflow management system, in: HICSS '01, IEEE Computer Society, Washington, DC, USA, 2001, p. 7051.
[5] K. Kochut, J. Arnold, A. Sheth, J. Miller, E. Kraemer, B. Arpinar, J. Cardoso, IntelliGEN: a distributed workflow system for discovering protein–protein interactions, DPD 13 (1) (2003) 43–72.
[6] M. Reichert, S. Rinderle, P. Dadam, On the modeling of correct service flows with BPEL4WS, in: EMISA '04, 2004, pp. 117–128.
[7] S. Rinderle, M. Reichert, P. Dadam, Correctness criteria for dynamic changes in workflow systems – a survey, DKE 50 (1) (2004) 9–34.
[8] S. Rinderle, Schema Evolution in Process Management Systems, Ph.D. Thesis, University of Ulm, 2004.
[9] A. Rowe, S. Stephens, Y. Guo, The use of business rules with workflow systems, in: W3C Workshop on Rule Languages for Interoperability, 2005.
[10] A. Boxwala, M. Peleg, S. Tu, GLIF3: a representation format for sharable computer-interpretable clinical practice guidelines, Biomed. Inform. 37 (3) (2004) 147–161.
[11] S. Quaglini, M. Stefanelli, A. Cavallini, G. Micieli, C. Fassino, C. Mossa, Guideline-based careflow systems, Artif. Intell. Med. 20 (1) (2000) 5–22.
[12] L.T. Ly, S. Rinderle, P. Dadam, Semantic correctness in adaptive process management systems, in: BPM '06, LNCS, vol. 4102, Springer, 2006, pp. 193–208.
[13] F. Hayes-Roth, N. Jacobstein, The state of knowledge-based systems, Commun. ACM 37 (3) (1994) 26–39.
[14] B. Schultheiß, J. Meyer, R. Mangold, T. Zemmler, M. Reichert, The modelling of a surgery process, Interne Ulmer Informatik-Berichte DBIS-6, Ulm University, Institute of Databases and Informations Systems, in german, 1996 (in German).
[15] S. Rinderle, M. Reichert, P. Dadam, Disjoint and overlapping process changes: challenges, solutions, applications, in: CoopIS '04, 2004, pp. 101–120.
[16] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, et al., BPELWS – Business Process Execution Language for Web Services, BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems, 2003.
[17] S. Rinderle, M. Reichert, P. Dadam, On dealing with structural conflicts between process type and instance changes, in: BPM '04, 2004, pp. 274–289.
[18] M. Reichert, P. Dadam, ADEPT$_{flex}$ – supporting dynamic changes of workflows without losing control, JIIS 10 (2) (1998) 93–129.
[19] M. Reichert, Dynamic Changes in Workflow-Management-Systems, Ph.D. Thesis, University of Ulm, Computer Science Faculty, 2000 (in German).
[20] J. Klein, Advanced rule driven transaction management, in: Proceedings of IEEE COMPCON '93, 1991, pp. 562–567.
[21] P. Attie, M. Singh, A. Sheth, M. Rusinkiewicz, Specifying and enforcing intertask dependencies, in: Proceedings of the VLDB '93, Morgan Kaufmann Publishers Inc., 1993, pp. 134–145.
[22] M.P. Singh, Semantical considerations on workflows: an algebra for intertask dependencies, in: Workshop on Database Programming Languages, Springer-Verlag, London, UK, 1996, p. 5.
[23] C. Heinlein, Workflow and process synchronisation with interaction expressions and graphs, in: ICDE '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 243–252.
[24] M. Pesic, W. van der Aalst, A declarative approach for flexible business processes management, in: Business Process Management Workshops, 2006, pp. 169–180.
[25] H. Davulcu, M. Kifer, C.R. Ramakrishnan, I.V. Ramakrishnan, Logic based modeling and analysis of workflows, in: PODS, 1998, pp. 25–33.
[26] W. Fan, S. Weinstein, Specifying and reasoning about workflows with path constraints, in: ICSC, 1999, pp. 226–235.
[27] P. Dourish, J. Holmes, A. MacLean, P. Marqvardsen, A. Zbyslaw, Freeflow: mediating between representation and action in workflow systems, in: CSCW '96, ACM Press, New York, NY, USA, 1996, pp. 190–198.
[28] G. Knolmayer, R. Endl, M. Pfahrer, Modeling processes and workflows by business rules, in: W. van der Aalst et al. (Eds.), BPM '00, LNCS, vol. 1806, Springer, 2000, pp. 16–29.
[29] J. Wainer, F. de Lima Bezerra, P. Barthelmess, Tucupi: a flexible workflow system based on overridable constraints, in: ACM SAC '04, 2004, pp. 498–502.
[30] M. Weske, Flexible modeling and execution of workflow activities, in: Proceedings of the Hawaii International Conference on System Sciences, Hawaii, 1998, pp. 713–722.
[31] W. van der Aalst, T. Basten, H. Verbeek, P. Verkoulen, M. Voorhoeve, Adaptive workflow: on the interplay between flexibility and support, Interprise Inform. Syst. (2000) 63–70.
[32] S. Sadiq, M. Orlowska, W. Sadiq, Specification and validation of process constraints for flexible workflows, Inf. Syst. 30 (5) (2005) 349–378.
[33] S. Deng, Z. Yu, Z. Wu, H. Lican, Enhancement of workflow flexibility by composing activities at run-time, in: ACM SAC '04, 2004, pp. 667–673.
[34] U. Greiner, J. Ramsch, B. Heller, M. Löffler, R. Müller, E. Rahm, Adaptive guideline-based treatment workflows with AdaptFlow, in: CGP '04, 2004, pp. 113–117.
[35] R. Müller, U. Greiner, E. Rahm, AgentWork: a workflow system supporting rule-based workflow adaption, DKE 51 (2004) 223–256.
[36] J. Pathak, D. Caragea, V. Honovar, Ontology-extended component-based workflows: a framework for constructing complex workflows from semantically heterogeneous software components, in: SWDB '04, 2005, pp. 41–56.
[37] S. Bowers, K. Lin, B. Ludäscher, On integrating scientific resources through semantic registration, in: SSDBM '04, IEEE Computer Society, Washington, DC, USA, 2004, p. 349.

[38] J. Kim, M. Spraragen, Y. Gil, An intelligent assistant for interactive workflow composition, in: International Conference on Intelligent User Interfaces, ACM Press, 2004, pp. 125–131.
[39] J. Cardoso, A. Sheth, Semantic e-workflow composition, JIIS 21 (3) (2003) 191–225.
[40] R. Zhang, I. Budak Arpinar, B. Aleman-Meza, Automatic composition of semantic web services, in: International Conference on Web Services, 2003, pp. 38–41.
[41] ILOG, ILOG JRules and IBM MQWF – White Paper, 2005.
[42] A. Abecker, A. Bernardi, S. Ntioudis, L. van Elst, R. Herterich, C. Houy, M. Legal, G. Mentzas, S. Müller, Workflow-embedded organizational memory access: the DECOR project, in: IJCAI '01, Workshop on Knowledge Management and Organizational Memories, 2001.
[43] C. Wargitsch, T. Wewers, F. Theisinger, An organizational-memory-based approach for an evolutionary workflow management system – concepts and implementation, in: HICSS (1), 1998, pp. 174–183.
[44] W. van der Aalst, H. de Beer, B.van Dongen, Process mining and verification of properties: an approach based on temporal logic, in: CoopIS '05, 2005, pp. 130–147.

Linh Thao Ly studied Computer Science at the University of Ulm (Germany). At present she is a Ph.D. candidate of the Institute of Databases and Information Systems of the University of Ulm. Her research interests include adaptive process management systems and integration and verification of semantic constraints in process management systems.

Stefanie Rinderle obtained her Ph.D. in Computer Science at the Institute of Databases and Information Systems, University of Ulm (Germany) where she is currently teaching and working on her habilitation. During her postdoc Stefanie stayed at the University of Twente (The Netherlands), the University of Ottawa (Canada), and the Technical University of Eindhoven (The Netherlands) where she worked on several projects on process visualization and modeling as well as on process mining. Her research interests include adaptive process management systems, integration of application knowledge, and the controlled evolution of organizational structures and access rules.

Peter Dadam has been full professor at the University of Ulm and director of the Institute of Databases and Information Systems since 1990. Before he came to the University he had been director of the research department for Advanced Information Management (AIM) at the IBM Heidelberg Science Center (HDSC). At the HDSC he managed the AIM-P project on advanced database technology and applications. Current researach areas include distributed, cooperative information systems, workflow management, and database technology and its use in advanced application areas.