Supporting Adaptive Workflows in Advanced Application Environments

Manfred Reichert, Clemens Hensinger, Peter Dadam
Department Databases and Information Systems
University of Ulm, D-89069 Ulm, Germany
Email: {reichert, hensinger, dadam }@informatik.uni-ulm.de

Abstract

The need for supporting adaptive workflows (WFs) is widely recognized. For many business processes (BPs) it is nearly impossible to consider all possible task sequences already at the design level. Besides this, ongoing business cases may also have to be adapted to organizational and functional changes in their environment. A basic step towards adaptive workflow management systems (WfMSs) is the support of run-time WF specification as well as of dynamic WF changes. Such changes may affect only a single active WF instance or may affect multiple instances of a particular WF type. To adequately support adaptive WFs, it is important to understand why processes change and which kinds of changes may occur. In this paper we use clinical application scenarios to explain and to elaborate the functionality needed to support dynamic WF changes in an advanced application environment. The paper addresses conceptual issues related to ad hoc changes of a single WF instance on the one hand, and it discusses issues related to WF schema changes and their propagation to its active instances on the other hand. We show that the different levels of changes must be considered in conjunction and we use the ADEPT concepts to illustrate how an integrated approach could look like.

1 Introduction

While data-centered application systems tend to remain stable (i.e., unchanged) for rather long periods of time, process-oriented applications must be modified whenever the BP they support changes, and this may happen rather frequently in real working environments (see [EKR95], [Shet96], [ShKo97], and [Sieb96]). In a hospital, for example, we find BPs whose planning and execution overlap, ad-hoc cases for which no standard plan exists, unforeseen events leading to ad-hoc deviations from the pre-planned BP, or functional and organizational changes requiring modifications in the definition of standard processes. Once an application system has been made to behave process-oriented, it should support these cases and reflect the changes of a BP very quickly; otherwise its benefit would be low.

Process-oriented WfMSs offer a promising technology to achieve this goal. They allow modeling the control and the data flow of a BP explicitly and separately from the implementation of the application components. In principle, it is possible to implement the application functions as isolated components, which can expect that their input parameters are provided by the run-time environment upon invocation and which only have to worry about producing correct values for their output parameters. If the components are properly implemented and the data dependencies between them have been made explicit, a WF can be adjusted to changes of a BP with a relatively low effort when compared to conventional programming approaches [LeRo97]. Today's WfMSs, however, have been primarily designed for the support of well-structured BPs showing little variations in their possible task sequences. They implicitly assume that all aspects of a BP and all tasks are known in advance to the WF designer, and they rather enforce a strict execution of the pre-modeled WF. Adaptive WFs and particularly dynamic WF changes are supported only rudimentarily. At the level of single WF instances, some WfMSs allow users to deviate from the pre-modeled WF at run-time, but at the risk of inconsistencies and errors. Finally, little support is available for changing the definition of a WF type and for applying these changes to already active instances as well.

To adequately support adaptive WFs, it is important to understand why processes change and which kinds of changes occur in practice. In this paper we use clinical application scenarios to explain and to elaborate the functionality needed to support dynamic WF changes in an advanced application environment. Although clinical processes are used for explanatory purposes, the problems addressed are also valid for other non-trivial application areas (see [BFG93], [ShKo97], [Sieb96], and [Wes98]).

The paper is organized as follows: In Section 2 we describe characteristic properties and requirements of clinical WFs, concentrating on dynamic WFs and on WF changes. In Section 3 we show that these aspects must be considered in conjunction, and we use the *ADEPT* concepts to illustrate how an integrated approach could look like. Section 4 discusses related work and concludes with a summary.

2 Clinical Processes

The in-depth understanding of the characteristic types of processing, the organizational structures, the flexibility requirements of the medical personnel, the kinds of exceptions and deviations occurring in clinical processes, and the adequate reactions on such events is indispensable for the support of clinical processes. In hospitals, we find BPs of different complexity and duration: Simple ones, like order entry and result reporting for laboratory and radiology, but also complex and long-running (even cyclic) BPs like diagnostic treatment of a patient or chemotherapy. These BPs may be highly dynamic with an overlapping planning and execution of tasks on the one hand, or may follow strictly predefined medical procedures that normally have to be obeyed on the other hand. But even for well-structured and repetitive BPs, there are many circumstances under which ad-hoc deviations are mandatory.

2.1 Working Situation

When efforts are taken to automate the flow of clinical processes, it is important to realize the *working situation* under which WF technology must prove its usefulness and applicability. The cooperation between organizationally separate units is an important task in a hospital with repetitive but nevertheless non-trivial character. Medical and nursing care involve clinical tasks that may be critical to patient care on the one hand, and it comprises time-consuming organizational responsibilities on the other hand. Medical procedures and tests must be planned and prepared, appointments be made, and results be obtained and evaluated. We find personnel working under extremely high time pressure who often must make important decisions about patient treatment within a rather short period of time, and we find personnel who is confronted with a massive load of unstructured data that have to be processed and put into relation to the problems of their patients. In addition, the working situation is burdened by frequent context switches. Unforeseen events and emergency situations occur, patient status changes, information necessary to react is missing. Because of this, many coordination problems result, leading to unnecessary long hospital stays and increasing costs or invasiveness of patient treatment. In critical situations, missing or erroneous patient information may even cause late or wrong decisions. For all these reasons, a process-oriented information system, which helps to coordinate and to schedule clinical and organizational tasks would be highly welcome by the medical personnel.

2.2 Ad-hoc Deviations From Pre-Planned Processes

For the WF-based support of even well structured and repetitive clinical processes it is extremely important not to restrict the physician or the nurse. Any attempt to automate the flow of patient processes will fail, if *rigidity* comes with it. Variations in the course of a disease or a pre-planned treatment process are deeply inherent to medicine; the *unforeseen event* is to some degree a "normal" phenomenon. Medical personnel must be free to react and is trained to do so. For example, if physicians come to the conclusion that for a patient an additional medical test, which has not been anticipated in the process plan, is needed they will adjust the plan accordingly. In emergency situations, physicians may perform an intervention immediately without finishing preparatory measures required for the normal case. In such situations, a process participant may wish to collect information about the patient (e.g., the result of a previous medical test) by phone and afterwards proceed with the process, without waiting for the (electronic) report to be written; i.e., this documentation step is skipped and worked on later. Finally, if the prerequisites for a medical examination are dissatisfied, the physician must be free to abort it and to repeat it later (including the repetition of preparatory steps), to schedule an alternative procedure, or to do anything else.

Such *exceptions* are frequent and inherent to clinical processes. Adequate reactions on them include (among other things) that tasks are repeated, skipped, modified, postponed, or undone, that pre-planned task sequences are changed (i.e., the order of tasks is rearranged), or that alternative or new tasks are scheduled. For *cyclic process structures* – as in the case of a chemotherapy – such ad-hoc changes may concern a single treatment cycle or all (or part) of them.

2.3 Dynamically Evolving Patient Processes

Besides well-structured and repetitive medical procedures, the personnel is involved in long-term patient processes for which the planning and execution of tasks overlap. In a treatment process, usually several well-structured diagnostic and therapeutic procedures are carried out. Before an invasive medical intervention is performed, for example, the patient has to undergo numerous preliminary medical examinations. Each of them may require additional preparations and aftercare. While some of these measures are known in advance and may therefore be considered in the overall process plan already at the design level, others have to be scheduled dynamically depending on the patient's state of health and on the results of previously performed tests. For these reasons, a patient process cannot be always modeled on a fine-grained level before the execution of the process starts.

The (dynamic) planning of the patient flow is a very complex and error-prone task, since activities may be closely related to each other due to clinical, organizational, or logistic reasons. Because of this, they can neither be executed sequentially nor completely independent from each other. For a particular patient, for example, medical interventions may have to be performed in a certain order or with a minimum or maximum time distance between them (see [DaKl98] for an example). Such interdependencies between tasks are deeply inherent to medicine and must be considered in the planning of the patient process. Finally, for dynamically evolving patient processes, again we are faced with the problem of ad-hoc changes as described in Section 2.2.

2.4 Changes of Standard Processes

Up to now we have only considered changes that affect a single (patient) process. Modifications in the definition of a standard process and the adaptation of its ongoing cases may become necessary as well. Reasons for them may be the availability of new diagnostic or therapeutic tests, the adjustment of processes to a new law, the optimization of processes in conjunction with reengineering and quality management efforts, or the restructuring of the hospital organization itself. A process-oriented clinical application will therefore not accurately represent the BPs of the health care organization for long. Instead, mechanisms must be foreseen for handling organizational changes at the system level; otherwise, over time the mismatch between the real (patient) processes and those supported by the system will increase. Ideally, changes of standard processes can be introduced on the fly without substantial delays. Especially in the context of long-running BPs, it may also be desirable to apply them to already ongoing patient cases as well.

2.5 Requirements

On the one hand, process-oriented application systems would be highly welcome by the medical personnel. On the other hand, for clinical processes it is nearly impossible – except in very simple cases – to consider all possible task sequences and all deviations that may occur already at the design level. For WF-based clinical applications, it is therefore extremely important that users may gain complete initiative whenever they need it. A process-oriented information system must offer simple to use interfaces to the medical personnel for the handling of scheduled tasks as well as for the *ad-hoc deviation* from the pre-planned process. In addition, methodological support is needed for the *dynamic planning* of a single patient process i.e., for the dynamic composition of pre-defined tasks descriptions. The resulting plan must then be automatically mapped onto an operational WF model.

Depending on their *privileges*, users must be able to change the structure of a single WF instance – temporarily or permanently – by adding or deleting tasks, by rearranging the order of tasks, by suspending single tasks or whole processes, or by changing task attributes (e.g., role assignments and deadlines). Corresponding changes must be properly integrated, especially with respect to *authorization* and *documentation*; i.e., any deviation from the standard process must be recorded. Furthermore, the alteration of a single WF instance must be possible without affecting its original template. Finally, to deviate from a pre-modeled WF must not be complicated for the user. For example, the *complexity* concerning the re-mapping of input/output parameters of the components affected by a change must be hidden to a large degree from users. Generally, it is also not acceptable that the user must check whether an envisaged modification may cause any run-time problems in the sequel (e.g., cyclic waits leading to deadlocks, lost updates, missing data due to the skipping of a process step, or violated temporal constraints). Such *checks* have to be done by the system, ideally without performance penalty.

The issues discussed so far mainly concern changes of a single WF instance without affecting its original template. In conjunction with changes in the definition of a standard BP (see Section 2.4), *modifications of a WF template* may become necessary as well. As many clinical processes are of long duration, it must be possible to make corresponding changes on the fly and to apply them not only to future instances of the corresponding WF template, but to in-progress WF instances as well. This is not as trivial as it looks like at first glance, since the propagation of template changes to a WF instance may not only depend on the current state of the instance, but also on ad-hoc changes previously applied to it.

3 Conceptual Issues in Supporting Dynamic Workflow Changes

Most of the issues addressed in the previous section cannot be treated reasonably well when considered in an isolated fashion only. In the *ADEPT* project we are trying to look at the different facets of dynamic WF changes in an integrated manner. In the following, we illustrate some of the problems and sketch how an integrated solution for supporting dynamic WF changes could look like. We only describe here those parts of the *ADEPT* methodology, which are necessary for this discussion.

3.1 WF Modeling and Execution

With increasing complexity and expressive power of a WF model, it becomes more and more difficult to handle dynamic WF changes and their side-effects in a proper and secure manner. The challenge on the one side is to find a modeling technique that allows the WF designer to *adequately* describe all types of relevant processes. The challenge on the other side is to keep such modeling techniques *learnable* and *usable* for both, the WF designer as well as (to some degree) the end-users. To change structural components of an active WF instance, its representation (or a partial view on it) must be understandable to the end-user, or at least to those end-users which have the privilege to perform dynamic changes. Apart from this, users must be sure that any change initiated by them will not cause inconsistencies (e.g., unintended lost updates) or run-time errors (e.g., program crashes due to the invocation of task components with missing input data). As motivated in Section 2 the necessary checks have to be performed by the system and not by the end-user. To achieve this, the model must define consistency and correctness properties allowing to detect (or to avoid) problems like non-termination of the WF, the passing of wrong or missing input data to an application component, or temporal inconsistencies, for example. For a given WF model, these properties must be validated already at the design level, and they have to be preserved whenever a change is applied to instances of that model at run-time.

For all these reasons we have dismissed the idea of using rather unstructured models like, for example, Petri nets or ECA rules for WF modeling. Instead, we have adopted concepts from *block-structured* process description languages. We have enriched them by introducing additional *control structures* (e.g., for synchronizing the execution of tasks from parallel execution branches) and by explicitly representing the *process state* in the model. Figure 1 illustrates the main philosophy how processes are modeled in *ADEPT*. It shows a very simplified part of a clinical WF: A patient passes through multiple treatment cycles. In each of them she or he is treated with a medicament of which the dose is taken depending on her or his current weight and height. In order to avoid interactions with possible problems on the side of the patient, an additional lab-test is performed before the medication. The graphical representation of the WF as shown in figure 1 is not intended for the physician or the nurse. However, even more "end-user-friendly" interfaces will somehow reflect the basic concepts described here to avoid too large discrepancies between the user’s mental model and the model used by the system.

In the *ADEPT* WF model different types of split and join nodes can be used to describe different kinds of *branching* (including parallel and conditional branching). In addition, in contrast to many other WF models, *ADEPT* provides an *explicit loop construct* (see figure 1). This does not only improve the readability of the WF model, but it also allows distinguishing between intentionally modeled loops and unintentionally modeled cycles. Furthermore, at run-time the event triggering the next iteration of the loop is well-known to the system, which allows supporting advanced features like the (efficient) undoing of temporary structural changes before the next loop iteration is started (see Section 3.2). A branching or a loop is always modeled in a block-oriented fashion having exactly one entry and one exit node. These blocks may be nested but they are not allowed to overlap. As this limits the expressive power of the model, in addition, *ADEPT* provides different types of so-called *synchronization edges* which can be used to express different kinds of “wait-for” situations in concurrent executions (see also Section 3.2). The use of these synchronization edges has to meet several constraints in order to avoid “bad cycles” leading to termination problems of the WF at run-time.

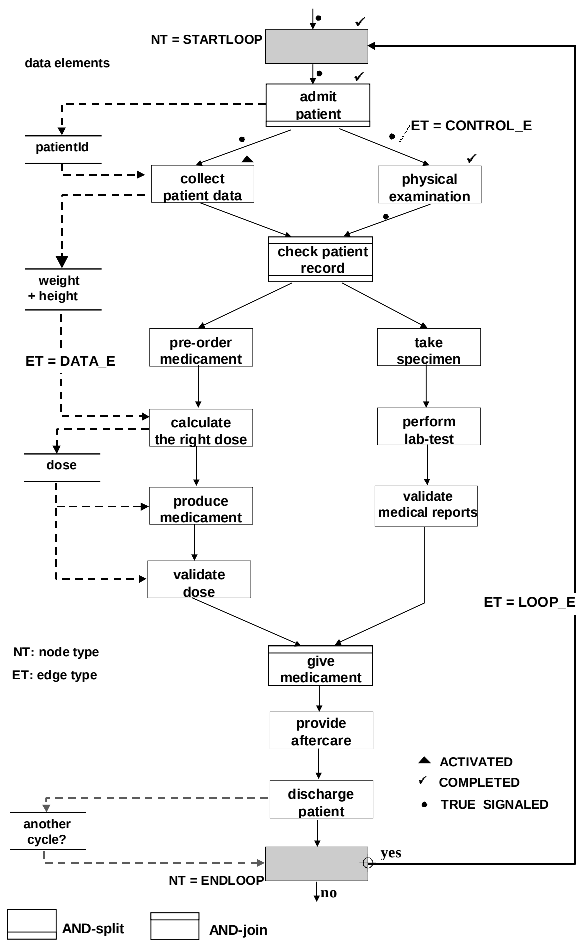

Fig. 1: Modeling and Executing Processes in *ADEPT*

The *data flow* of a WF is defined by a set of *data edges* connecting the input/output parameters of tasks with global *data elements* of the WF. Data elements store the data versioned to allow partial rollback – even to an earlier iteration of a loop. *ADEPT* imposes a set of *constraints* governing the nature of a correctly modeled *data flow schema*. Before a task component is invoked, for example, all mandatory input parameters must be supplied. That means, all data elements to which the task’s input parameters are connected must be written at least once within all valid task sequences leading to the activation of the task. Furthermore, tasks from different branches of a parallel branching may not write the same data element, unless they are explicitly synchronized by a synchronization edge.

The control structures described so far are the same for the description of WF templates and of WF instance graphs. At the WF instance level, in addition, special labels are used to explicitly describe the current status of nodes (e.g., *ACTIVATED, RUNNING*, or *COMPLETED*) and the status of edges (e.g., *TRUE_SIGNALED* or *FALSE_SIGNALED*). *ADEPT* uses a set of *marking rules*, which define the conditions under which the labeling of nodes and edges must be changed. After completing an AND-split node, for example, all outgoing edges are signaled as TRUE. This, in turn, may lead to the activation of succeeding steps.

In summary, we have selected the block structure because it is rather quickly understood by users, it allows to provide syntax-driven WF editors, and it also allows the implementation of efficient algorithms for checking the correctness properties defined by *ADEPT*. For more details on the ADEPT workflow model, the interested reader is referred to [ReDa98].

3.2 Ad-hoc changes of a Single WF Instance

When a WF instance graph is changed, it must be ensured that the resulting graph is *syntactically correct* and has a *legal state*. Any modification must lead again to a proper block structure, and it must preserve the consistency of the WF instance graph. *ADEPT* offers a set of change operations to end-users, which can be applied for the proper and secure handling of ad-hoc deviations from pre-modeled WF templates. Depending on their privileges, users may *add tasks* as well as whole task blocks (as sequential or as parallel steps), may *delete tasks*, may *rearrange the order of tasks*[1], may initiate a *partial rollback* of the WF, or may *change task attributes* (e.g., role assignments, deadlines, or binding of resources). All changes are registered in a *history* and they are properly integrated with respect to *authorization* and *documentation*. For each change operation, *ADEPT* defines the *preconditions* for its use, *graph transformation* rules[2], the *semantics* of the resulting graph substitutions, mechanisms for detecting possible *problems* and *side-effects*, and *policies* for handling these problems. It is important to mention that even simple operations like skipping a task may require a non-trivial restructuring of the WF instance graph in order to regain its correctness and consistency (see below).

The applicability of a change operation depends on the state of the WF instance under consideration and on its structural properties. For the deletion of a task, for example, the following *state constraint* must be made: A task may not be removed from a WF instance graph if it has already been labeled as *COMPLETED* or *RUNNING*; i.e., the component associated with this task has already been invoked. As an example for a *structural constraint* take the addition of a new task to a WF instance graph as a successor of the node *X* and as a predecessor of the node *Y*. This insertion would not be allowed if *Y* precedes *X* in the flow structure; in this case bad cycles leading to deadlocks at run-time might result. Other kinds of constraints (e.g., temporal constraints, security constraints, and user defined integrity constraints) are outside the scope of this paper.

Due to lack of space, we omit further details and present an example instead. Let us assume that the WF instance graph at a certain point in time looks like as the one depicted in figure 1. The steps represented by the nodes “admit patient” and “physical examination” have been completed in the current loop iteration and the task “collect patient data” has been activated (i.e., routed to worklists). Let us further assume that an *exceptional situation* occurs making it impossible for the nurse to perform the activated step at the moment; e.g., she may not know where her patient is. Instead the nurse may wish to skip this step for the time being (i.e., to shift it to a later point in time) and to work on the task “check patient record” immediately. This task is executable, in principle, as it is not data-dependent on the task “collect patient data” and all other predecessors have already been completed.

Skipping the task “collect patient data” means to remove it *temporarily* from the WF instance graph. In doing so, its data edges are deleted as well, which may lead to missing or incomplete input data of succeeding steps and thus to a violation of the correctness of the WF instance graph. To avoid such cases, at first *ADEPT* checks whether there are succeeding tasks that are data-dependent on the step to be deleted. In the example, there is exactly one successor of the task “collect patient data” satisfying this criterion, namely the task “calculate the right dose”. Assume that this task represents an automated step of which the WF designer has disallowed the deletion (by having marked the node accordingly). If we had removed the node “collect patient data” without any further adaptation, this would have caused serious consequences. The component for the step “calculate the right dose” would then be invoked later, although mandatory input data would be missing. This might lead to a program crash or – which would be even more terrible – to wrong output data (i.e., a wrong dose). Because of this, *ADEPT* accepts a change request only if no violation of the defined correctness properties occurs. Concerning the deletion of a task *X*, for example, several *policies* are provided to deal with the problem of missing data:

1. Data-dependent steps are deleted as well

[1] e.g., by *skipping tasks* with or without working on them later or by *jumping forward* to a currently inactive part of a WF instance graph

[2] These rules are based on a complete and minimal set of basic change primitives like AddNode, DeleteNode, SetNodeAttribute, AddEdge, AddDataElement, or SetEdgeState, for example.

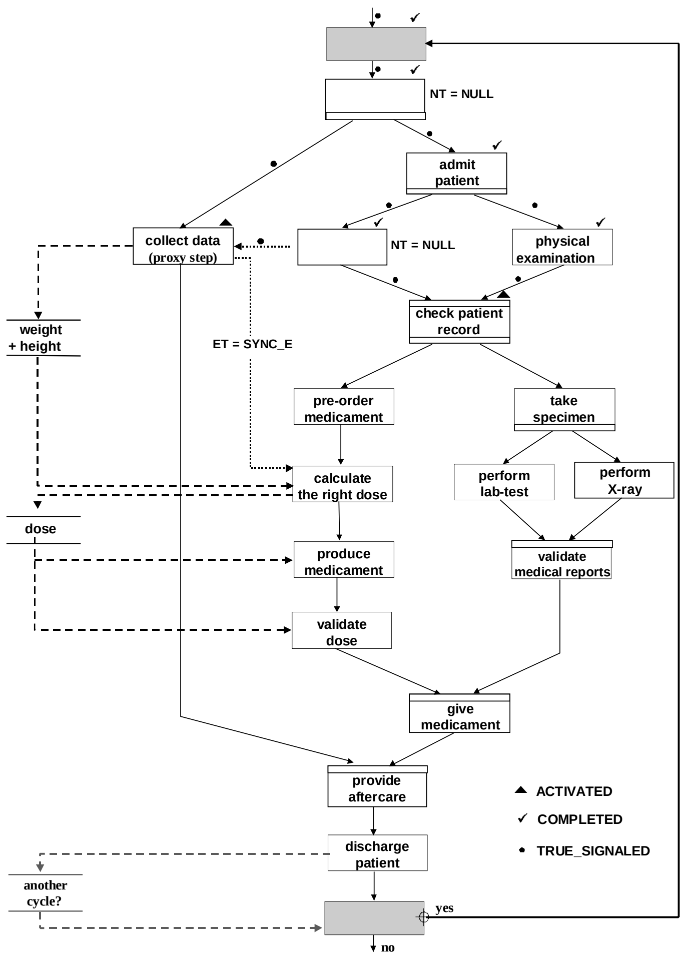

Fig. 2: Ad-hoc changes of a single WF instance graph in *ADEPT*

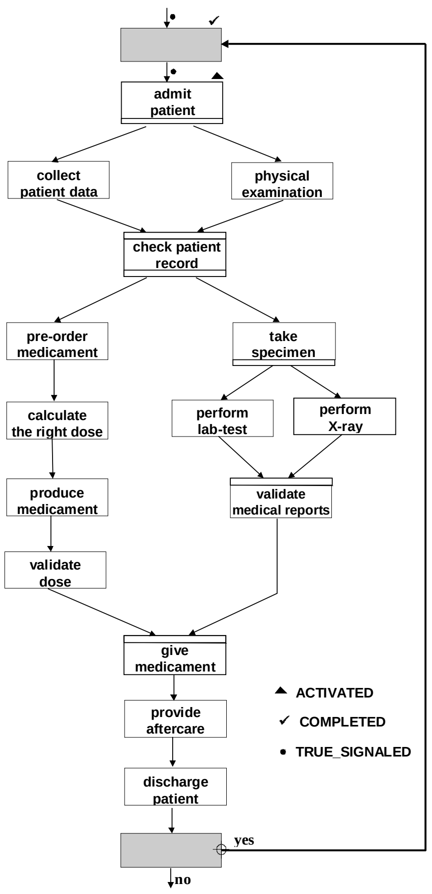

Fig. 3: The same graph after entering the next loop iteration and after undoing the temporary change.

2. A dynamically generated form is activated when the user starts the task with the missing data.
3. An additional provider step X_{prox} is inserted into the instance graph substituting X; i.e., X_{prox} takes over the data links of X and must be completed before any task data-dependent on X is activated.

In our example, the first two variants are not applicable. The step "calculate the right dose" may not be deleted, and it is an automated activity; i.e., prompting the user for the missing data when starting this task will not be possible. Instead, *ADEPT* will offer the user to generate a form and to prompt for the missing values – either immediately or when needed; i.e., variant 3 is chosen. Now the restructuring of the WF graph can begin: First of all, the task "collect patient data" is deleted; this is realized by removing it from worklists and by substituting a "null task" for it in the graph. Then a corresponding provider task ("collect data") is inserted as a parallel path into the instance graph. As this insertion must lead to a proper block structure again, additional nodes and edges are added by the system. This transformation alone would not be correct, however, because we must ensure that the newly inserted step is completed before the task "calculate the right dose" is activated (why?). To enforce this, a synchronization edge leading from the step "collect data" to the step "calculate the right dose" is added to the graph. Following this transformation, the status of nodes and edges is re-evaluated according to the *marking rules* defined by *ADEPT*. Afterwards, the steps "check patient record" and "collect data" can be executed immediately as in both cases all incoming edges are marked as *TRUE_SIGNALED*. The resulting instance graph is shown in figure 2. This graph reflects a second change, which we have not discussed here, namely the insertion of the task "perform X-ray" between the two steps "take specimen" and "validate medical reports".

At this point, it is important to mention that users are not really burdened with the restructuring of the instance graph. They express a change request in a rather declarative way ("skip task *X*", "insert task *Z* between *X* and *Y*", etc.), and they may choose between different policies for handling side-effects. Internally, graph transformation and reduction rules (see [ReDa98]) are used to perform the necessary modifications.

As a last interesting aspect, consider once again the WF instance graph depicted in figure 2. As already mentioned, the skipping of a task – together with the insertion of a substituting provider task – may be only of temporary nature. That means, when the next loop iteration is entered, the modifications made have to be undone. Concerning changes of an instance graph, users may also desire that the applied modifications remain permanent; i.e., the change must be valid for the current as well as for all future iterations of the loop. In our example from figure 2, this might be the case for the insertion of the task "perform X-ray". The differentiation between *loop-temporary* and *loop-permanent* changes is essential when both, loops and dynamic changes are supported in the same system. The handling of them, however, is not as trivial as it looks like at first glance. *ADEPT* defines additional rules that describe when a modification may be loop-permanent and when it can only become loop-temporary. The most important constraint is that a loop-permanent change must not depend on a previously performed loop-temporary modification; otherwise severe inconsistencies or run-time errors may result when the next iteration of the loop is entered. Figure 3 shows how the WF instance graph from figure 2 may look like when the next iteration of the loop is entered (In this graph we omitted the presentation of the data flow.) The loop-temporary change is undone while the loop-permanent one is maintained A more comprehensive treatment of theses issues can be found in [ReDa98].

3.3 Workflow Type Changes and the Handling of Active Instances

So far, we have concentrated on ad-hoc changes at the level of a single WF instance; i.e., modifications of a WF instance graph – either temporarily or permanently – without affecting its original WF template. In this section, we address issues related to *modifications in the definition of a WF type*. In this context, the important question arises how to deal with the active instances of a WF type when its definition is changed. Shall they be finished according to the old template version, or shall all (or part) of them be "migrated" to its new version? And, if the latter is the case, under which circumstances are such on-the-fly changes desirable and possible, and how must we deal with in-progress instances that cannot be (immediately) migrated to the new template? In the following we sketch how these issues are related to each other and how *ADEPT* treats them.

3.3.1 Template Changes and their Complexity

Though there are similar problems between the ad-hoc modification of a WF instance graph and its adaptation due to the release of a new template version, changes of a WF template and the necessary graph transformation tend to be more complex. As an example, think of a modification during which it becomes necessary (among other things) to add a new loop to the WF graph surrounding an already existing block of the flow structure. Generally, we cannot expect the end-user to perform such changes. The WF modeler, however, must be free and is trained to do so. Nevertheless, *ADEPT* describes changes of a WF template similarly to ad hoc changes of a WF instance graph. A change corresponds to a sequence of change operations c_1 ... c_n that – when applied to a correct WF template T – lead again to a WF template T^* with a proper block structure and a correct data flow schema (see figure 4 for an example). To ensure this, *ADEPT* provides a syntax-driven WF editor to the modeler with built-in operations like *AddActivity*, *DeleteActivity*, *AddBlockStructure*, *AddSyncEdge*, or *AddDataElement*. The set of basic change operations offered is *minimal* and *complete* in the sense of allowing each possible form of restructuring of a given WF template. At this point, it is important to mention that the high-level change facilities offered to end-users (e.g., to skip a task or to jump forward to currently inactive tasks) are realized based on the same set of basic change operations, which is also used by the WF modeler when changing a WF template.

Finally, a new version of a WF template is released only if its correctness is ensured. Multiple versions may be derived from the same WF template.

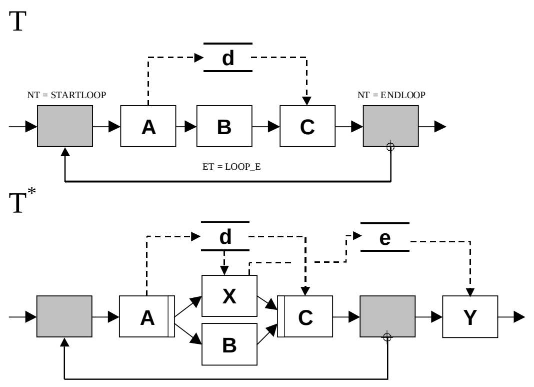

Fig. 4: Change of the template T leading to the new template version T^*. The data element e has been added, activity X has been inserted between A and C (together with two data edges), and activity Y (reading the data element e) has been added.

3.3.2 Policies for Change Propagation

Generally, different policies for handling the instances of a modified WF template are conceivable. In rare cases it may not be desirable that instances of the same WF type, but which are based on different template versions are operational at the same time.

When a new template version T^* is installed, then all instances executed according to the old version T must either be aborted (and re-started if necessary) or, first of all, they must be finished before instances of T^* may be created. Another policy is to allow the WF designer to generate a new version of a WF template and to base the execution of future instances on it, while already running instances of that type are still executed according to the old template. For many practical cases, however, these simple approaches will not be sufficient. Especially in conjunction with long-running WFs – think of, for example, a medical treatment process of which the duration may be up to several months or years – it is often desirable that changes in the definition of a WF type are applied to already active instances as well.

ADEPT does therefore not "hard-wire" any of these policies. Instead, the modeler is free to choose between them and to describe which instances may be adapted to changes of their original template and which may not. For example, she or he may specify that template changes may be applied to a WF instance graph only if it has (not yet) reached a certain state or if it satisfies a set of conditions (like "The instance was created after a certain point in time" or "No ad-hoc changes have been applied to it"). For this, the corresponding WF instances must be qualified in a predicate-like manner. In principle, it is also possible to derive different versions of the same template T and to split the set of currently active instances of T accordingly. Finally, the modeler may fix a time interval in order to define when a new template version is valid.

3.3.3 Handling of Active Instances

To propagate changes of a WF template to in-progress instances is not as trivial as it looks like at first glance. Whether a template change can be correctly applied to a WF instance graph or not, does not only depend on the current state of the instance, but may also be influenced by ad-hoc changes previously applied to its instance graph (see Section 3.2). Further dependencies (e.g., temporal constraints) may be considered as well, but are outside the scope of this paper.

In the following, let T^* denote a WF template that has been derived from the template T by applying a set of change operations $c_1...c_n$ to it. Assume further that w_T denotes a WF instance graph that was created from the template T and of which the execution has not yet been finished. The instance graph w_T may differ from its original template T in two respects: the labeling (i.e., the status) of nodes and edges and – in some cases – the structural components. The latter will be the case if ad-hoc changes are applied to w_T (see Section 3.2). Propagating the template changes $c_1...c_n$ to the WF instance graph w_T now means to apply these changes to this graph as well. Obviously, a necessary condition is that the resulting instance graph satisfies the defined correctness and consistency properties. Generally, this will not always be possible and even if, the propagation may not be desirable due to "semantic conflicts" between the changes $c_1...c_n$ applied to the template T on the one hand and ad-hoc changes $c_1^w...c_m^w$ applied to the instance graph w_T on the other hand.

First of all, let us assume that *no ad-hoc structural changes* have been applied to the instance graph w_T so far. Then – except for the labeling of nodes and edges – the graph w_T and its original template T correspond to each other. For this case, the deciding factor whether the template changes $c_1...c_n$ may be propagated to the instance graph or not, is the current status of its nodes and edges. For each change operation, *ADEPT* defines the pre-conditions an instance graph must satisfy regarding its state. Template changes $c_1...c_n$ may be applied to the instance graph w_T, only if for each change operation c_i $(1 \le i \le n)$ w_T meets these conditions.[3] This will always be the case, for example, if the region of w_T affected by the change has not yet been entered; i.e., the nodes and edges from this region have not been labeled so far.

As an example, consider the template evolution from T to T^* as depicted in figure 4. In the figures 5a and 5b two instance graphs $w_T^{(1)}$ and $w_T^{(2)}$ are shown, which were created from the template T. As they have not been structurally modified so far, the propagation of the template changes solely depends on the current

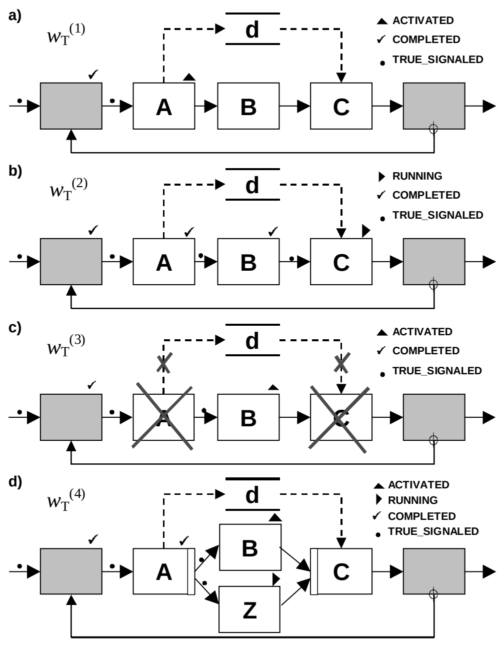

Fig. 5: WF instance graphs created from the template *T* (cf. figure 4a). Note that ad-hoc changes have been applied to the instance graphs shown in figures c) and d). Consequently, their structure differs from the structure of the original template *T*.

state of the instance graphs. The template changes described in figure 4 may be immediately applied to $w_T^{(1)}$ as all change operations – both, the insertion of the task *X* between *A* and *C* and the insertion of *Y* after the loop block – are allowed in the current state of $w_T^{(1)}$. The instance graph resulting from this change propagation is depicted in figure 6a.

Regarding the instance graph $w_T^{(2)}$, however, the template changes cannot be (immediately) applied to it, since the insertion of the task *X* is not permitted in the current state of this graph.[4] In such cases, a simple approach would be to *dismiss the changes* for the WF instance under consideration and to proceed with the flow according to the present template version *T*. Other solutions that can be devised and that have been proposed in the literature (e.g., [BPS97], [Casa96]) include

- the *partial rollback* of the flow to a previous state, that allows correctly applying the changes $c_1...c_n$
- the "migration" of the instance graph to an *alternative template T***, which may be valid only temporarily to handle such specific cases.

These approaches are easy to handle, but they will restrict the practical usability of this feature significantly. Think of the treatment cycle from figure 2; the partial rollback of the flow would not be practicable here. Instead, it would be desirable to dismiss the changes for the current iteration, but to apply them for following iterations of the loop. Because of this, *ADEPT* supports the propagation of template changes at a later point in time as well. The change request will not be dismissed, but will be registered if it cannot be immediately applied to the instance graph due to a status conflict. The modifications will then become effective at the next possible point in time. If no ad-hoc changes are applied to the instance graph $w_T^{(2)}$ in the following, this *late propagation* will be possible when the next iteration of the loop is entered.

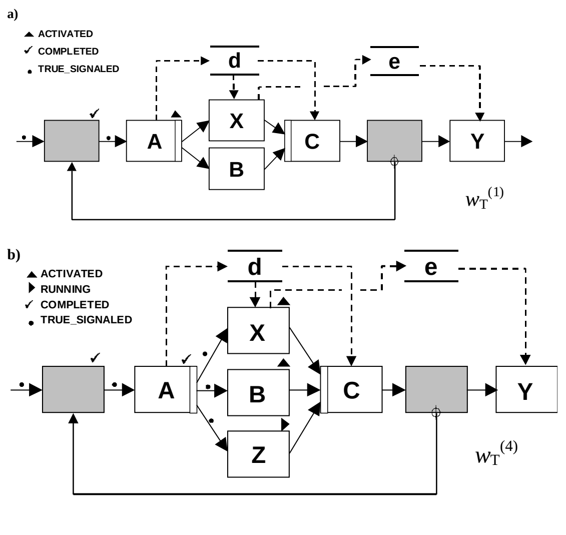

Fig. 6: WF instance graphs from figures 5a and 5d after propagating the template changes to them.

[3] Further checks are not necessary since the structural correctness of T^* has been already validated at the modeling level.

[4] Assume that we have applied the two insert operations to $w_T^{(2)}$ nevertheless. If the loop block is left after the current iteration, the component of the task *Y* will then be invoked with missing input data.

If both, changes at the *instance level* as well as at the *type level* are supported in one system, the important question arises how to deal with WF instances to which ad hoc changes have been previously applied by end-users when their original template *T* is changed. One may argue that in such cases changes in the definition of *T* may not be propagated to these instances at all. This, however, would be too restrictive for many applications – especially in clinical environments where both types of changes frequently occur – and it would also be not necessary in general. Similar like with concurrency control in cooperative environments (cf. [WäKl96]), the problem is to avoid structural as well as semantic conflicts between changes made independently from each other and applied to the same object – in our scenario to the same WF graph (respectively to a copy of it). If such conflicts occur between the ad-hoc changes $c_1^w...c_m^w$ applied to the WF instance graph w_T on the one hand and changes $c_1...c_n$ of its original template *T* on the other hand, the template changes may not be propagated to the instance graph (at least as long as these conflicts cannot be resolved). Due to lack of space, we omit technical details (e.g., concerning conflict tests) as well as issues related to the handling of semantic conflicts. Instead we present two examples focusing on structural conflicts.

As a first one, consider the WF instance graph $w_T^{(4)}$ as shown in figure 5d. This graph differs from its original template *T* (see figure 4) since an ad-hoc change – the insertion of *Z* between *A* and *C* – has been applied to it. Nevertheless, the changes of the template *T* (as shown in figure 4) may be propagated to $w_T^{(4)}$ without causing structural conflicts. The instance graph resulting from this propagation is shown in figure 6b.

As a second example take the instance graph $w_T^{(3)}$ from figure 5c, where the tasks *A* and *C* (together with their data edges) have been deleted. Due to this ad-hoc change, the modifications of the template *T* may not be propagated to $w_T^{(3)}$ at the moment; *X* reads the data element *d*, which is not currently supplied due to the deletion of *A*. If the deletion of the task *A* is loop-temporary (see Section 3.2), the changes may be propagated to $w_T^{(3)}$ at a later point in time; i.e., after the temporary changes, which have caused the structural conflict, are undone and no new conflicts do occur due to ad hoc changes applied in the meantime. In the presented example, this may be the case when the next iteration of the loop is entered (i.e., when the deletion of task *A* is undone). If task *A* was deleted loop-permanently from $w_T^{(3)}$, however, the template changes may not be propagated at all. In all these cases, the system must allow performing the necessary checks very efficiently.

4 Discussion and Summary

The need for adaptive WFs has been identified by several groups (e.g., [BPS97], [Casa96], [DMP97], [EKR95], [ShKo97], [Sieb96], and [Wes98]). The majority of these approaches, however, concentrate only on some aspects related to the dynamic change problem. The proposals made in [BPS97], [EKR95] and [Casa96], for example, deal with WF type changes and their propagation to running WF instances. How to treat WF instances, to which ad hoc changes have been previously applied, is not discussed in this context. Issues concerning the consistency and correctness of dynamic changes (e.g., with respect to the flow of data), the management of loop-temporary and loop-permanent changes, or the late propagation of WF template changes are also not sufficiently addressed. Finally, some interesting proposals have been made in the field of dynamic planning processes (e.g., [DMP97], [Hei96]), which aim at the methodological support of dynamically evolving WFs. More comprehensive treatments of these approaches and of other related work can be found in [ReDa98].

The discussion on dynamic WF changes has shown that many non-trivial interdependencies exist between the different varieties of dynamic changes, which must be carefully analyzed and understood. In the *ADEPT* project we attempt to consider most of the challenges described in conjunction with each other. We provide a proper framework with a clear semantics, which also allows arguing on the correctness of dynamic WF changes. In addition, we have been working on issues like the transactional support of WF changes, the control of concurrent changes, the support of temporal constraints, and security. Human-machine-interaction is also a major issue in this context; users must be able to understand the consequences of a change they are going to perform, and they should also be able to understand why the system is refusing to perform a certain change request. The work on large-scale aspects as well as on supporting dynamically evolving WFs is on its way. During the last years, we have implemented several dedicated prototypes to study implementation and usability aspects of some of these features. Recently we have finished the implementation of the first version of the *ADEPT*-WfMS, which comprises many of the features addressed within one system. The description of the system architecture and the discussion of implementation issues will be the subject of other papers.

We are convinced that the approach we have taken in the *ADEPT* project will allow supporting the clinical as well as many other application domains in an adequate way.

References

[BFG93] Bandinelli, S.; Fuggetta, A.; Ghezzi, C.: *Software Process Model Evolution in the SPADE Environment*. IEEE Transactions on Software Engineering, 19(12):1128-1144, 1993.

[BPS97] Bichler, P.; Preuner, G.; Schrefl, M.: *Workflow Transparency*. Proc. 9th Int'l Conf. on Advanced Information Systems Engineering (CAiSE 97), pp. 423-436, Barcelona, 1997

[Casa96] Casati, F.; Ceri, S.; Pernici, B.; Pozzi, G.: *Workflow Evolution*. Proc. 15th Int'l Conf. on Conceptual Modeling, pp. 438-455, Cottbus, Germany, 1996.

[DaKl98] Dadam, P., Klas, W.: *The Database and Information System Research Group at the University of Ulm*. SIGMOD Record, 26(4):75-79, 1997.

[DMP97] Dellen, B.; Maurer, F.; Pews, G.: *Knowledge Based Techniques to Increase the Flexibility of Workflow Management*. Data & Knowledge Engineering, 1997.

[EKR95] Ellis, C.A.; Keddara, K.; Rozenberg, G.: *Dynamic Change Within Workflow Systems*. Proc. COOCS'95, pp. 10-21, Milpitas, CA, 1995.

[Hei96] Heimann, P. et al.: *DYNAMITE: Dynamic Task Nets for Software Process Management.* Proc. 18th Int. Conf. Software Engin., pp. 331-341, Berlin, 1996,

[LeRo97] Leymann, F.; Roller, D.: *Workflow-based Applications*. IBM Systems Journal, 36(1):102-123, 1997.

[ReDa98] Reichert, M.; Dadam, P.: *ADEPT$_{flex}$ - Supporting Dynamic Changes of Workflows Without Loosing Control*. Journal of Intelligent Information Systems, Special Issue on Workflow and Process Management. 10 (2), March 1998 (to appear)

[Shet96] Sheth, A.; Georgakopoulos, D.; Joosten, S.; Rusinkiewicz, M. et al.: *Report from the NSF Workshop on Workflow and Process Automation in Information Systems*, SIGMOD Record, 25 (4):55-67, 1996.

[ShKo97] Sheth, A.; Kochut, K.: *Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems*. Proc. NATO Adv. Study Institute on Workflow Management Systems and Interoperability. Istanbul, Turkey, 1997.

[Sieb96] Siebert, R.: *Adaptive Workflow for the German Public Administration*. Proc. 1st Int. Conf. on Practical Aspects of Knowledge Management, Workshop on Adaptive Workflow, Basel, Switzerland, 1996

[WäKl96] Wäsch, J.; Klas, W.: *History Merging as a Mechanism for Concurrency Control in Cooperative Environments*. Proc. RIDE-NDS'96, New Orleans, pp. 76-85, 1996.

[Wes98] Weske, M.: *Flexible Modeling and Execution of Workflow Activities*. Proc. 31st Hawai'i Int'l Conf. on System Sciences, Software Technology Track (Vol VII), pp. 713-722, 1998