Available online at www.sciencedirect.com

SCIENCE DIRECT®

Data & Knowledge Engineering 51 (2004) 223–256

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

AgentWork: a workflow system supporting rule-based workflow adaptation

Robert Müller, Ulrike Greiner *, Erhard Rahm

Department of Computer Science, University of Leipzig, Augustusplatz 10/11, Leipzig 04109, Germany

Received 22 March 2003; received in revised form 12 August 2003; accepted 31 March 2004
Available online 25 May 2004

Abstract

Current workflow management systems still lack support for dynamic and automatic workflow adaptations. However, this functionality is a major requirement for next–generation workflow systems to provide sufficient flexibility to cope with unexpected failure events. We present the concepts and implementation of AgentWork, a workflow management system supporting automated workflow adaptations in a comprehensive way. A rule-based approach is followed to specify exceptions and necessary workflow adaptations. AgentWork uses temporal estimates to determine which remaining parts of running workflows are affected by an exception and is able to predictively perform suitable adaptations. This helps to ensure that necessary adaptations are performed in time with minimal user interaction which is especially valuable in complex applications such as for medical treatments.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Workflow management; Adaptive systems; Active rules; Temporal logics; Agents

1. Introduction

Workflow management is widely adopted as a core technology to support long-term application processes in heterogeneous and distributed environments [2,17,21]. Main characteristics include the clear separation of application program code from the overall process logic and the integration of automated and manual activities. Workflow technology is increasingly used to

[*] Corresponding author. Tel.: +49-341-97-32241; fax: +49-341-97-32209.
E-mail address: greiner@informatik.uni-leipzig.de (U. Greiner).

0169-023X/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2004.03.010

manage complex processes in Internet-based e-commerce, virtual enterprises, or medical institutions [33,49]. For example, due to precisely specified treatment procedures in many medical disciplines, workflow management systems can be used to implement diagnostic and therapeutic processes [13,40,49]. Major goals include the improved and timely treatment of patients and a significant workload reduction for the hospital personnel.

However, conventional workflow management systems do not provide sufficient flexibility to cope with the broad range of failures that may occur during workflow execution. In particular, not only *system* failures such as hardware or software crashes need to be dealt with but also *logical* failures or exceptions. These logical failures refer to application-specific exceptional events for which the control and data flow of a workflow is not adequate anymore and thus has to be adapted [54]. The automatic treatment of such logical failures is the main subject of this paper.

In the cancer chemotherapy workflow shown in Fig. 1, assume it is detected just before the administration of drug *C* that the leukocyte count (i.e., the number of white blood cells) has become critically low, so that there is the risk of a serious infection for the patient. As drug *C* is known to reduce the leukocyte count additionally as a negative side effect, the activity "Administer drug *C*" dynamically has to be removed from the workflow while the execution of the other activities can be continued without change. To protect the patient from an infection, it may also be necessary to dynamically *add* an activity supporting the administration of an antibiotic drug after the cancer chemotherapy. Note that explicit conditional routing paths in the workflow definition are not sufficient to deal with such exceptions. For example, checking the condition "leukocyte count < 1000" before the "Administer drug *C*" activity would not help if this condition is violated at different points in time possibly requiring different actions (e.g., dropping drug *A* instead of drug *C*). Inserting conditional branches at any potentially relevant position would significantly reduce workflow readability and maintainability. Thus a more flexible exception handling is required to decide on how to best react to logical failures.

Previous work on dynamic workflow adaptation mostly focused on a *manual* approach where the administrator or an authorized user has to decide which events constitute logical failures and which adaptations have to be performed [45]. However, the manual approach can be time-consuming and error-prone thereby threatening the goals to be achieved with workflow management. For example, during a therapy such as the one shown in Fig. 1, a physician is usually faced with up to 20 patients and 10–30 findings per patient every day. With a manual failure handling, the physician always would have to keep in mind which findings may induce which adaptations, or at

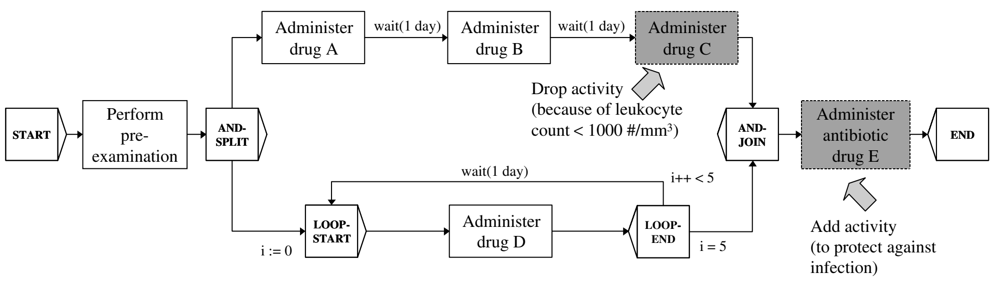


Fig. 1. Workflow adaptation example.

least would have to look it up in text books in a time-consuming manner. Hence, events constituting logical failures may be overseen or detected too late.

Recent approaches supporting *automated* workflow adaptation typically limit adaptations to the currently executed workflow activities [7,9]. Such an approach is only of limited usefulness as all workflow parts not yet reached by the control flow are not adapted automatically. This may also lead to situations where necessary adaptations are performed too late so that significant problems can occur. For example, adding a new drug administration in a cancer therapy typically requires ordering the necessary drugs one or two days before the scheduled administration to prepare a patient-specific infusion. Thus, in order to allow a timely drug administration the corresponding workflow adaptation should be performed as soon as possible. Similarly, the dropping of a cancer drug (such as drug *C* in Fig. 1) should not be performed in a "last minute" manner but in advance to avoid that a very expensive drug infusion has to be poured away. Of course, early scheduling of new activities and avoiding the unnecessary execution of originally planned activities are of great importance in many workflow application domains, e.g., for product delivery in supply chain management and writing reviews in evaluation processes.

To overcome the limitations of existing systems and comprehensively support automated workflow adaptations, we designed and developed the workflow management prototype AgentWork. It is the first system we know of that can predictively adapt the yet unexecuted parts of running workflows in a largely automated manner. The implementation of such a capability poses many challenges, in particular support for a temporal model in the specification and treatment of logical failures. This paper gives an overview of AgentWork and its underlying concepts. The contributions of our work are as follows:

- We support two strategies for automatic workflow adaptation called *reactive* and *predictive adaptation*. *Predictive adaptation* adapts workflow parts affected by a logical failure in advance (predictively) based on temporal estimates of the affected workflow activities. The adaptation typically takes place as soon as the failure is detected thereby often providing enough time to meet organizational constraints for adapted workflow parts. *Reactive adaptation* is performed when predictive adaptation is not possible. In this case, adaptation is performed when the affected workflow part is to be executed. In particular, before an activity is executed it is checked whether it is subject to a workflow adaptation such as dropping, postponement or replacement. We provide mechanisms to decide whether reactive or predictive adaptation is more suitable for a particular failure situation.
- We provide an ECA (Event/Condition/Action) rule model to automatically detect logical failures and to determine the necessary workflow adaptations. To support predictive workflow adaptations, we use a temporal object-oriented logic that allows us to specify the valid time interval for which an adaptation has to be performed. Furthermore, our approach supports the integrity of ECA rule sets.
- We provide workflow estimation algorithms to determine which workflow part is affected by a logical failure and needs to be adapted.
- We support a comprehensive set of operators for automatic workflow adaptation, including control flow operators which for example allow us to add or delete workflow activities. Furthermore, data flow operators are provided that adapt the data flow after a control flow adaptation, if necessary.

- Finally, we provide mechanisms to monitor adapted workflows by checking whether the used time estimates are met when the adapted workflow is continued.

As a first application area, AgentWork supports workflows for cancer treatment in an interdisciplinary medical project at the University of Leipzig [39,40]. Though important conceptual decisions are motivated by this medical workflow application, AgentWork has been designed to be usable in other workflow application domains as well (such as insurance business or banking). In particular, the basic AgentWork model only assumes generic events and workflow activities. By sub-classing, these generic events and activities can be refined in a domain-specific manner (e.g., for a business domain) without affecting the workflow adaptation model.

The paper is organized as follows. In the next section, we give an overview of the AgentWork system. Section 3 describes our ECA rule model. Section 4 presents the approaches for selecting the adaptation strategy, workflow duration estimation, control and data flow adaptation, and workflow monitoring. Finally, we discuss related work (Section 5), and summarize and sketch future work (Section 6).

2. AgentWork overview

In this section, we first sketch the architecture of the AgentWork system. Then, we outline the main model components (e.g., rules and workflows) and their principal interactions.

2.1. Architecture

Fig. 2 shows the three architectural layers of AgentWork:

The *workflow definition and execution layer* provides components for the definition and execution of workflows. A workflow editor and a workflow engine form its main components. In contrast to most other workflow management systems, the AgentWork engine supports the suspension or adaptation of currently executed workflows.

The *adaptation layer* implements the main concepts of AgentWork and provides three agents for the handling of logical failures. The components of this layer are called *agents* because they have several properties which are associated with agent-oriented modeling and programming, such as "*intelligence*", *autonomy*, and *cooperation* [28].

- The *event monitoring agent* decides which events constitute relevant logical failures. It uses ECA rules specifying under which condition an event induces that a workflow becomes logically inadequate, and which adaptation operations have to be performed on a workflow to cope with this event (Section 3).
- The *adaptation agent* performs the adaptation. In particular, it decides which adaptation strategy (reactive or predictive) is suitable, and applies the necessary control flow adaptations to the workflow. If necessary, it adjusts the data flow as well. In case of predictive adaptation, it performs a workflow estimation. This estimation determines which workflow part will be executed during the temporal interval for which adaptation operations have to be performed. All adaptations are subject to a manual confirmation (Section 4).

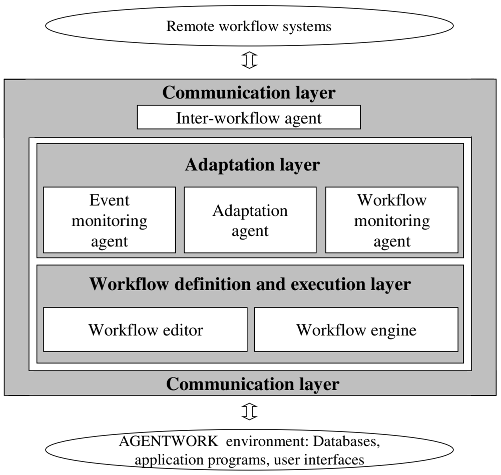


Fig. 2. AgentWork overview.

- The *workflow monitoring agent* checks whether the assumptions of the adaptation agent are met when the adapted workflow is continued. In particular, it checks whether the estimated execution durations are met by the execution reality. If this is not the case, it induces a correction of the estimation and a readaptation of the workflow (Section 4).

The *communication layer* manages the communication between AgentWork components and the environment, including remote workflow systems. It is based on the middleware Corba [4] and uses an XML message format. Its *inter-workflow agent* determines whether a logical failure occurring to a workflow has any implications for other workflows cooperating with this workflow, and informs affected workflows. As we have already addressed such inter-workflow aspects in [41], we do not further consider this agent here.

2.2. Model overview

Fig. 3 shows the main model components of AgentWork. Workflow definitions and specifications of ECA rules are based on a shared common metadata schema. This metadata schema consists of a class hierarchy for cases, events, activities, and resources. A *Case* object represents a person or institution for which an enterprise or organization provides its services. For example, if patient John Miller is admitted to a hospital, he is represented by such a *Case* object. Objects of class *Event* represent anything that may lead to logical workflow failures, such as the new leukocyte laboratory value in the example of Fig. 1. The *Activity* class is used to represent activities (e.g., a drug infusion) that are executed in workflows for cases. Activities are performed by *Resource* "objects", such as doctors, clerks, application programs, or devices.

In AgentWork, we use a *graph-oriented* workflow definition model. Within a workflow definition, activities are represented by *activity nodes*. An activity node has an associated *activity definition* to specify the details of the activity (e.g., the dosage of a drug administration). An activity definition is based on the *Activity* class of the metadata schema. The details of our logic-based

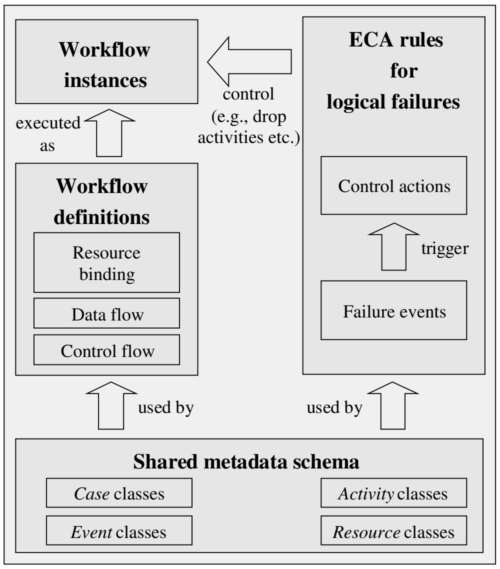


Fig. 3. Workflows and ECA rules.

activity definition approach are described in Section 3, where the particular logic used by AGENTWORK is introduced.

The control flow is specified by *edges* and *control nodes*. AGENTWORK provides control node types for conditional branching (node types OR-SPLIT/OR-JOIN), for parallel execution (AND-SPLIT/AND-JOIN), and loops (LOOP-START/LOOP-END). We use *symmetrical blocks* for control flow definition, i.e., for every split node or LOOP-START node there must be exactly one closing join node resp. LOOP-END node (Fig. 4). These symmetrical blocks may be arbitrarily nested. This principle of symmetrical blocks, which supports readability and facilitates temporal estimations, is known from structured programming and has recently also been applied to workflow management [31,45]. We additionally support so-called *synchronization edges* [45] to synchronize nodes belonging to parallel control flow paths of a block. In Fig. 4, the synchronization edge between X and Y means that Y must not be started before X has been processed successfully, unless X cannot be reached anymore by the control flow (e.g., because the condition of the OR-SPLIT path to which X belongs has been evaluated to false). We do not allow that a synchronization edge has its source node within a LOOP-START/LOOP-END block and its target node outside this block, or vice versa. This is because it then would be unclear for which loop iteration the synchronization shall take place.

The data flow is represented by data flow edges. *Internal* data flow edges specify the data flow between nodes within one workflow. *External* data flow edges specify the data flow between activity nodes and external data sources such as databases or user interfaces. Based on [45], AGENTWORK supports several dataflow correctness constraints. For example, it is controlled that for every input object of an activity there is an associated internal or external data flow edge to provide the data.

As usual, the term *workflow instance* (or simply workflow) refers to an instantiation of a workflow definition executed by the workflow engine to process cases. For simplicity, we assume

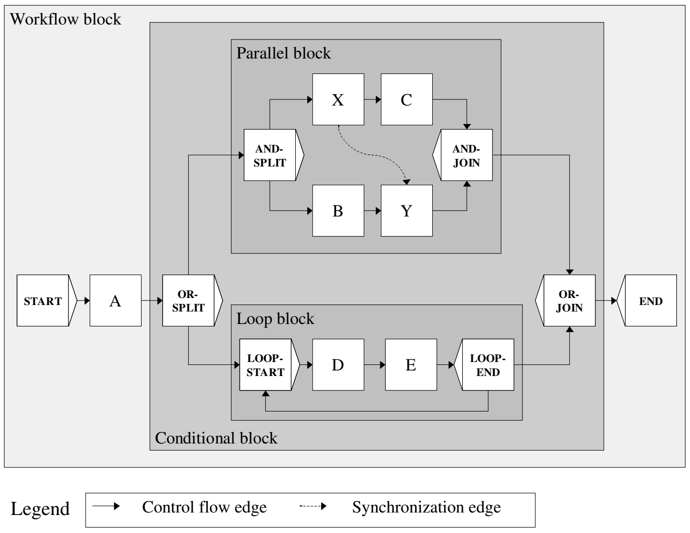


Fig. 4. Symmetrical control flow blocks and synchronization edges.

in the following that a workflow runs for exactly one case (e.g., one patient or customer) and that at most one workflow is executed for a case at a given point in time. Other possibilities, such as that one workflow runs for different cases during its life span, can be mapped to this 1:1 relationship between cases and workflows [39].

Finally, AgentWork uses ECA rules [44] to specify which events constitute logical failures and how to deal with them. For the latter, ECA rules state which *control actions* have to be performed for workflow adaptation, i.e., it is specified which activities have to be dropped, added, replaced etc. (as illustrated in Fig. 1). Such a rule-based approach is highly flexible as rules are able to react on events at any time during workflow execution without making assumptions about when these events occur. This is in contrast to an approach based on adding *conditional branches* to a workflow definition to test for logical failure events. These conditional branches would have to be inserted at many places and reduce workflow readability and maintainability significantly. For the same reasons, exception handling approaches from the field of programming languages, such as Java's *try* & *catch* blocks, cannot be used. This is because they require that the relative point in time of the failure event occurrence w.r.t. a particular position in the program (i.e., the workflow definition) is known at definition time. However, this is not possible for most types of failure events.

3. Temporal ECA rule model

In this section, we first sketch the principal structure of our ECA rules (3.1). Then, we introduce the temporal logic ActiveTFL to specify on a formal level our ECA rules and the workflow activities they refer to (3.2). Finally, we sketch rule integrity aspects (3.3). For simplicity, we concentrate on events occurring to cases (e.g., patients). Logical failures concerning workflow *resources*, such as a broken computer tomography device making it temporarily impossible to execute some activities, can be treated analogously [39]. Furthermore, in the examples we omit application-specific details such as the units of laboratory values and drug dosages.

3.1. Structure of ECA rules

In AgentWork, ECA rules have the following basic structure:

WHEN	*event*
WITH	*condition*
THEN	*control action*
VALID-TIME	*time period*

The *event-condition* (*WHEN*/*WITH*) part specifies which *event* constitutes a failure event under which *condition*. The *action* (*THEN*) part declaratively states on a high level of abstraction which control action has to be performed on a workflow to cope with the event, e.g., which activities may have to be dropped or added. In particular, a control action does not make any assumptions about how the activities are spread over different workflow definitions. This has the advantage that reorganizing activities within workflows has no or only minimal effects on ECA failure rules. For example, if a drug administration node is placed at a different location within a workflow definition, ECA rules dealing with this drug administration do not have to be changed (of course, more comprehensive workflow definition changes may require the changing of ECA rules, too). The *VALID-TIME* clause of the control action specifies the time period during which the control action is valid, i.e., during which the respective adaptation needs to be applied. An (informal) sample ECA rule is:

$$\begin{array}{ll} \textit{WHEN} & \textit{new finding of patient } P \\ \textit{WITH} & \textit{leukocyte count} < 1000 \\ \textit{THEN} & \textit{drop drug Etoposid for } P \\ \textit{VALID-TIME} & \textit{during the next seven days} \end{array} \tag{1}$$

If two rules refer to the same event, the union of the triggered control actions is valid.

3.2. ActiveTFL

To specify our adaptation model and in particular our ECA rules formally, we use a logic. In particular, the well-defined declarative and unambiguous semantics and proof theory of logics are suitable for *automating* workflow adaptation. As existing logics such as First-Order Logic [11], Frame Logic [30], or Description Logic [19] either do not provide sufficient data specification capacities (e.g., First-Order Logic) or temporal support (e.g., Frame Logic), we designed the logic ActiveTFL (Active Temporal Frame Logic). Basically, ActiveTFL combines a powerful object-oriented logic (namely Frame Logic) with elements from temporal logics and active rules known from active databases (Fig. 5). [1]

[1] We have not selected Description Logic, as this logic focuses on terminological reasoning and natural language processing [19] which is not relevant for logical failure handling.

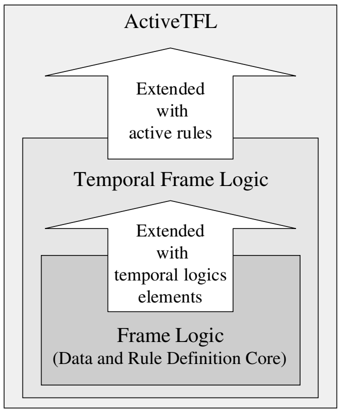

ActiveTFL
Extended with active rules
Temporal Frame Logic
Extended with temporal logics elements
Frame Logic
(Data and Rule Definition Core)

Fig. 5. Structure of ActiveTFL.

3.2.1. Frame Logic

In the following, we introduce the relevant Frame Logic (FL) components by means of medical examples. This includes FL classes, objects, object extensions, object patterns, predicates, formulas, and rules. [2]

FL *class definitions* have the general form

<class-name>[*<attribute-name$_1$>*: *<attribute-type$_1$>*, *<attribute-name$_2$>*:*<attribute-type$_2$>*,. . .].

The main classes in AgentWork used for the representation of cases (e.g., patients), events and activities are as follows:

Case [*case-id*: *Integer*, *name*: *String*, *events*: *Set<Event>*, *activities*: *Set<Activity>*]
Event[*date: Date, time: Clock-Time, of: Case*]
Blood-Finding[*parameter: Enum* { *Leukocyte-Count,. . .*}*, value: Float*]
Activity[*date: Date, time: Clock-Time, activity-for: Case*]
Drug-Administration[*drug: String, dosage: Float*]
Patient[*social-num: Integer, diagnosis: String*]
Physician[*name: String,. . .,degree: Enum*{*Senior, Assistant,. . .*}, *speciality: String, patients: Set<Patient>*].
Patient IS-A Case Blood-Finding IS-A Event Drug-Administration IS-A Activity

"IS-A" denotes the subclass relationship. For example, class *Drug-Administration* defines two attributes *drug* and *dosage* of type *String* resp. *Float* and is a subclass of class *Activity*. The latter class defines when (*date* and *time*) and for whom (*Case* object *activity-for*) the drug is administered.

[2] For better readability, we have adapted the FL syntax. Nevertheless, the language model is that of FL as described in [30].

FL *objects* (i.e., class instances) are denoted as follows:

$$d : \textit{Drug-Administration}[\textit{date} = 7/2/04, \textit{time} = 9.00\ \textit{am}, \textit{activity-for} = \textit{bob},$$
$$\textit{drug} = \text{“}\textit{Etoposid}\text{”}, \textit{dosage} = 100]$$

(with *bob* denoting some *Case* instance). The symbol “denotes” the *is-object-of* relationship (e.g., *d* is an object of class *Drug-Administration*).

For storage purposes, objects can be collected persistently in so-called *object extensions*. Such an extension has the structure

$$\textit{Extension} < \textit{extension-name} > [< \textit{class-of-extension-objects} >].$$

For example

$$\textit{extension patients}(\textit{Patient}) \tag{2}$$

defines an extension of *Patient* objects called *patients*. The mapping between these object extensions and the physical data sources is the task of the communication layer (Fig. 2).

An FL *object pattern* constrains the structure of an object. It has the form

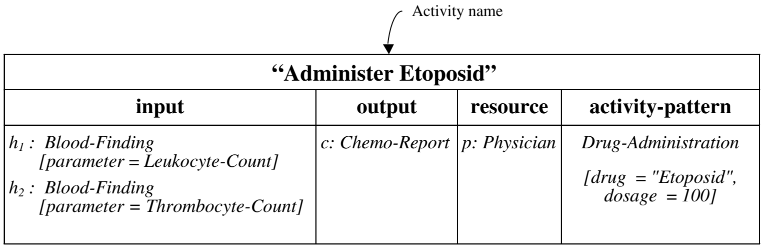
$$\textit{Class}[\textit{constraints}]$$

with *Class* being an FL class and *constraints* being a set of constraints w.r.t. the attributes of *Class* objects. For example,

$$\textit{Drug-Administration}[\textit{drug} = \text{“}\textit{Etoposid}\text{”}, \textit{dosage} > 50] \tag{3}$$

specifies the pattern of *Drug-Administration* objects representing Etoposid dosages higher than 50. The *type* of an object pattern is denoted with *Obj-Patt<Class>*, e.g., *Obj-Patt<Drug-Administration>* for our drug administration example. Patterns of type *Obj-Patt<Activity>* are called *activity patterns*.

In AgentWork, object and activity patterns are used to specify the details of workflow activities and to constrain the input and output objects needed or produced by activities. For example, the activity pattern shown in Fig. 6 specifies that the drug Etoposid has to be administered as an infusion with a dosage of 100 (the *date/time/activity-for* attributes are left unspecified as their values cannot be determined before workflow execution time). Furthermore, it is specified that two *Blood-Finding* objects, h_1 and h_2, are expected as input representing the leukocyte resp. thrombocyte count. Furthermore, it is specified that a physician is needed as a resource to perform this activity, and that a *Chemo-Report* object is produced as output. As a


Activity name

“Administer Etoposid”			
input	**output**	**resource**	**activity-pattern**
h_1 : Blood-Finding [parameter = Leukocyte-Count] *h_2 : Blood-Finding [parameter = Thrombocyte-Count]*	*c: Chemo-Report*	*p: Physician*	*Drug-Administration [drug = "Etoposid", dosage = 100]*

Fig. 6. Activity definition example.

shorthand, we use the terms *A-activity* and *A-node* to denote an activity resp. activity node based on an activity pattern *A*.

Furthermore, *predicates* can be defined in FL to express properties that hold for some objects. In AgentWork, predicates are primarily used to express control actions such as the *drop* control action in Fig. 1, e.g., we can define the predicate

$$drop(A, CS) \tag{4}$$

with *A* being of type *Obj-Patt*(*Activity*) and *CS* being an object of class *Case* to state that any activity executed for case *CS* and matching pattern *A* has to be dropped.

Analogously to first-order logic [11], FL *formulas* can be constructed inductively on base of FL objects, predicates, Boolean operators, and quantifiers [30].

Rules in FL are used to express which formulas imply other formulas. For example, if *A* is the activity pattern *Drug-Administration*[*drug* = *"*Etoposid*"*], then the rule

$$\begin{array}{ll} \mathit{WHEN} & \mathit{critical\text{-}blood\text{-}status}(P) \\ \mathit{THEN} & \mathit{drop}(A, P) \end{array} \tag{5}$$

states that whenever a patient *P* has a critical blood status (e.g., leukocyte count < 1000)—expressed by some predicate *critical-blood-status*(*P*)—*that then* Etoposid *has to be dropped for P*. Note that such a rule is not yet an ECA rule as it has no notion of "data events" such as inserting blood data into an extension. This will be described in 3.2.3, where we introduce our notion of active rules.

3.2.2. Temporal FL

So far, an FL rule such as (5) does not specify the *valid time* of the derived control action, i.e., for how long the activity specified by *A* shall be dropped for patient *P*. To restrict the validity of a statement to some period of time, ActiveTFL supports so-called *temporal frames* and *temporal formulas* allowing us to assign a valid time to a formula.

A *temporal frame* $(T, <)$ consists of a non-empty discrete set *T* of "points in time" (i.e., the "time axis"), ordered by a non-reflexive binary relation < of precedence ("earlier than") [6]. A frequently used temporal frame is the set of points in time of the Gregorian calendar.

On base of a temporal frame $(T, <)$, valid times can be assigned to formulas. We support two principal types, *fixed* and *conditional* valid time, covering a broad range of time periods considered sufficient for most application areas.

Fixed valid time. A fixed valid time is any set $S \subset T$ which is described by an explicit listing of points in time or by temporal functions. For example, [2 March 2004:8 pm, 2 March 2004: 8pm+(72,hour)] specifies the set of points in time starting at 2 March 2004: 8 pm and ending after 72 h (i.e., at 5 March 2004: 8 pm). Expressions of the structure (*amount,time-unit*) specify an amount of time, e.g., (*72,hour*) for 72 h. The interval [*now*, *now*+ (*72, hour*)] specifies the set of points in time starting at the current system time *now* (rounded to the closest point in time of *T*) and ending after 72 h.

Such a fixed valid time *S* then can be assigned to any FL formula via the *VALID-TIME* statement, i.e.,

$$F \ \mathit{VALID\text{-}TIME} \ S$$

states that F holds at every $t \in S$. An example for a rule with such a *VALID-TIME* statement is

$$\begin{array}{lll} \mathit{WHEN} & \mathit{critical\text{-}blood\text{-}status}(P) & \mathit{VALID\text{-}TIME}[\mathit{now} - (5, \mathit{day}), \mathit{now}] \\ \mathit{THEN} & \mathit{drop}(A, P) & \mathit{VALID\text{-}TIME}[\mathit{now}, \mathit{now} + (7, \mathit{day})]. \end{array} \tag{6}$$

This rule states that whenever the predicate *critical-blood-status*(P) has been valid during the last five days, that then $drop(A,P)$ is valid for the next seven days.

Conditional valid time. To describe a valid time conditionally by a termination condition, we use the temporal operators *Until* and *Unless* [12,36]. With these operators, it can be stated how the valid time of an FL formula is related to the valid time of another formula. In the following, F and G are FL formulas while t, t', t'' denote points in time.

- *Until*. This operator is used to express that a formula G eventually will be valid in the future and that a formula F is valid at least until G (first) becomes valid, i.e.,

It holds:	iff	it holds:
(F *Until* G)		It exists $t' > t$ with:
VALID-TIME t		
		G *VALID-TIME* t' and for all t'' with
		t, $t'' < t'$ it holds:
		F *VALID-TIME* t'' *AND NOT*
		(G *VALID-TIME* t'')

  A typical medical example for the *Until* operator is the rule

$$\begin{array}{ll} \mathit{WHEN} & \mathit{critical\text{-}blood\text{-}status}(P)\ \mathit{VALID\text{-}TIME}[\mathit{now} - (3, \mathit{day}), \mathit{now}]\ \mathit{AND} \\ & \mathit{present\text{-}in\text{-}further\text{-}workflow}(\mathit{Drug\text{-}Administration}[\mathit{drug} = \text{“}\mathit{Etoposid}\text{”}], P) \\ \mathit{THEN} & \mathit{add} - \mathit{repetitively}(\mathit{Drug\text{-}Administration}[\mathit{drug} = \text{“}\mathit{Doxycyclin}\text{”}], (1, \mathit{day}), P) \\ & \mathit{Until}\ \mathit{drop}(\mathit{Drug\text{-}Administration}[\mathit{drug} = \text{“}\mathit{Etoposid}\text{”}], P) \\ & \mathit{VALID\text{-}TIME}\ \mathit{now} \end{array}$$

  This rule is triggered whenever a patient P has had a critical blood status during the last three days and is receiving the drug ETOPOSID during further workflow execution (the latter expressed by the predicate *present-in-further-workflow*). Then, P must get the drug DOXYCYCLIN repetitively every day until the drug ETOPOSID is dropped. Note that this rule does not specify at which workflow position a new *Drug-Administration* node shall be inserted. This is because this position depends on when the exception occurs which must be determined at runtime (see 4.3). The *Until* line states that the *add-repetitively* predicate holds until drug ETOPOSID is dropped, i.e., the *drop* predicate is here used as a condition (not as an action).
- *Unless* (*Waiting-for*). As F *Until* G by definition requires that G will eventually occur, sometimes weaker statements are needed stating that F is valid either until G becomes valid, or is valid forever in case that G will never become valid in the future. This is done by the *Unless* operator which is defined as

It holds:	iff	it holds
$(F\ Unless\ G)\ VALID\text{-}TIME\ t$		$(F\ Until\ G)\ VALID\text{-}TIME\ \ t\ OR$
(at point in time t,		$(F\ VALID\text{-}TIME\,[t,\infty)$
F is valid unless G is valid)		

With *Unless* we can express statements such as that Etoposid has to be dropped when a patient has had a critical blood status for the last five days, and that Etoposid can only be given again when the blood status becomes normal again (leukocyte count >1000):

$$
\begin{aligned}
&WHEN\ \ critical\text{-}blood\text{-}status(P)\,VALID\text{-}TIME[now-(5,day),now]\ AND\\
&\qquad\quad present\text{-}in\text{-}further\text{-}workflow(Drug\text{-}Administration[drug=\text{“}Etoposid\text{”}],P)\\
&THEN\ \ drop(Drug\text{-}Administration[drug=\text{“}Etoposid\text{”}],P)\\
&\qquad\quad Unless\ normal\text{-}blood\text{-}status(P)\\
&\qquad\quad VALID\text{-}TIME\ now
\end{aligned}
$$

Note that in contrast to fixed valid times, the duration of such a conditional valid time typically is not known beforehand. This difference between the two valid time types will be of particular importance for the adaptation strategy as we will see in Section 4.

3.2.3. ActiveTFL

We now describe the principals of ActiveTFL, which extends Temporal FL with the notion of primitive and composite events, actions, and active rules. Basically, ActiveTFL is a data-driven (or forward chaining) rule model. This means, that—similar to triggers in relational databases—whenever an event occurs (e.g., when data is inserted into a database) it is checked whether ActiveTFL rules qualify for execution as their *WHEN*/*WITH* part evaluates to true.

A *primitive event* is the occurrence of a basic operation on an object extension. ActiveTFL supports the primitive event types INSERT, REMOVE, and UPDATE corresponding to the respective operations on extensions. In our context, especially the insertion, removing or updating of an *Event* object is important, as such an *Event* object can trigger a logical failure.

To filter relevant events, a condition can be assigned to a primitive event in the *WITH* part of a rule. This condition may consist of any temporal FL formula f on the object referenced in the *WHEN* part. The symbols *new* and *old* refer to the new resp. old object after the INSERT, UPDATE, or REMOVE operation. An example is

$$
\begin{aligned}
&WHEN\ INSERT\ ON\ blood\text{-}findings\\
&WITH\ new.parameter = Leukocyte\text{-}Count\ AND\ new.value < 1000
\end{aligned}
\tag{7}
$$

with *blood-findings*(*Blood-Finding*) being an extension of *Blood-Finding* objects. This primitive event is triggered whenever a new *Blood-Finding* object is inserted in the extension *blood-findings*, for which the measured parameter is a leukocyte count less than 1000.

Composite events can be constructed from already defined events. ActiveTFL supports the composite event types *conjunction*, *disjunction*, *negation*, and *time series* [8,38]. As the definition of conjunctions, disjunctions, and negations is straightforward, we only describe *time series* which are of particular importance especially for medical domains (e.g., temporal course of laboratory findings) or business domains (e.g., stock exchanges). For example, a single critical finding such as a low leukocyte value does not necessarily induce a logical failure but often only the *repetitive* occurrence of critical findings.

Given some (primitive) event E, we say that a time series event over E with length n, minimal and maximal temporal distances $d_{\min}$ and $d_{\max}$ occurs during the temporal interval I, if E occurs repetitively during I at a sequence of n points in time with a minimal distance of $d_{\min}$ and a maximal distance of $d_{\max}$ between two successive points of the sequence, i.e.

$$\begin{array}{lll} \mathit{TIME\text{-}SERIES}(E, n, d_{\min}, d_{\max}, I) \ \mathit{occurs} & \text{iff} & \text{it exist } t_1 < t_2 < \cdots < t_n, t_i \in I \text{ with:} \\ & & d_{\min} \leqslant |t_i - t_{i-1}| \leqslant d_{\max},\ i = 2, \ldots, n \\ & & E \text{ occurs at every } t_i. \end{array}$$

The point in time at which an instance of $\mathit{TIME\text{-}SERIES}(E, n, d_{\min}, d_{\max}, I)$ occurs is t_n (as then the last instance of E establishing the time series occurred).

A typical medical example for a time series event is the following: Let E be the (primitive) event that the leukocyte count of a patient is less than 1000 as defined in (7). Then,

$$\mathit{WHEN}\ \ \mathit{TIME\text{-}SERIES}(E, 3, (2, \mathit{day}), (4, \mathit{day}), [\mathit{now}, \mathit{now} + (2, \mathit{week})]) \tag{8}$$

occurs if during two weeks the leukocyte count of a patient is less than 1000 at three points in time with a minimal distance of 2 days and a maximal distance of 4 days between two leukocyte measurements. The possibility to specify $d_{\min}$ respectively, $d_{\max}$ is helpful as often occurrences of E being too close together or too far away from each other have a limited significance. For example, two leukocyte count measurements at two subsequent days do not mean more information than one measurement, as the leukocyte value usually does not change significantly during two days.

In ActiveTFL, the *action* part of a rule consists of a single control action in order to reduce language complexity and to facilitate the handling of control flow failures. Still, the AgentWork editor allows to define rules with n AND-connected control actions in the *THEN* part ($n = 1, 2, 3, \ldots$). Such a rule is then internally translated into n rules with one control action each and the same *WHEN*/*WITH* part. [3] For example, the conjunction of two control actions is translated into two rules, which are both triggered by the same event and where each rule triggers one of the two actions.

Two main types of control actions are supported, namely *global* and *local* control actions:

Global control actions state that a workflow is not adequate anymore *as a whole*. We support the global control actions *abort* and *suspend* for the entire abortion resp. suspension of a workflow. In the latter case a valid time statement assigned to the suspension control action has to specify for how long the workflow shall be suspended.

Local control actions state that only *some* activities of a workflow are not adequate anymore. Thus, the workflow can be continued but has to be adapted locally. AgentWork supports the following local control actions which are motivated by the fact that activity nodes cover the main semantics of a workflow, and that nodes can either be dropped, replaced, added, or postponed

[3] Note that due to Horn clause theory, we have to forbid the disjunction of control actions to keep rules satisfiable.

(A and A' denote activity patterns, and CS denotes a case). The question which control action pairs are compatible is discussed in Section 3.3.1.

- $drop(A, CS)$: For CS, A-activities must not be executed anymore.
- $replace(A, A', CS)$: For CS, every A-activity execution has to be replaced by an A'-activity.
- $add(A, CS)$: For CS, an A-activity has additionally to be executed exactly once.
- $add\text{-}repetitively(A, d, CS)$: Additional A-activities have to be performed repetitively for CS. The duration between two subsequent A-activity executions is specified by d.
  Both for $add(A, CS)$ and $add\text{-}repetitively(A, d, CS)$ the particular insertion position of a new A-node is determined at workflow execution time (see 4.3) and is restricted by the valid time specification of the corresponding ECA rule.
- $postpone(A, d, CS)$: For CS, every A-activity execution has to be postponed by duration d (relative to its control flow position at the point in time the control action has been triggered).
  Postponing activities may be a sufficient reaction compared to applying the global control action *suspend*. For example, for a treatment workflow it may be sufficient to postpone only the patient's drug administrations while the diagnostic activities may be continued as specified in the original workflow definition.
- $review(A, CS)$: For CS, every execution of an A-activity has to be reviewed by a user (manual control).

3.3. Rule integrity

In the following, we discuss several aspects of rule integrity, in particular rule incompatibility (3.3.1), and rule termination (3.3.2).

3.3.1. Rule incompatibility

In our context, the term *rule incompatibility* refers to the situation that two rules trigger incompatible control actions at the same point in time. For example, it has to be avoided that two rules trigger a $drop(A, CS)$ and an $add(A, CS)$ control action with the same activity pattern A for the same case CS with overlapping valid time intervals. To cope with this, AgentWork uses incompatibility tables which we explain now.

Let

$$ca_1(A_1, [B_1, d_1, p_1, f_1,]C_1)\ \mathit{VALID\text{-}TIME}\ VT_1$$

and

$$ca_2(A_2, [B_2, d_2, p_2, f_2,]C_2)\ \mathit{VALID\text{-}TIME}\ VT_2$$

denote the control actions of the *THEN* part of two rules R_1 and R_2 with overlapping event patterns (the parameters B_i, d_i, p_i and f_i are only needed for $ca_i = replace$, *postpone* or *add-repetitively*). Then R_2 is said to be incompatible with R_1 (w.r.t. $VT_1 \cap VT_2$), if the following conditions hold:

- C_1 and C_2 refer to the same case, and the valid time intervals VT_1 and VT_2 overlap.
- A_1 subsumes A_2 , i.e. performing an activity based on activity pattern A_2 means that implicitly an activity based on pattern A_1 is performed. For example, if we have

$A_1 := \textit{Drug-Administration}[\textit{drug} = \text{“E}\textsc{toposid}\text{”}, \textit{dosage} > 100, \textit{unit} = \textit{mg}]$
$A_2 := \textit{Drug-Administration}[\textit{drug} = \text{“E}\textsc{toposid}\text{”}, \textit{dosage} > 150, \textit{unit} = \textit{mg}]$
then A_1 subsumes A_2 as every Etoposid administration with more than 150 mg is also an Etoposid administration of more than 100 mg.

- The control actions ca_1 and ca_2 are incompatible according to Table 1.

For selected control action pairs in Table 1, we explain why they are viewed as incompatible or compatible (for the other pairs analogous arguments hold). In the following, for a pair (i, j), i refers to the row and j to the column of Table 1. Furthermore, to avoid an overlapping of the different cells of Table 1, we agree on the convention that for any cell *above* the grey diagonal the *equality* of the patterns A_1 and A_2 is allowed, but forbidden for any other cell (i.e., the equality case for A_1 and A_2 is covered by the cells above the grey diagonal). Note further, that the matrix of Table 1 is not symmetrical due to the subsumption relationship between A_1 and A_2 (i.e., in general A_1 and A_2 are *not* identical patterns).

- **Pair (1,3)** $\textit{drop}(A_1)$, $\textit{postpone}(A_2, d_2)$. This pair is viewed as incompatible, as on one side it is specified that A_2-activities shall be postponed, but on the other side shall be dropped due to $\textit{drop}(A_1)$ (as A_1 subsumes A_2).
- **Pair (3,1)** $\textit{postpone}(A_1, d_1)$, $\textit{drop}(A_2)$. This pair is viewed as compatible. It is specified that A_1-activities shall be postponed, but only some of them (i.e., the A_2-activities) shall be dropped. For example, let us assume

  $A_1 := \textit{Drug-Administration}[\textit{drug} = \text{“E}\textsc{topodsid}\text{”}]$
  $A_2 := \textit{Drug-Administration}[\textit{drug} = \text{“E}\textsc{toposid}\text{”}, \textit{dosage} > 150, \textit{unit} = \textit{mg}]$.

  Then, $\textit{postpone}(A_1, d_1)$ and $\textit{drop}(A_2)$ mean that the etoposid administrations principally have to be postponed but that some of them (i.e., those with a dosage higher than 150 mg) have to be dropped. From a medical point of view, such a combination makes sense and thus should not generally be forbidden. Therefore, AgentWork informs the rule modeler about such combinations but does not force the rule modeler to drop or re-edit such rules.
- **Pair (3,4)** $\textit{postpone}(A_1, d_1), \textit{add}(A_2)$. This pair is viewed as compatible, as an A_2-activity can be added to A_1-activities that shall be postponed. However, the *order* in which the two control actions shall be processed has to be determined at execution time, i.e., whether the A_2-activity shall be first added and then postponed with all A_1-activities, or whether the A_2-activity shall be added after the A_1-activities have been postponed. The order of the control action typically depends on the event constellation triggering the two control actions and thus cannot be fixed at definition time.
- **Pair (1,4)** $\textit{drop}(A_1)$, $\textit{add}(A_2)$. This pair is viewed as incompatible, as on one side it is specified that an A_2-activity shall be added, but on the other side also shall be dropped due to $\textit{drop}(A_1)$ (as A_1 subsumes A_2).
- **Pair (4,1)** $\textit{add}(A_1)$, $\textit{drop}(A_2)$. This pair is viewed as compatible as adding a more general activity (i.e., an A_1-activity) does not exclude that more specific activities are dropped from a workflow.

Table 1
Incompatibility table for control actions (with A_1 subsuming A_2)

		1	2	3	4	5
		ca$_1$				
	ca$_2$	$drop(A_2)$	$replace(A_2, B)$	$postpone(A_2, d_2)$	$add(A_2)$	$add-repetitively(A_2, d_2)$
1	$drop(A_1)$	CP	CP[a] *ICP* if $A_1 \subset B$[b]	ICP	ICP	ICP
2	$replace(A_1, B)$	CP	CP	ICP	ICP	ICP
3	$postpone(A_1, d_1)$	CP	CP	*CP* (*ICP* only for $d_1 \neq d_2$)	CP[c]	CP[c]
4	$add(A1)$	CP	CP	CP	CP	CP
5	$add-repetitively(A_1, d_1)$	CP	CP	CP	CP	CP (ICP only for $d_1 \neq d_2$)

ICP = Incompatible, CP = Compatible.

The C_i-parameters for the cases have been omitted as conflicts only occur if control actions refer to the *same* case (e.g., the same patient). The *review* control action is not listed as it has to be manually transformed to one of the other control actions.

[a] It is *not* viewed as incompatible that a subset of the A1-activities to be dropped (namely the $A2$-activities) shall be replaced by B-activities.

[b] As the new B-activities would directly have to be dropped due to $drop(A1)$.

[c] Order to be determined (manually) at execution time.

Table 2
Excerpt from autorization table for a medical application (cancer therapy)

Staff Member Pattern (Obj-Patt<Physician> according to 3.2.1)	Allowed operations
Physician[degree = Senior; speciality = "Oncology"];	MANUAL_RESOLVEMENT_OF INCOMPATIBLE_CONTROL_ACTIONS

For example, if we have

$A_1 := \mathit{Radiodiagnostic\text{-}Activity}[\mathit{focus} = \text{``}\mathit{Liver}\text{''}]$

$A_2 := \mathit{MRT\text{-}Examination}[\mathit{focus} = \text{``}\mathit{Liver}\text{''}]$ [4]

this would mean that a radiodiagnostic activity shall be added (e.g., a computer tomography examination), but that any MRT examination shall be dropped.

If incompatible control actions have been triggered simultaneously, an "authorized" user has to be informed and has to resolve the situation manually. To decide which users are authorized in a particular context an authorization table is used. An excerpt of the authorization table for our project at the University of Leipzig is shown in Table 2. In this excerpt, it is specified that any physician instance that fulfills the pattern in the topmost row (i.e., a physician being a senior oncologist) is allowed to resolve incompatible control actions manually. Such authorization tables are also used in the following to specify which staff members are allowed to interact w.r.t. to other workflow-related actions, such as the local dropping or adding of workflow activities.

Note that the sketched incompatibility situations generally cannot be detected at build time. This is because two of the main criteria stating whether two rules produce an incompatibility—i.e. the criterion that the two case variables C_1 and C_2 of the two rules refer to the same case, and the criterion that the two valid time intervals VT_1 and VT_2 of the rules overlap—can only be determined at run time.

3.3.2. Rule termination

Generally, for a rule base it should be guaranteed that for any data constellation rule processing cannot continue forever, i.e., that rules cannot activate each other indefinitely. Though our failure rules have a very restricted structure (e.g., only one control action in the *THEN* part), the problem of rule termination also has to be considered for our rule base of failure rules. This is because control actions may also appear in the *WHEN* part of a rule, as they formally are predicates and thus F-Logic formulas. Thus, cycles principally can occur. For example, if we have

$A_1 := \mathit{Drug\text{-}Administration}[\mathit{drug} = \text{``}\mathit{Etoposid}\text{''}]$ *and*

$A_2 := \mathit{Drug\text{-}Administration}[\mathit{drug} = \text{``}\mathit{Doxycyclin}\text{''}]$

there may be the rule

$\mathit{WHEN} \quad \mathit{drop}(A_1, C)\,\mathit{VALID\text{-}TIME}[\mathit{now}, \mathit{now} + (n, \mathit{day})]$

$\mathit{THEN} \quad \mathit{drop}(A_2, C)\,\mathit{VALID\text{-}TIME}[\mathit{now}, \mathit{now} + (n, \mathit{day})]$

[4] Assuming that MRT-Examination (MRT = Magnet Resonance Tomography) is a subclass of Radiodiagnostic-Activity.

stating that whenever the drug Etoposid is dropped for *n* days, the drug Doxycyclin can also be dropped for the same time. The rationale for this is that often an antibiotic drug is given in parallel to immunsupressive drugs such as Etoposid to prevent bacterial infections. Thus, when the immunsupressive drug is dropped the antibiotic drug can also be dropped. If a second rule triggered by a $drop(A_2, C)$ control action then would trigger a $drop(A_1, sC)$ control action, these two rules together can induce a cycle which should be avoided.

As in ActiveTFL the *action* part of a rule consists of only one control action and thus is rather simple, we use a straightforward static analysis approach to detect cycles [44]. A triggering graph is built where the nodes stand for rules in the rule set and the edges express which rule may trigger which other rules. If the graph has no cycles, termination of rule execution is guaranteed. If a cycle is detected, it has to be resolved manually.

4. Workflow adaptation and monitoring

We now describe how AgentWork processes triggered control actions. As the global control actions *abort* and *suspend* do not require workflow adaptations, we concentrate on *local* control actions. The technical challenges that have to be faced in this context are the following:

- First, decision criteria are needed to decide which adaptation strategy (reactive or predictive) is suitable.
- Second, based on the temporal model introduced in Section 3, mechanisms for the *estimation* of a workflow's future execution behavior have to be provided, in particular for predictive adaptation.
- Third, control actions have to be translated into structural workflow adaptations taking into account the current execution state of running workflows. Furthermore, the respective adaptations of a workflow's node and edge set have to maintain the structural consistency of the workflow. In addition, structural adaptations such as adding new activities should minimally delay workflow execution.
- Fourth, workflow monitoring is necessary for predictive adaptation to control whether the actual execution of an adapted workflow matches the temporal estimates. Deviations may require to modify the original adaptations.

The section is organized as follows: we first characterize the principal adaptation strategies (reactive or predictive) that can be performed to handle a triggered control action (4.1). We then describe workflow duration estimation for predictive adaptation (4.2). In 4.3 and 4.4, we outline control flow and data flow adaptation, in particular the use of adaptation operators to translate control actions into structural workflow changes. Section 4.5 describes workflow monitoring.

4.1. Adaptation strategies

To support automated workflow adaptations for a broad range of failure situations, AgentWork supports both reactive and predictive adaptation. AgentWork tries to use

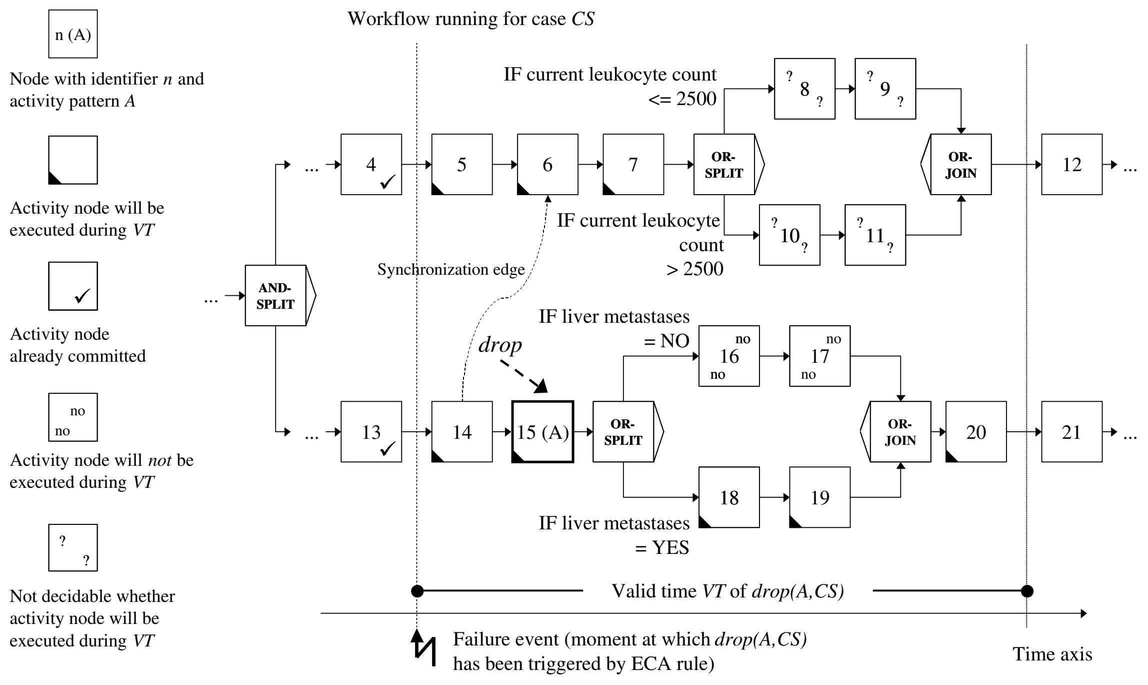

n (A)
Node with identifier *n* and activity pattern *A*
Activity node will be executed during *VT*
Activity node already committed
no no
Activity node will *not* be executed during *VT*
? ?
Not decidable whether activity node will be executed during *VT*
Workflow running for case *CS*
AND-SPLIT
OR-SPLIT
OR-JOIN
IF current leukocyte count <= 2500
IF current leukocyte count > 2500
Synchronization edge
drop
15 (A)
IF liver metastases = NO
IF liver metastases = YES
Valid time *VT* of *drop(A,CS)*
Failure event (moment at which *drop(A,CS)* has been triggered by ECA rule)
Time axis

Fig. 7. Workflow estimation and adaptation.

predictive adaptation whenever possible and to correct running workflows as soon as possible to avoid the execution of unnecessary activities, reduce delays for new activities, etc.

To illustrate the two strategies, we use the workflow example shown in Fig. 7. For this workflow running for case *CS*, two parallel paths are executed when a logical failure event occurs after nodes 4 and 13 have committed. The ECA rule for this event is assumed to trigger the control action $drop(A, CS)$ with valid-time *VT*.

Fig. 8 shows how selection of the adaptation strategy is performed. Principally, predictive adaptation can be selected if a *fixed* valid time *VT* is assigned to the control action. Then, the

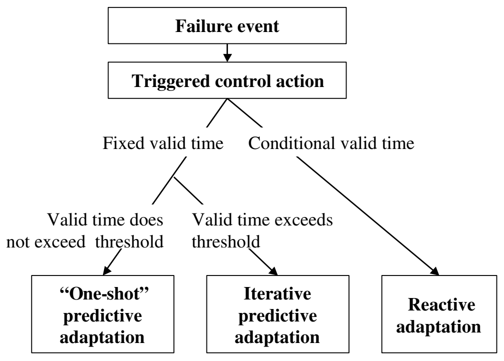

Failure event
Triggered control action
Fixed valid time
Conditional valid time
Valid time does not exceed threshold
Valid time exceeds threshold
"One-shot" predictive adaptation
Iterative predictive adaptation
Reactive adaptation

Fig. 8. Strategy selection.

temporal interval during which the control action is valid is exactly known already at the moment of the failure event. Thus, AgentWork can estimate which workflow part P_{VT} will be executed during *VT*. This is done by using temporal meta information about the duration of workflow activities (see 4.2). For example, if *VT* in Fig. 7 is a fixed valid time, AgentWork can estimate which workflow part P_{VT} starting at nodes 5 and 14 will be executed during *VT*. This would result in predictively dropping *A*-node 15 during the processing of the ECA rule and before continuing with the execution of nodes 5 and 14.

We differentiate between two subtypes of predictive adaptation depending on whether or not the fixed valid time interval exceeds some specified threshold value (Fig. 8). Normally, a "*one-shot*" predictive adaptation is applied where the workflow part corresponding to the full valid time interval is estimated at once. For long time intervals exceeding the threshold, the accuracy of the workflow estimates is likely reduced so that an *iterative* approach with multiple predictions is applied. For this purpose, the valid time *VT* is divided into several sub-intervals $VT1, VT2, \ldots, VTn$ whose durations do not exceed the threshold. Then, it is first estimated which workflow part P1 will be executed during *VT*1, and the adaptation is only applied to P1. After P1 has been executed, the procedure is continued for *VT*2 and so on. Suitable values for the time threshold depend on the application domain and the quality of workflow estimation; the threshold is thus a configuration parameter of AgentWork.

The strategy of *reactive adaptation* is selected whenever predictive adaptation is not possible. In particular, if a *conditional* valid time is assigned to a control action, it is not possible to derive which part of the remaining workflow will be executed during the corresponding valid time interval. The reactive strategy is also selected, if a *fixed* valid time has been assigned to the control action, but if an estimation is *not* possible, e.g., for some conditional parts of the workflow such as conditional OR-SPLIT or LOOP-END [5] nodes (see 4.2). For example, if *VT* in Fig. 7 would be a conditional valid time, reactive adaptation would be selected, and it would be checked for every node *n* that is reached by the control flow during *VT* whether *n* is an *A*-node. As node 15 is such an *A*-node, it would be dropped after node 14 has committed.

For both predictive and reactive adaptation, the data flow may have to be adapted as well after the control flow adaptation, e.g., by removing or adding data flow edges. For example, if a node *n* of the remaining control flow in Fig. 7 needs output data from the dropped *A*-node 15, it may be necessary to compensate the dropping of the *A*-node by generating a data flow edge for *n* which retrieves the needed data from external data sources (see 4.4).

4.2. Workflow duration estimation

In this section, we sketch our approach of estimating workflow execution durations for predictive adaptation.

To estimate which workflow part P_{VT} will be executed during a valid time interval *VT* we use *temporal meta information* about the estimated duration for each node and edge type. For simplicity we assume a negligible duration of control nodes, control edges, external writing and

[5] In AgentWork, a loop termination condition is specified at the LOOP-END node as loops have a repeat/until semantics.

internal data flow edges. On the other hand, duration estimates are needed for the execution of activity nodes, for data flow edges reading external data and for synchronization edges. AGENTWORK supports two ways to obtain such duration estimates: They can either be specified at workflow definition time or they are derived from actual measurements during workflow execution. To support high estimation accuracy the durations can be grouped according to different dimensions. Thus we can have different values per activity type, user type and data source, e.g., to account for that a beginner may need substantially more time for an activity than a sophisticated user. The measurement-based estimates can also consider queuing effects such as execution delays due to a lack of resources.

In the current implementation, estimations are based on *average* duration values. *Worst*-case durations using the *maximal* duration are viewed as too pessimistic as not enough adaptations may be performed, frequently requiring additional adaptations. *Best*-case durations using the *minimal* duration cause the opposite effect so that too many adaptations are triggered that may have to be revoked later on. Using average durations may also require reconsidering adaptations based on monitoring (see 4.4) but it is expected that this is needed in fewer cases. [6] Moreover, workflow applications often specify precise execution durations, e.g. medical workflows based on therapy guidelines (e.g., [25]) precisely specify the duration of drug infusions to maximize the therapeutic effect and minimize side-effects. With such specifications and measured execution durations we expect to be able to correctly estimate the workflow part P_{VT} in many cases.

Estimation of P_{VT}. To estimate the workflow part P_{VT} to be executed during the valid time interval *VT* it is first determined which running workflow is affected by the logical failure and which of its nodes are activated, i.e. would have to be executed next (e.g., nodes 5 and 14 in Fig. 7). These nodes form the failure node set. The execution durations of all paths starting at these nodes are estimated. This is done by estimating and adding the durations of the blocks the paths consist of. Estimation of one path stops if *VT* is "consumed" by that path or if there are irresolvable conditions at OR-SPLIT or LOOP-END nodes (see below). P_{VT} then consists of all nodes and edges of the estimated paths which are assumed to be executed during *VT*. In the sequel we discuss how the execution duration of blocks is estimated.

The duration of a sequence of activity nodes (e.g., nodes 5, 6, 7 in Fig. 7) is estimated by summing up the average execution durations of all its activities, data flow and synchronization edges.

In the case of an AND-SPLIT/AND-JOIN block implying the parallel execution of multiple paths, each of the paths starting at the AND-SPLIT node is separately estimated. Since the AND-JOIN node delays further execution until the slowest path is finished we take the maximum of the estimated (average) path durations as an estimate of the block duration.

For OR-SPLIT/OR-JOIN blocks the duration of the individual paths can be estimated as for AND-SPLIT/AND-JOIN blocks. If all paths do not significantly differ in their estimated duration, the average path duration is used as an estimate for the block duration. Otherwise, it is tried to predict which paths will be executed. This is difficult because the split decision depends on current data values. If the data needed for the decision is already available when the control action

[6] Note that even a more complex temporal constraint model using time intervals (e.g., [14]) to capture the uncertainty about the actual execution durations would face similar problems as only a minimal and a maximal workflow part P_{VT} may be estimated. So the exact part P_{VT} is not known and a monitoring is required.

has been triggered, AgentWork uses this data to determine the corresponding path. For the example of Fig. 7, if it is known at estimation time (e.g., from former examinations) that the patient has liver metastases it is predicted that for the lower OR-SPLIT only the lower path (sequence of nodes 18 and 19) has to be considered. If some split conditions cannot be evaluated in advance AgentWork excludes the entire OR-SPLIT/OR-JOIN block (and all later workflow parts of that path) from predictive adaptation and switches to reactive adaptation. For example, the split decision for the upper OR-SPLIT in Fig. 7 cannot be predicted if the current leukocyte count is unknown. Note that the availability of older blood values is of no help here as leukocyte counts may change significantly within a few days.

The estimation of *loops* faces similar difficulties than for OR-SPLITs since the exact number of loop iterations mostly depends on current data which may be produced during an loop iteration. For the estimation of a loop duration, we distinguish two cases. If the control action has been triggered before the LOOP-START node has been reached the duration of a loop is estimated by determining the duration of the loop's body and multiplying it with the estimated number of loop iterations. In AgentWork, the estimated number of iterations is either specified at workflow definition time (based on heuristics such as "On average, the radiotherapy unit of type *A* has to be repeated three times until the tumor vanishes") or on the measured average number of iterations of the respective loop during previous workflow executions. If the control action has been triggered during loop execution, estimation of this path stops at the LOOP-END node and AgentWork switches to reactive adaptation for the further loop iterations and all later parts of this path.

Recently, several other workflow estimation approaches have been suggested to support tasks such as deadline management and scheduling for workflows [16,29,37]. However, they differ from our approach as they do not use execution duration measurements and do not try to resolve conditional splits predictively.

4.3. Control flow adaptation

To translate control actions of ECA rules into structural control flow adaptations on specific workflow nodes and edges, AgentWork provides a structural control flow operator for each control action. Note that these structural control flow operators have to be distinguished from control actions of the rules, which are not linked to specific workflows. Control actions, e.g. for adding or dropping activities, cannot consider the current state of running workflows and do not specify which structural changes to the workflow definition, such as adding and removing nodes and edges, need to be performed to implement the respective workflow adaptation. This is the task of the structural control flow operators described now. We only sketch the operators *drop-node, add-node* and *add-node-loop* (implementing the *drop*, *add* and *add-repetitively* control actions, respectively), as the other operators can be mapped to variations and combinations of these ones. For instance, a node *replacement* is achieved by dropping a node and adding another one at the same position, while *postponement* is achieved by dropping a node and inserting it at a later position of the workflow. An update of activity attributes is currently handled as a replacement, but this solution may be improved if the operation turns out to be frequently used. The operators are used for both predictive and reactive adaptation.

Node dropping. For dropping a node, the operator *drop-node*(*n*: *Integer*) is provided. This operator takes as input the identifier *n* of an activity node to be dropped. Note that the shift from activity definitions and patterns (i.e., the activity semantics) to the specific node identifiers is already done by the mechanisms described in Section 4.1. The mechanisms described there decide whether or not a node with identifier *n* is affected by a control action such as *drop*.

The effect of the *drop-node* operator depends on the particular structure of the workflow part to which the affected node *n* belongs:

(a) If *n* is located in a sequence, it is simply removed from the control flow. Incoming and outgoing control flow edges are merged, and incoming and outgoing data flow edges are removed.
(b) If *n* is the only node of a path *p* in an AND-SPLIT/AND-JOIN block, *p* is removed. If there is only one remaining path after *p* has been removed, the AND-SPLIT and the AND-JOIN node are removed as well, as they are not needed anymore.
(c) If *n* is the *only* node of a path *p* in an OR-SPLIT/OR-JOIN block, *n* is removed, but *p* is left within the block as an empty path. This is necessary to keep the conditional semantics of the workflow.

Node adding. For adding a node, the operator $add\text{-}node(A : Activity\text{-}Pattern, n : Integer)$ is provided. The first parameter specifies the activity pattern *A* that shall be assigned to the new node. As this node has not been present in the workflow so far, its activity semantics has to be specified initially (in contrast to a node to be dropped). The semantics of the second parameter *n* is that the new *A*-node shall be inserted either directly after *n* or parallel to *n*, if possible. By default, AGENTWORK selects a node just committed or currently executed for *n*, but the user can specify any other node.

We distinguish two principal mechanisms to add an *A*- node, namely *sequential* and *parallel* add.

Sequential add. The straight-forward way to insert the new *A*-node is to insert it directly behind node *n* (Fig. 9a). If *n* identifies an AND-SPLIT node any path can be chosen for node adding as all paths will be executed. If *n* identifies an OR-SPLIT node it is either estimated which of the conditional paths will be executed (if possible), or the addition is delayed until the conditional branching is executed so that the new *A*-node can be inserted into a qualifying path.

Parallel add. Sequential add has the disadvantage that it typically delays the execution of successor nodes of *n* (e.g., node 2 in Fig. 9a). Parallel add aims at minimizing such execution delays by inserting the new node into a new parallel path. In the example of Fig. 9b, a new A'-node 4 has been inserted into a new path, within a new AND-SPLIT/AND-JOIN block, parallel to nodes 1 and 2. AGENTWORK tries to use temporal estimates to find an optimal AND-SPLIT/AND-JOIN block so that the new parallel path does not take longer to execute than the other parallel path consisting of already existing nodes. For this reason, in Fig. 9b the AND-JOIN node was not inserted directly after node 1 but after node 2, to avoid that the new A'-node 4 (which is assumed to take longer than node 1 but less than nodes 1 and 2 together) delays the execution of node 2.

By default AGENTWORKuses parallel add for implementing *add-node*. Sequential add is only applied when the temporal optimization for parallel add is not possible. This is the case when the

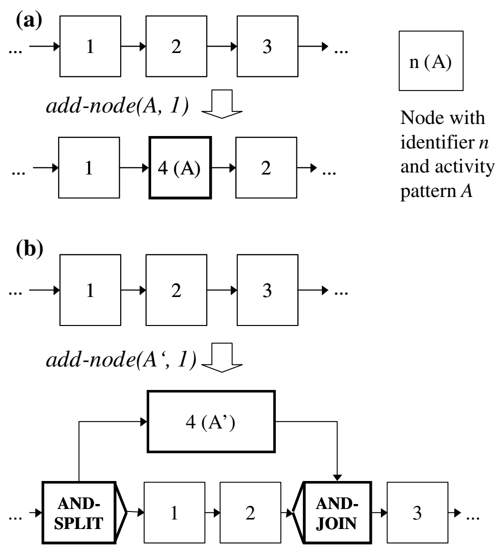


Fig. 9. Application of control flow operator *add-node*. (Node rectangle length proportional to execution duration.).

needed temporal estimations cannot be performed due to loops or OR-SPLIT/OR-JOIN blocks with unpredictable paths (as discussed in 4.2).

Operator for repetitive node adding. To add nodes for repetitive activity executions, AgentWork provides the control flow operator *add-node-loop*$(A : \textit{Activity-Pattern}, p : \textit{Duration}, \textit{cond} : \textit{Condition})$.

The first parameter specifies the activity pattern A that shall be assigned to the new node to be executed repetitively. The second parameter specifies the duration between two subsequent executions of the A-node. The third parameter specifies the termination condition of the loop.

Add-node-loop realizes the repetitive execution of an A-node by inserting a loop with an A-node and a termination condition *cond* into the workflow. In contrast to *add-node, add-node-loop* does not try to find a temporally optimized AND-SPLIT/AND-JOIN block since this would require a reliable estimation for the execution duration of the loop. As already discussed in 4.2, such loop estimations are difficult in particular when the loop is terminated by a qualitative condition such as "until leukocyte count higher than 2500". To still avoid temporal delays, AgentWork follows the simple approach of inserting the loop within a "maximal" AND-SPLIT/AND-JOIN block, as shown in Fig. 10. Hence, a new AND-SPLIT node is directly inserted after the START node and the new AND-JOIN node directly before the END node. Moreover, a loop of the structure

$$\textit{LOOP-START} \rightarrow \textit{A-node} \rightarrow \textit{LOOP-END}$$

is inserted into the new empty path. The loop duration specified by p is translated into a waiting condition with $min = max = p$ which is assigned to the edge between the LOOP-END node and the LOOP-START node. The termination condition specified by parameter *cond* is assigned to the edge between the LOOP-END node and the new AND-JOIN node.

As dropping or adding activities such as drug administrations may have significant influence on a workflow, workflow adaptations are viewed as suggestions that have to be confirmed by

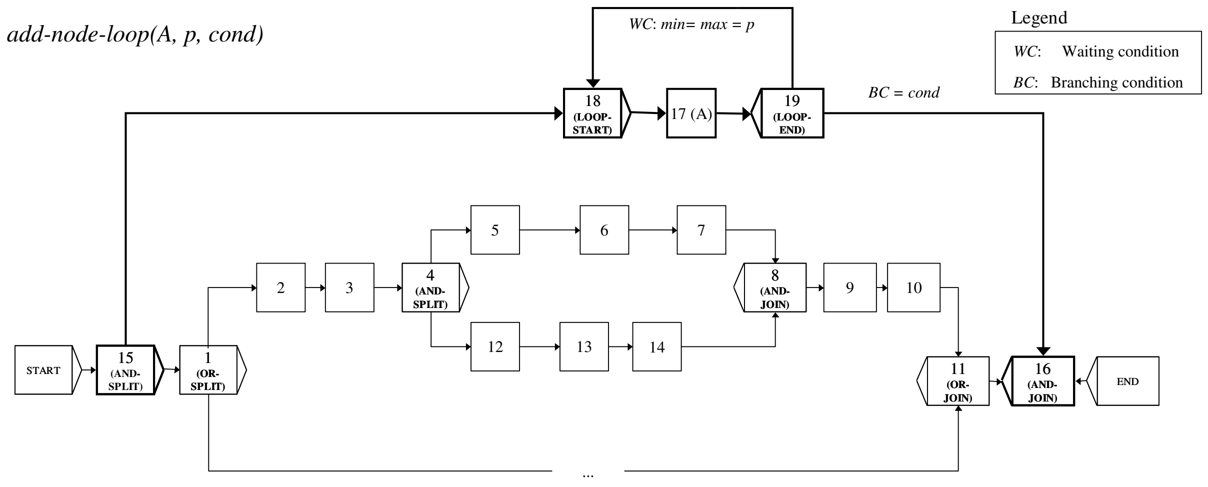


Fig. 10. Application of control flow operator add-node-loop.

a user. For example, a physician may reject the dropping of a drug administration node despite some negative side-effects, if he thinks that the drug administration is important for the patient.

4.4. Data flow adaptation

A data flow adaptation is required if a control flow adaptation results in activity nodes for which at least one input object is not provided by the data flow anymore. For example, the output of a dropped node may be needed by a remaining activity node or the input for a newly added node may have to be provided. Thus, appropriate data flow edges have to be generated to provide necessary input objects.

Analogously to control flow adaptation, data flow adaptation can be done *reactively* or *predictively. Reactive data flow adaptation* means that the input object completeness of a node *n* is checked directly before *n* is executed. If at least one input object is missing, the data flow is adapted. Reactive data flow adaptation strategy can be combined both with reactive and predictive control flow adaptation. That is, even if the control flow is handled predictively, the necessary data flow adaptations may be delayed until the respective activity nodes are to be executed.

Predictive data flow adaptation means that directly after a control flow adaptation input object completeness is checked for all nodes that still have to be executed. If at least one input object is missing, the data flow is adapted predictively. Note that predictive data flow adaptation can also be used for a reactive control flow adaptation. For instance, when reactively dropping a node *n*, it can be checked whether this leaves a successor node of *n* without an input object (as this input object has been provided by *n*). In this case, it may be possible to predictively adapt the data flow to provide the input object in a different way.

Analogously to control flow adaptation, the predictive approach is used whenever possible since reactive data flow adaptation can result in significant delays. For example, if a new therapeutic node requiring an X-ray finding as input is added to a medical workflow, the new node may have to be delayed until the X-ray examination has been performed.

Data flow adaptation is based on constraints that can be specified in the activity definition for input and output objects. In addition to value constraints on these objects we also support temporal constraints, in particular to specify the currentness of input data. These constraints are used to decide whether a missing input object can be provided by activity nodes in the workflow. We illustrate this by the example of Fig. 11 where a data flow adaptation is needed to provide the input object *h* for a newly added node *n*. According to the activity pattern of *n*, *h* has to meet the value constraint that it should represent the leukocyte count. Furthermore, the temporal *NOT-OLDER-THAN* constraint requires that the leukocyte count must not be older than 2 days when *n* is executed.

To perform data flow adaptation in this case, the temporal neighborhood of *n* is explored first. This means that by workflow estimations it is checked whether there is any output object *o* of an activity node *m* meeting the constraints, i.e.,

- *o* is of the same type (e.g., *Blood-Finding* in Fig. 11) and attribute values (e.g., *parameter* = *Leukocyte-Count*) as the input object *h* of *n*, and
- is provided not earlier than the point in time when *n* needs the input object *h* minus the distance specified by the respectively *NOT-OLDER-THAN* constraint of *h*, and not later than the point in time when *n* needs *h*.

If these conditions are fulfilled, an *internal* data flow edge is generated that maps *o* to *h*. In the example of Fig. 11, it has been determined that node *m* belongs to the relevant neighborhood w.r.t. *h*, and provides an output object *o* with the same type and attribute values as *h*. Therefore, an internal data flow edge mapping *o* to *h* is generated.

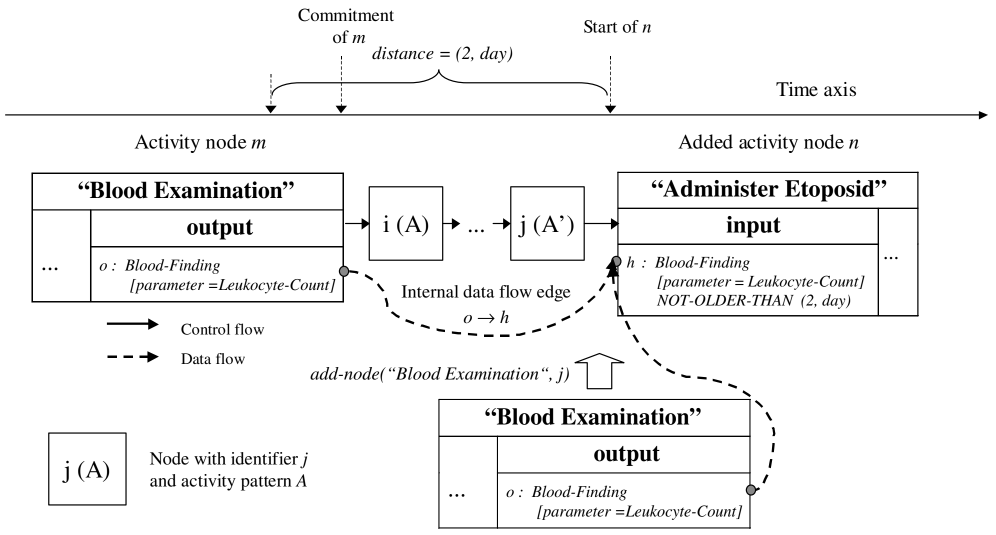

Commitment of m
Start of n
distance = (2, day)
Time axis
Activity node m
Added activity node n
"Blood Examination"
output
...
o : Blood-Finding [parameter =Leukocyte-Count]
i (A)
...
j (A')
"Administer Etoposid"
input
h : Blood-Finding [parameter = Leukocyte-Count] NOT-OLDER-THAN (2, day)
Internal data flow edge
o → h
Control flow
Data flow
add-node("Blood Examination", j)
"Blood Examination"
output
o : Blood-Finding [parameter =Leukocyte-Count]
j (A)
Node with identifier j and activity pattern A

Fig. 11. Generation of data flow edges.

If the local temporal neighborhood of the workflow does not provide such a suitable output object *o*, AgentWork inserts an activity node providing the needed data object (due to the activity node's output specification). For example, in Fig. 11 an additional blood examination node would be inserted by the *add-node* operator (with the blood examination activity pattern as first parameter and the activity identifier of the predecessor node of *n* (i.e., *j*) as the second parameter of *add-node*).

For details about data flow generation we refer to [39].

4.5. Workflow monitoring

Workflow monitoring is needed to supervise predictive adaptation and is performed by a monitoring agent. It continuously checks whether the temporal estimation on which an adaptation is based matches the actual execution of the adapted workflow. For example, estimations may be imprecise or be invalidated by technical errors or subsequent workflow adaptations which

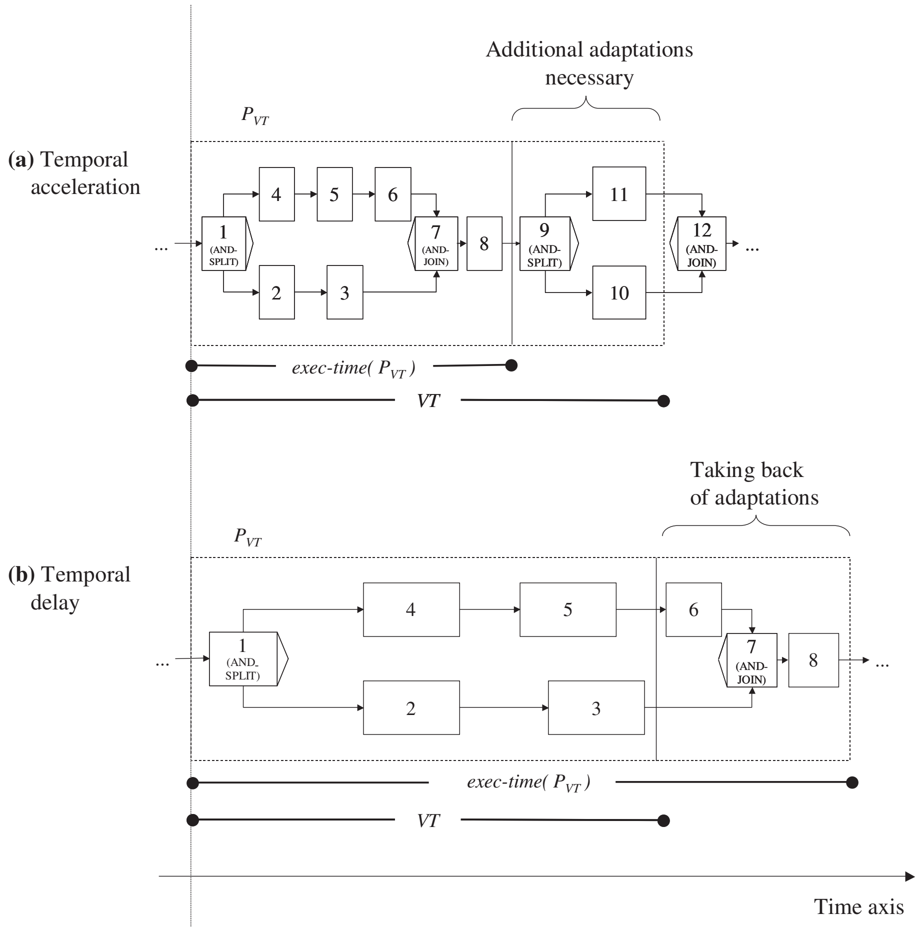


Fig. 12. Temporal mismatches and workflow monitoring.

add or drop nodes. Detected deviations from the original estimates may require to perform additional adaptations or to take back some of the planned adaptations.

We distinguish two principal mismatch types between estimations on one side and the actual execution on the other side, namely temporal *acceleration* and temporal *delay* (Fig. 12). Let P_{VT} again denote the workflow part that is assumed to be executed during a valid time *VT*, and let *exec-time*(P_{VT}) denote the interval actually needed to execute P_{VT}.

Temporal acceleration. P_{VT} is executed *faster* than it has been estimated. This means that workflow parts which have not been considered so far are now additionally to be executed during *VT*. For example, in Fig. 12a nodes 1–8 are assumed to have executed faster than estimated so that the workflow part consisting of nodes 9–11 may now be executed during *VT* as well. Hence, this part may have to be additionally adapted to satisfy control actions that are valid during *VT*.

Temporal delay. The execution of P_{VT} is delayed. This means that parts of P_{VT} that have been assumed to be executed during *VT* will not be executed anymore during *VT*. For example, in Fig. 12b the execution of the nodes 1–5 has taken longer than estimated causing that a part of P_{VT} (nodes 6–8) cannot be executed during *VT*. Hence, adaptations for this part may have to be taken back.

5. Related work

In this section we discuss related work from the fields of commercial workflow management, advanced transaction models, adaptive workflow management, and artificial intelligence.

Several vendors and researchers have addressed failure and exception handling in workflow management systems [7,10,22,26,27,35,45–48]. However, only a few commercial systems such as ProMInanD, [26], InConcert [48], Action Request System [46], lotus domino workflow [22] or Ensemble [27] provide some support for workflow adaptation. For example, Action Request System [46] is able to derive by ECA rules that an additional activity has to be executed. However, the user has to select an appropriate insertion point in the workflow manually. Furthermore, ECA rules with a valid time dimension and predictive adaptation are not supported by any commercial system.

Several studies focused on workflow recovery from system failures, typically guided by an advanced transaction model supporting compensation and forward recovery [3,23,43,53]. These approaches do not deal with changing the structure of running workflows to handle logical failures.

Due to their formal capabilities, Petri nets have been applied to workflow management [1] and especially to dynamic workflow changes [18,51]. For example, in [18] a special subtype of Petri nets, so-called flow nets, is used for workflow modeling and control. At execution time, each flow net can control several workflow instances. With a graphical editor, structural adaptations of a flow net can be performed, such as that two activities which have been executed sequentially so far now have to be executed in parallel. The correctness of the adapted networks is guaranteed in terms of reachability analysis of the Petri net theory. However, the authors do not provide a set of predefined adaptation operators or a high-level adaptation interface, so that the person performing the adaptations has to know the syntactical details. Furthermore, as a flow net usually controls several workflow instances, the direct adaptation of single workflow instances is not possible. In addition to this, the authors do not consider the adjustment of data flows that may become necessary when a flow net is adapted.

In [51], the authors describe an approach where every workflow instance is controlled by exactly one so-called workflow net, which is a workflow-oriented Petri net subtype. Thus, ad hoc adaptation of single instances becomes possible. For this, a number of predefined transformation rules is provided, e.g., to refine an activity with a subworkflow, or to split up sequences to several paths executed in parallel or conditional, and to join them again. However, the limitation of this approach is that data flow aspects are neglected, and that the handling of loops remains unclear.

The TAM [7] system [35,55] provides constructs to specify interaction dependencies between activities in an application-dependent manner. These dependencies can dynamically be restructured if exceptions occur. Furthermore, any activity may be dynamically split into subactivities. Thus, manual reactive adaptation can be achieved by splitting activities, but automation of failure handling via ECA rules and predictive adaptation are not supported.

Several recent research approaches have used ECA rules to specify which actions have to be performed on workflows when failures occur [7,10]. For example, in Chimera-Exc [7] Datalog-based rules can be defined to monitor events and to derive appropriate actions. In [10], ECA rules are used in combination with a nested transaction model to consider data dependencies between sub-workflows. However, these approaches do not consider the valid time dimension of triggered workflow adaptations and do not support predictive adaptation.

The ADEPT$_{\text{flex}}$ system [45] provides an operator set for workflow adaptation (e.g., for dropping and inserting nodes and edges) by preserving correctness and consistency of adapted workflows. Temporal implications of workflow adaptations such as deadline violations for workflow activities are considered, too [13]. However, no algorithms are specified that decide automatically under which circumstances which structural adaptations should be applied. The operator applications have to be selected by a user. Thus, automated and predictive workflow adaptation is not supported.

In [47], partially defined workflows can be executed that contain so-called "pockets of flexibility" with a set of workflow fragments and rules stating how these fragments may be refined at runtime. Thus, a workflow can only be adapted at predefined places. In particular, the temporal dimension of adaptations such as "drop this activity for the next 5 days" is not supported.

In a medical project, we have used workflow refinement for dynamic workflow adaptation [40]. At execution time, when all patient data is available, it is decided automatically which particular sub-workflow shall be selected to treat the patient in an optimal manner. However, in this approach predictive and event-oriented adaptation is not supported.

Recently, techniques from the field of artificial intelligence have been applied to workflow management, in particular planning techniques [5,24,34,42,50] and cooperative agent approaches [15,28,32,43,52]. However, the usage of planning techniques for our specific problem of predictive workflow adaptation is limited. This is because these approaches typically do not support the temporal dimension of failures sufficiently and do not consider operational aspects such as the consequences of an control flow adaptation for the data flow. Cooperative agent approaches

[7] TAM = Transactional Activity composition Model.

provide a sophisticated way to detect and handle failures (e.g., [32]), but do not address the structural consequences of failure handling on the graph level.

6. Summary and future work

In this paper, we have given an overview of the workflow management prototype AgentWork which provides a comprehensive support for automated workflow adaptation. AgentWork uses ECA rules based on a temporal logic to automatically cope with logical failures occurring during workflow execution. AgentWork supports both reactive and predictive adaptation of workflows and tries to apply predictive adaptations whenever possible. This is achieved by suitable estimation algorithms based on pre-specified or measured execution durations of activities and for external data access. In addition to control flow adaptations, predictive and reactive data flow adaptations are supported.

We believe that the timely and largely automated handling of logical failures can significantly improve the flexibility and quality of workflow executions, in contrast to currently available solutions. In the considered application area of cancer therapies these advantages are of critical importance. They cannot only reduce the administrative burden for the personnel but also improve the treatment of patients.

Within a research project funded by the German Research Association (DFG), we have implemented a prototype of the AgentWork system. As a core, we use the ADEPT$_{\mathrm{flex}}$ workflow management system [45] for the *workflow definition and execution layer* of agentwork. ADEPT$_{\mathrm{flex}}$ has been selected, as it—in contrast to most commercial workflow management systems and research prototypes—supports the specification of execution durations for activities and provides basic operators for dropping and adding nodes in workflow instances during runtime which can be invoked via a Java API (Application Programming Interface).

For the *event monitoring agent* we could not use existing F-Logic implementations (e.g., [20]), mainly because of their insufficient API capabilities. Therefore, we map active rules specified in ActiveTFL to database triggers. Control actions derived by these database triggers are then sent to the *adaptation agent* using XML (eXtensible Markup Language) messages. We decided to use XML as it is a widespread data interchange format for which various communication infrastructures exist (e.g., XML-RPC). The algorithms for workflow estimation as described in 4.2 have been implemented in Java in a straightforward manner using the activity execution durations provided by the ADEPT$_{\mathrm{flex}}$ workflow model. The adaptation itself has been realized by directly invoking the ADEPT$_{\mathrm{flex}}$ control and data flow operators using ADEPT$_{\mathrm{flex}}$ API. The workflow monitoring agent has been implemented in Java, too.

The implementation has shown the feasibility of our failure handling approach. We plan empirical studies on the usability of AgentWork and the quality of temporal estimations for real-world workflows. Furthermore, we plan to consider other factors than "time", such as cost and "quality of service/product". Future work also has to concentrate on "schema evolution" aspects, such as the problem how to deal with changes of an ECA rule which previously led to a predictive change, and on a more elaborated rule termination approach. We will also evaluate the applicability of the approach in different application domains such as e-business.

Acknowledgements

We thank the Database Section (Head: Prof. Dr. Peter Dadam) of the Department of Computer Science, University of Ulm, Germany, for kindly providing us the ADEPT$_{\text{flex}}$ prototype. The work of Ulrike Greiner has been supported by the German Research Association. Furthermore, we thank the two anonymous reviewers for their detailed and helpful remarks.

References

[1] N.R. Adam, V. Atluri, W.-K. Huang, Modeling and analysis of workflows using Petri nets, Journal of Intelligent Information Systems 10 (2) (1998) 131–158.
[2] G. Alonso, C. Mohan, WFMS: the next generation of distributed processing tools, in: S. Jajodia, L. Kerschberg (Eds.), Advanced Transaction Models and Architectures, Kluwer, Dordrecht, 1997, pp. 35–62.
[3] V. Atluri, W.-K. Huang, E. Bertino, A semantic based execution model for multilevel secure workflows, Journal of Computer Security 8 (1) (2000) 3–41.
[4] S. Baker, CORBA distributed objects, Addison Wesley, Reading, MA, 1997.
[5] C. Beckstein, J. Klausner, A meta level architecture for workflow management, Journal of Integrated Design and Process Science 3 (1) (1999) 15–26.
[6] J. Benthem, Temporal logic, in: D.M. Gabbay, C.J. Hogger, J.A. Robinson (Eds.), Handbook of logic in artificial intelligence and logic programming, Epistemic and Temporal Reasoning, vol. 4, Oxford University Press, Oxford, UK, 1995.
[7] F. Casati, S. Ceri, S. Paraboschi, G. Pozzi, Specification and implementation of exceptions in workflow management systems, ACM TODS 24 (1999) 405–451.
[8] S. Chakravarthy, V. Krishnaprasad, E. Anwar, S.-K. Kim, Composite events for active databases: semantics contexts and detection, in: Proc. VLDB'94, Morgan Kaufmann, Los Altos, 1994, pp. 606–617.
[9] D.K.W. Chiu, Q. Li, K. Karlapalem, Web interface-driven cooperative exception handling in ADOME workflow management system, Information Systems 26 (2001) 93–120.
[10] D.K.W. Chiu, A three-layer model for workflow semantic recovery in an object-oriented environment, in: Proceedings of ER 2001, Lecture Notes in Computer Science, vol. 2224, Springer, Berlin, 2001 pp. 541–554.
[11] J. Chomicki, G. Saake, Logics for databases and information systems, Kluwer, New York, 1998.
[12] J. Chomicki, D. Toman, Temporal logic in information systems, in: J. Chomicki, G. Saake (Eds.), Logics for Databases and Information Systems, Kluwer, New York, 1998, pp. 31–70.
[13] P. Dadam, M. Reichert, K. Kuhn, Clinical workflows—the killer application for process-oriented information systems? in: Proceedings of the International Conference on Business Information Systems, Springer, Berlin, 2000, pp. 36–59.
[14] R. Dechter, I. Meiri, J. Pearl, Temporal constraint networks, Journal of Artificial Intelligence 49 (1991) 61–95.
[15] S.M. Deen, Cooperating agents for holonic manufacturing, in: Proceedings of the Multi-Agent-Systems and Applications, 2001, pp. 119–136.
[16] J. Eder, E. Panagos, M. Rabinovich, Time constraints in workflow systems, in: Proceedings of the CAiSE 1999, Springer, Berlin, 1999, pp. 286–300.
[17] E. Ehud Gudes, M.S. Olivier, R.P. van de Riet, Modeling, specifying and implementing workflow security in cyberspace, Journal of Computer Security 7 (4) (1999).
[18] C.A. Ellis, K. Keddara, W. Wainer, Modeling workflow dynamic changes using timed hybrid flow nets, in: Proceedings of the ICATPN, 1998, pp. 109–128.
[19] E. Franconi, Description logics for natural language processing, in: F. Baader, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider (Eds.), Description Logics Handbook, Cambridge University Press, Cambridge, 2002.
[20] F. Frohn, R. Himmeröder, P.-Th. Kandzia, G. Lausen, C. Schlepphorst, FLORID—a prototype for F-Logic, in: Proceedings of the 7th Bi-Annual German Database Conference (BTW'97), Springer, Berlin, 1997, pp. 100–117.
[21] D. Georgakopoulos, M. Hornick, A. Sheth, An overview of workflow management: from process modeling to infrastructure for automation, Journal on Distributed and Parallel Database Systems 3 (1995) 119–153.

[22] G. Giblin, R. Lam, Programming workflow applications with Domino, R&D Publications, 2000.
[23] P.W.P.J. Grefen, J. Vonk, P.M.G. Apers, Global transaction support for workflow management systems: from formal specification to practical implementation, VLDB Journal 10 (4) (2001) 316–333.
[24] K.J. Hammond, Explaining and repairing plans that fail, Artificial Intelligence 45 (1990) 173–228.
[25] K. Havemann, H. Köppler, U. Haag, Integratives Konzept zur Behandlung Hoch-Maligner Non-Hodgkin-Lymphome. Studie A, Marburg, Germany, 1994.
[26] IABG, Reference manuals of ProMInanD, IABG Company, Munich, 1999.
[27] InterSystems Corporation, Ensemble. Available from <http://www.intersystems.com/ensemble/index.html>.
[28] N.R. Jennings, P. Faratin, T.J. Norman, P. O'Brien, B. Odgers, Autonomous agents for business process management, International Journal of Applied Artificial Intelligence 14 (2) (2000) 145–189.
[29] E. Kafeza, K. Karlapalem, Temporally constrained workflows, in: Proceedings of the ICSC 1999, Lecture Notes in Computer Science, vol. 1749, Springer, Berlin, 1999, pp. 246–255.
[30] M. Kifer, G. Lausen, J. Wu, Logical foundations of object-oriented and frame-based languages, Journal of the ACM 42 (1995) 741–843.
[31] B. Kiepuszewski, A.H.M. ter Hofstede, C. Bussler, On structured workflow modeling, in: Proceedings of the CAiSE'2000, Lecture Notes in Computer Science, vol. 1789, Springer, Berlin, 2000, pp. 431–445.
[32] M. Klein, C. Dellarocas, A knowledge-based approach to handling exceptions in workflow systems, Journal of Computer-Supported Collaborative Work 9 (3/4) (2000) 399–412.
[33] A. Lazcano, H. Schuldt, G. Alonso, H.-J. Schek, WISE: process based e-commerce, IEEE Data Engineering Bulletin 24 (1) (2000) 46–51.
[34] C. Liu, R. Conradi, Automatic replanning of task networks for process model evolution in EPOS, in: Proceedings of the ESEC'93, 1993, pp. 434–450.
[35] L. Liu, C. Pu, Methodical restructuring of complex workflow activities, in: Proceedings of the ICDE 1998, IEEE Computer Society Press, Silver Spring, MD, 1998, pp. 342–350.
[36] Z. Manna, A. Pnueli, The temporal logic of reactive and concurrent systems, Springer, Berlin, 1992.
[37] O. Marjanovic, M.E. Orlowska, On modeling and verification of temporal constraints in production workflows, Knowledge and Information Systems 1 (1999) 157–192.
[38] I. Motakis, C. Zaniolo, Temporal aggregation in active database rule, in: Proceedings of the SIGMOD 1997, SIGMOD Record 26 (2) (1997) 440–451.
[39] R. Müller, Event-oriented dynamic adaptation of workflows, Ph.D. Thesis, Department of Computer Science, University of Leipzig, 2002.
[40] R. Müller, B. Heller, A petri net-based model for knowledge-based workflows in distributed cancer therapy, in: Proceedings of the EDBT'98 Workshop on Workflow Management Systems, 1998, pp. 91–99.
[41] R. Muller, E. Rahm, Dealing with logical failures for collaborating workflows, in: Proceedings of the CoopIS 2000 Lecture Notes in Computer Science, Proceedings of the CoopIS 2000 Lecture Notes in Computer Science, vol. 1901, Springer, Berlin, 2000, pp. 210–223.
[42] K. Myers, Towards a framework for continuous planning and execution, in: Proceedings of the AAAI Symposium on Distributed Continual Planning, 1998.
[43] K. Nagi, J. Nims, P.C. Lockemann, Transactional support for cooperation in multiagent-based information systems, in: Proceedings of vertIS2001, 2001, pp. 177–191.
[44] N. Paton (Ed.), Active Rules in Database Systems, Springer, Berlin, 1999.
[45] M. Reichert, P. Dadam, ADEPT$_{\text{flex}}$—supporting dynamic changes of workflows without losing control, Journal of Intelligent Information Systems 10 (1998) 93–129.
[46] Remedy Corporation, Action request system 4.0 reference manuals, Remedy Corporation, 2000.
[47] S.W. Sadiq, W. Sadiq, M.E. Orlowska, Pockets of flexibility in workflow specification, in: Proceedings of the ER 2001, Lecture Notes in Computer Science, vol. 2224, Springer, Berlin, 2001, pp. 513–526.
[48] S.K. Sarin, Workflow and data management in InConcert, in: Proceedings of the Twelfth International Conference on Data Engineering ICDE 1996, IEEE Computer Society, Silver Spring, MD, 1996, pp. 497–499.
[49] A. Sheth, K. Kochut, et al., Supporting state-wide immunization tracking using multi-paradigm workflow technology, in: Proceedings of the VLDB 1996, Morgan Kaufmann, Los Altos, 1996, pp. 263–273.

[50] M.P. Singh, M.N. Huhns, Automating workflows for service order processing, integrating AI and database technologies, IEEE Expert 9 (5) (1994).

[51] M. Voorhoeve, W. van der Aalst, Ad-hoc workflow: problems and solutions, in: Proceedings of the 8th DEXA, 1997, p. 3641.

[52] W. de Vries, F.S. de Boer, W. van der Hoek, J.-J.Ch. Meyer, A truly concurrent model for interacting agents, in: Proceedings of PRIMA 2001, Lecture Notes in Computer Science, vol. 2132, Springer, Berlin, 2001, pp. 16–30.

[53] H. Wächter, A. Reuter, The ConTract model, in: A.K. Elmagarmid (Ed.), Database Transaction Models for Advanced Applications, Morgan Kaufmann, Los Altos, CA, 1992, pp. 219–263.

[54] D. Worah, A. Sheth, Transactions in transactional workflows, in: S. Jajodia, L. Kerschberg (Eds.), Advanced Transaction Models and Architectures, Kluwer, Dordrecht, 1997, pp. 3–34.

[55] T. Zhou, L. Liu, C. Pu, TAM: a system for dynamic transactional activity management, in: Proceedings of the SIGMOD Conference 1999, 1999, pp. 571–573.

Robert Müller received his doctoral degree on a dissertation on workflow management from the University of Leipzig, Germany, in 2002. From 1994 to 1996, he was a research scientist at the University Hospital of Mainz at the Department of Medical Informatics. Since 1996, he has been working at the Department of Computer Science, Database group, of the University of Leipzig, Germany. After having hold a deputy professorship for databases at the Department of Computer Science of the University of Munich in the summer term 2003, he is now a professor at the Leipzig University of Applied Sciences.

Ulrike Greiner received her diploma in computer science from the University of Leipzig, Germany, in 2000. Since then, she has been a research scientist at the Department of Computer Science, Database group, of the University of Leipzig. Her current research topics include workflow management and e-services.

Erhard Rahm received his Ph.D. degree in Computer Science from University of Kaiserslautern, Germany, in 1988. From 1988 to 1989, he has been a visiting scientist at the IBM T.J. Watson Research Center, Hawthorne, NY, USA. Being an assistant professor at the Department of Computer Science, University of Kaiserslautern from 1989 to 1994, he received his habilitation degree in Computer Science in 1993 (habilitation thesis on “Architecture of high-performance transaction systems”). Since 1994, he is a full professor for Computer Science and the head of the Database group at the University of Leipzig, Germany. His current research topics include XML data management, adaptive workflow management, metadata management, web usage mining, data warehouses, and bioinformatics.