AgentWork: A Workflow System Supporting Event-Oriented Workflow Adaptation

(Technical Report)

Robert Müller[1], Ulrike Greiner, Erhard Rahm

Department of Computer Science, University of Leipzig, Germany

---

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, an event-based workflow management system supporting automated workflow adaptations in a comprehensive way. In particular, AgentWork uses temporal estimates to determine which remaining parts of running workflows are affected by an exception and is able to predictively perform suitable workflow adaptations. This helps ensure that necessary adaptations are performed in time with minimal user interaction which is especially valuable in complex applications such as for medical treatments.

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 [1,16,19]. 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 manage complex processes in internet-based e-commerce, virtual enterprises, or medical institutions [29,44]. For example, due to precisely specified treatment procedures in many medical disciplines, workflow management systems can be used to implement diagnostic and therapeutic processes [13,36,44]. Major goals are an 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 refer to application-specific exceptional events for which the control and data flow of a workflow is not adequate anymore and

[1] Corresponding author. E-mail addresses: {mueller, greiner, rahm}@informatik.uni-leipzig.de.

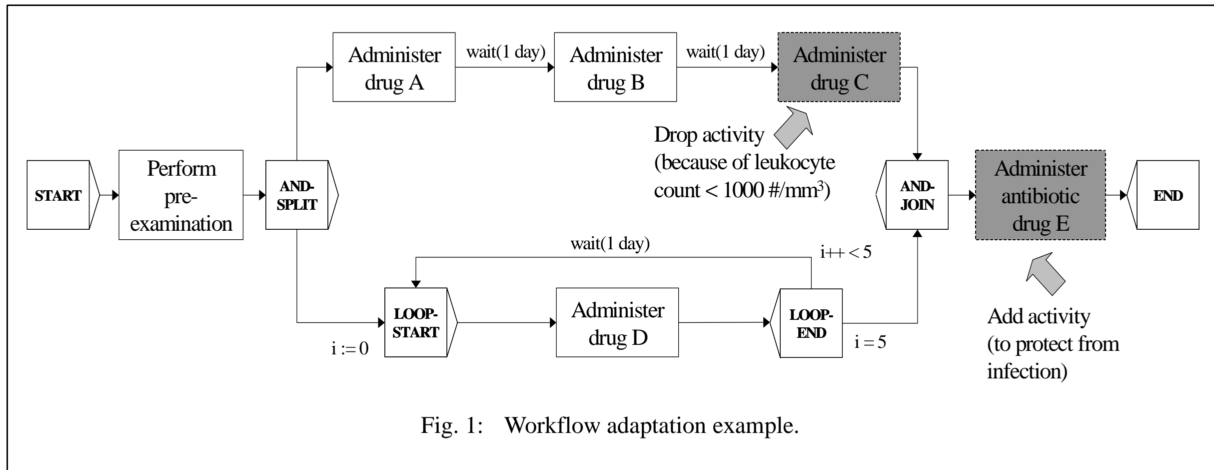

Fig. 1: Workflow adaptation example.

thus has to be adapted [48]. The automatic treatment of such logical failures is the main subject of this paper.

For example, in the cancer chemotherapy workflow shown in Fig. 1 it is detected just before the administration of drug *C* that the leukocyte count (i.e., the number of white blood cells per mm^3 of blood) 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 activity nodes can be continued without change. Furthermore, 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.

Previous work on dynamic workflow adaptation mostly focussed on a *manual* approach where the administrator or a user has to decide which events constitute logical failures and which adaptations have to be performed (e.g., [41]). 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 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 [7,9] typically limit adaptations to the currently executed workflow activities. 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, in cancer therapy adding a new drug administration typically requires ordering the necessary drugs one or two days before the planned administration in order 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, writing reviews in evaluation processes etc..

To overcome the limitations of existing systems and comprehensively support automated workflow adaptations, we designed and developed the event-oriented 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 *predictive* and *reactive* adaptation. **Predictive adaptation** adapts workflow parts affected by a logical failure in advance (predictively), typically as soon as the failure is detected. In many situations this gives enough time to meet organizational constraints for adapted workflow parts. **Reactive adaptation** is typically performed when predictive adaptation is not possible. In this case, adaptation is performed when the affected workflow part is to be executed, e.g., 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) model to automatically detect logical failures and to determine the necessary workflow adaptations. To support predictive workflow adaptations, we provide a novel ECA model based on 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 [35,36]. 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 subclassing, 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 sketch 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.

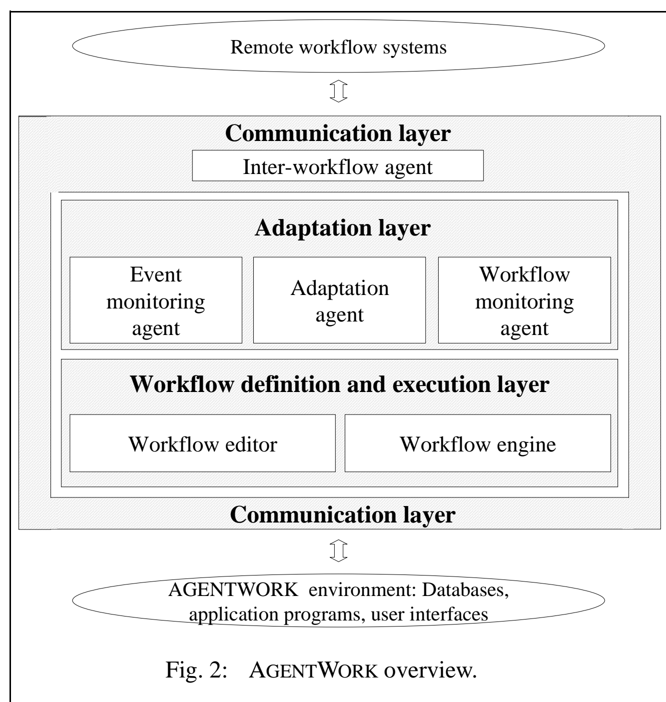

Fig. 2: AGENTWORK overview.

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* [24].

- 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 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).

- 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 [3] 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 [37], we do not further consider this agent here.

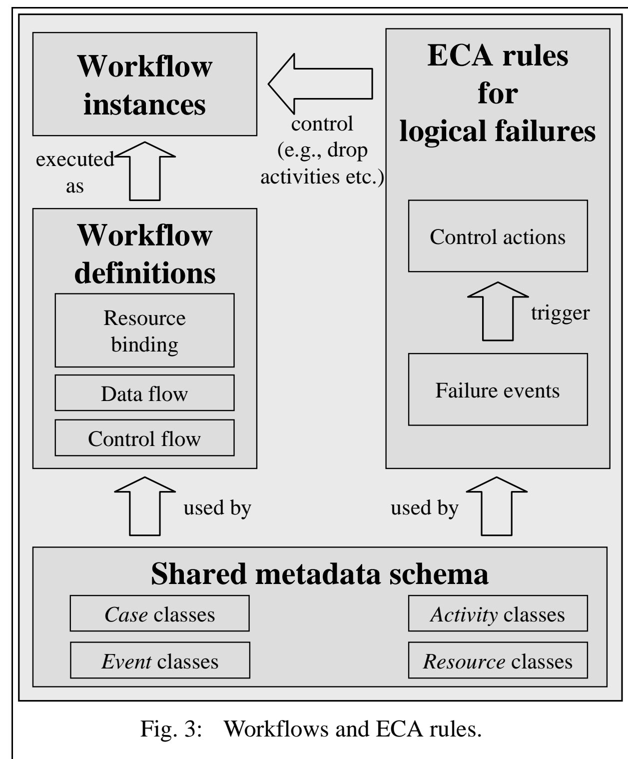

Fig. 3: Workflows and ECA rules.

2.2. Model overview

Fig. 3 shows the main model components of AGENTWORK. Workflow definitions and specification 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 (e.g., a patient or a customer). Objects of class *Event* represent anything that may lead to logical workflow failures, such as the new 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 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. 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 [27,41].

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.

As usual, the term *workflow instance* (or simply workflow) refers to an instantiation of a workflow definition executed by the workflow engine. For simplicity, we assume 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 [35].

Finally, AGENTWORK uses ECA rules [40] 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 for workflow *resources*, such as a broken computer tomography device making it temporarily impossible to execute some activities, can be treated analogously [35]. 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 and workflows has no or only minimal effects on ECA failure rules. 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:

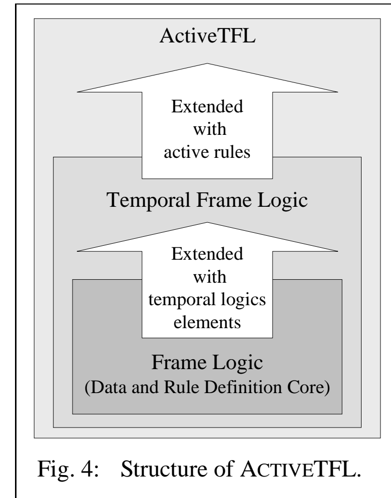

Fig. 4: Structure of ACTIVETFL.

WHEN	*new finding of patient P*	(I)
WITH	*leukocyte count < 1000*	
THEN	*drop drug Etoposid for P*	
VALID-TIME	*during the next seven days*	

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 [26], or Description Logic [17] 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. 4).[2]

3.2.1 Frame Logic

In the following, we introduce the relevant Frame Logic (FL) components by means of medical examples. This

[2] We have not selected Description Logic, as this logic focuses on terminological reasoning and natural language processing [17] which is not relevant for logical failure handling.

includes FL classes, objects, object extensions, object patterns, predicates, formulas, and rules.[3]

FL **class definitions** have the form

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]

Patient IS-A Case *Blood-Finding IS-A Event* *Drug-Administration IS-A Activity*

"IS-A" denotes the subclass relationship (e.g., *Patient* is a subclass of *Case)*. FL **objects** (i.e., class instances) are denoted as follows:

d: Drug-Administration[date = 7/2/03, time = 9.00 am, activity-for = bob,
drug = "ETOPOSID", 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**. For example,

extension patients(Patient) (II)

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

Class[constraints]

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

Drug-Administration[drug = "ETOPOSID", dosage > 50] (III)

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*.

---

[3] For better readability, we have adapted the FL syntax. Nevertheless, the language model is that of FL as described in [26].

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. 5 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 can be determined not 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 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 *drop* in (I), e.g., we can define the predicate

drop(A, CS) (IV)

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 [26].

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

WHEN *critical-blood-status(P)* (V)
THEN *drop(A, P)*

states that whenever a patient *P* has a critical blood status (e.g., leukocyte count < 1000) – expressed by the 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 (V) does not specify the *valid time* of the derived control action, i.e., for how long the

Fig. 5: Activity definition example.

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]*

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") [5]. 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 2003: 8 pm, 2 March 2003: 8 pm + (72, hour)*] specifies the set of points in time starting at 2 Dec 2003: 8 pm and ending after 72 hours (i.e., at 5 Dec 2003: 8 pm). Expressions of the structure *(amount, time-unit)* specify an amount of time, e.g., *(72, hour)* for 72 hours. 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 hours.

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

F VALID-TIME S

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

WHEN	*critical-blood-status(P)*	*VALID-TIME [now – (5, day), now]*	(VI)
THEN	*drop(A, P)*	*VALID-TIME [now, now + (7, day)].*	

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* [12,32]. 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: *(F Until G) VALID-TIME t*	iff	it holds: It exists $t' > t$ with: *G VALID-TIME t'* and for all t'' with $t \le t'' < t'$ it holds: *F VALID-TIME t'' AND NOT (G VALID-TIME t'')*

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

WHEN	*critical-blood-status(P) VALID-TIME [now – (3, day), now]*
THEN	*add-repetitively(Drug-Administration[drug = "DOXYCYCLIN"], (1, day), P)* *Until drop(Drug-Administration[drug = "ETOPOSID"], P)* *VALID-TIME now*

This rule states that whenever a patient *P* has had a critical blood status during the last 3 days, *P* must get the drug DOXYCYCLIN repetitively every day until the drug ETOPOSID is dropped. The medical background for this example is that a cytostatic drug such as ETOPOSID significantly increases the probability of serious bacterial infections because of its immune-suppressive side-effects. Therefore, antibiotic drugs such as DOXYCYCLIN are given prophylactically during chemotherapy when the blood situation becomes critical.

- ***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: *(F Unless G) VALID-TIME t* (at point in time t, F is valid unless G is valid)	iff	it holds: *(F Until G) VALID-TIME t OR* *F VALID-TIME [t, ∞)*

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):

WHEN	*critical-blood-status(P) VALID-TIME [now – (5, day), now]*
THEN	*drop(Drug-Administration[drug = "ETOPOSID"], P)* *Unless normal-blood-status(P)* *VALID-TIME now*

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.

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

WHEN INSERT ON blood-findings (VII)
WITH new.parameter = Leukocyte-Count AND new.value < 1000

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* [8,34]. 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.

$TIME\text{-}SERIES(E, n, d_{min}, d_{max}, I)$ occurs	iff	it exist $t_1 < t_2 < ... < t_n$, $t_i \in I$ with: $d_{min} \leq \lvert t_i - t_{i+1} \rvert \leq d_{max}$, $i = 0, ..., n$ E occurs at every t_i.

The point in time at which an instance of $TIME\text{-}SERIES(E, n, d_{min}, d_{max}, I)$ 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 (VII). Then,

WHEN TIME-SERIES(E, 3, (2, day), (4, day), [now, now + (2, week)]) (VIII)

occurs if during two weeks the leukocyte count of a patient is less than 1000 at 3 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} respective 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 control action. To reduce language complexity and to facilitate the handling of control flow failures only *one* control action is allowed in the action part. For the conjunction of two control actions, *two* active rules have to be defined which are both triggered by the same event and where each rule triggers one of the two control actions.[4]

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 (*A* and *A'* denote activity patterns, and *CS* denotes a case).

- *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-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*.
- *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).
- *review(A,CS)*: For *CS*, every execution of an *A*-activity has to be reviewed by a user (manual control).

3.3. Rule integrity

Generally, rule integrity covers rule incompatibility and rule termination [40]. Concerning *rule termination*, our approach does not add any additional complexity to rule processing as known from active databases and temporal logics, so that we refer to [6,12,40].

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 inter-

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

	drop(A)	*replace(A,A')*	*postpone(A,d_2)*	*add(A)*
drop(A)	CP	CP*	ICP	ICP
replace(A,A')		CP	ICP	CP**
postpone(A,d_1)			ICP for $d_1 \neq d_2$	CP***
add(A)				CP

ICP = Incompatible, *CP* = Compatible, *A, A'*= activity pattern, d_i = duration

* *replace* is weighted stronger, i.e., *A*-activities are not only dropped but replaced by *A'*-activities

** Processed in order *add* → *replace*, i.e., first the new *A*-activity is added, then it is replaced together with already existing *A*-activities

*** Order to be determined manually at execution time.

Tab. 1: Incompatibility table for local control actions (with overlapping valid time intervals).

The case parameters have been omitted as incompatibilities only have to be considered if two actions refer to the same case (e.g., the same patient). The *review* action is not listed as it has to be manually transformed to one of the other actions. The *add-repetitively* action is handled analogously to the *add* action.

vals. To cope with this, AGENTWORK uses incompatibility tables such as the one shown in Tab. 1 (there also exist tables for global control actions and combinations of local and global control actions [35]). For example, the pair

add(A), *replace(A,A')* (IX)

is viewed as compatible, as it may be necessary to add an activity and then to replace it directly. As an example, imagine two rules where the first one states that drug *A* has to be administered in case of a particular infection, and where the second one states that in case of an allergy w.r.t. *A* this drug has to be replaced by drug *A'*. If the patient suffers from this particular infection *and* has an allergy w.r.t. *A*, *add(A)* and *replace(A,A')* would be triggered simultaneously, and would have to be processed in the order *add* → *replace*.

Analogously, the pair

add(A), *postpone(A,d_1)*

is viewed as compatible, as an *A*-activity can be added to other *A*-activities that shall be postponed. However, if this pair is triggered the *order* in which both control actions shall be processed has to be determined (manually) at execution time, i.e., whether the *new A*-activity shall be added first and then postponed with all other already existing *A*-activities, or whether the new *A*-activity shall be added *after* the other *A*-activities have been postponed.

If incompatible control actions have been triggered simultaneously, the user has to be informed and has to resolve the situation manually.

4. Workflow adaptation and monitoring

We now describe how AGENTWORK processes triggered control actions. As the global control actions *abort* and

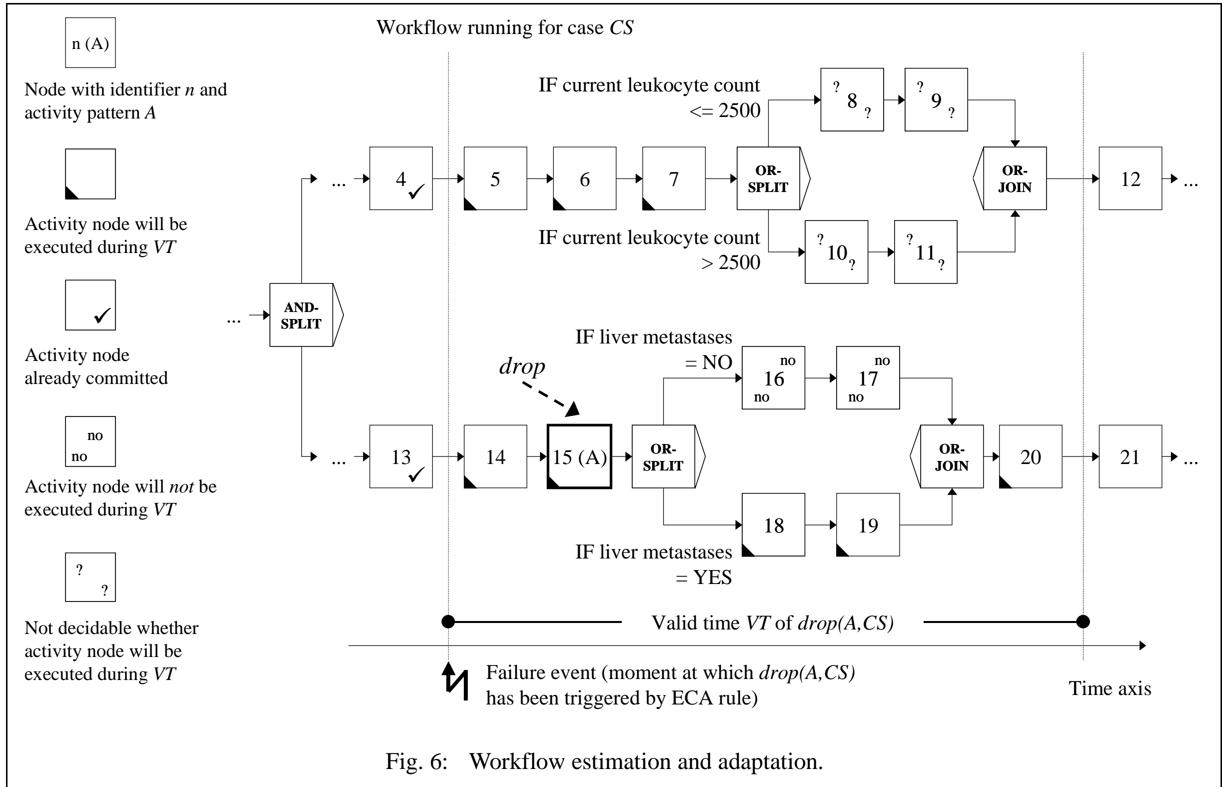

Fig. 6: Workflow estimation and adaptation.

suspend do not require workflow adaptations, we concentrate on *local* control actions. 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 the use of adaptation operators to translate control actions into structural adaptations of a workflow. Section 4.5 describes how (predictively) adapted workflows are monitored to check whether the adaptation assumptions are met by the actual execution.

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 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. 6. 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. 7 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 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. 6 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.

Since longer time intervals reduce the accuracy of the workflow estimates we differentiate between two subtypes of predictive adaptation depending on whether or not the fixed valid time interval exceeds some specified threshold value (Fig. 7). If not, the workflow part corresponding to the full valid time interval is estimated at once. Otherwise, *predictive adaptation with sub-intervals* is selected. The valid time *VT* is divided into several sub-intervals VT_1, VT_2, ..., VT_n whose durations do not exceed the threshold. Then, it is first estimated which workflow part P_1 will be executed during VT_1, and the adaptation is only applied to P_1. After P_1 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.

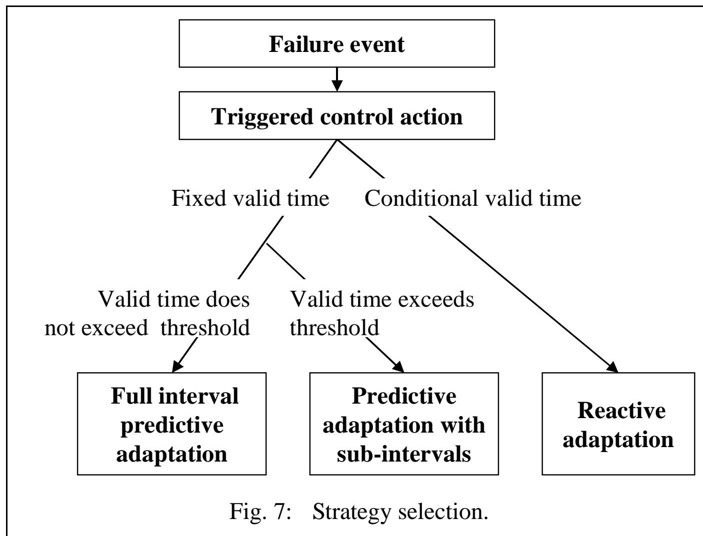

Fig. 7: Strategy selection.

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. 6 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 a *A*-node. As node 15 is such a *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. 6 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).

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

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 internal data flow edges. Instead the estimates are based on duration estimates for the execution of activity nodes and on duration estimates for data flow edges reading external data. AGENTWORK supports two ways to obtain such duration estimates: They can either be specified by the workflow modeler during workflow definition or they are derived from measurements during workflow execution. To support a high estimation accuracy the durations can be grouped according to different dimensions. Thus we can have different values per activity type and data source, e.g., to differentiate between user types to account for the fact that a beginner may need substantially more time for an activity than a sophisticated user.

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.

Estimation of $\mathbf{P_{VT}}$. 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 would have to be executed next (e.g., nodes 5 and 14 in Fig. 6). 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 unresolvable 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, and 7 in Fig. 6) is estimated by summing up the average execution durations of all its activities and data flow edges.

In the case of a *parallel* block (AND-SPLIT/AND-JOIN) estimation is performed for each of the paths starting at the AND-SPLIT node and the duration of the whole block is set to the maximum of the estimated durations of all its paths.

Particular problems for estimation arise w.r.t. *conditional* blocks (OR-SPLIT/OR-JOIN) as they require data dependent prediction of the paths that will qualify for execution when the workflow is continued. The duration of the whole OR-SPLIT/OR-JOIN block is then set to the maximum of the estimated durations of the predicted paths.

Prediction of which paths will be executed within a conditional block may be possible if the data needed for the decision is already available when the control action has been triggered. For the lower OR-SPLIT in the example

of Fig. 6, it may be known at estimation time (e.g., from former examinations) that the patient has liver metastases so that it is known that only the lower path has to be considered. If no current data is available, execution probabilities of previous workflow executions can be used to determine the most likely paths to be considered. If some conditions cannot be evaluated at all 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 these parts. In Fig. 6, this was assumed to be the case for the upper OR-SPLIT, as leukocyte counts may change significantly during a few days so that former blood examinations cannot be considered for a predictive evaluation of the leukocyte condition.

Similar problems arise w.r.t. the estimation of *loops*. 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. However, the exact number of loop iterations mostly depends on data which is produced during an iteration. 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 previously executed workflows.

If the control action has been triggered during loop execution it is tried to resolve the loop's termination condition in a similar way as it is done w.r.t. OR-SPLITS to predict if the loop will be executed again. This may be possible if the required data has been produced already. Otherwise estimation stops at the LOOP-END node and AGENTWORK switches to reactive adaptation for the further loop iterations and all later workflow parts of this path.

Recently, several other workflow estimation approaches have been suggested to support tasks such as deadline management and scheduling for workflows [15,25,33]. 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 control flow operator for each control action introduced in 3.2.3. We sketch only the operators *drop-node* and *add-node* (corresponding to the *drop* and *add* control actions), as the other operators can be mapped to variations and combinations of these two. A node *replacement* is achieved by dropping a node and adding another one at the same position, while a node postponement is achieved by dropping a node and inserting it at a later position of the workflow. The operators are used for both predictive and reactive adaptation.

Operator for node dropping. For node dropping, the operator *drop-node (n: Integer)*

is provided. This operator takes as input the identifier *n* of an activity node to be dropped. The effect of this operator depends on the particular structure of the workflow part to which the affected node *n* belongs to:

a) If *n* is located in a sequence, it is simply removed from the control flow. Incoming and outgoing control flow edges are merged together, 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.

Operator for node adding. For node adding, the operator *add-node(A: Activity-Pattern, n: Integer)*

is provided. The first parameter specifies the activity pattern *A* that shall be assigned to the new node. 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.

Concerning the effect on the control flow, we distinguish two principal mechanisms of *add-node* to add a *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. 8 a). The main disadvantage of sequential add is that it may delay the execution of successor nodes of *n* (e.g., node 2 in Fig. 8 a) more than necessary.

Parallel add: To minimize execution delays of successor nodes of *n*, parallel add inserts the new node into a new parallel path. In the example of Fig. 8 b, a new *A´*-node 4 has been inserted into a new empty path parallel to nodes 1 and 2. AGENTWORK tries to use temporal estimates to optimize a new AND-SPLIT/AND-JOIN block so that a new parallel path is not longer than the execution duration of the other parallel path consisting of already existing nodes. For this reason, in Fig. 8 b 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.

AGENTWORK by default uses parallel add for implementing *add-node.* Only if a temporally optimized parallel add is not possible, e.g., when the needed temporal estimations cannot be performed, the new node is sequentially added.

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 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.

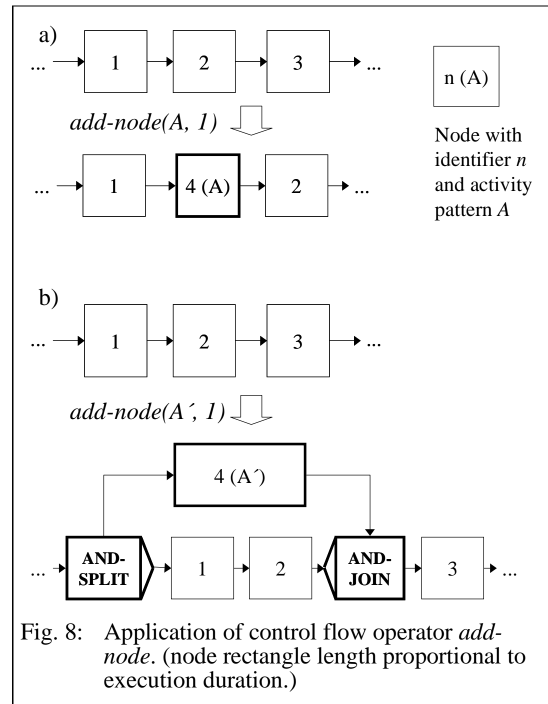

Fig. 8: Application of control flow operator *add-node*. (node rectangle length proportional to execution duration.)

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 of a newly added node may have to be provided. Thus, appropriate data flow edges have to be generated to provide needed 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.

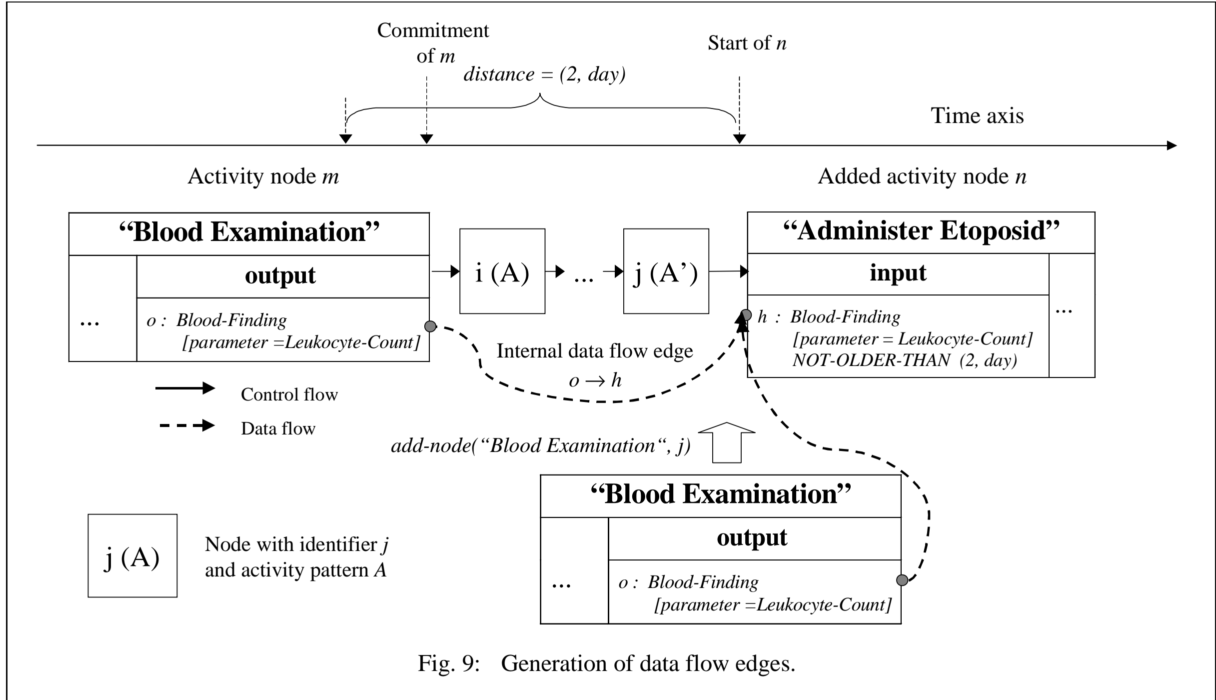

Fig. 9: Generation of data flow edges.

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. 9 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. 9) 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 resp. *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. 9, 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.

If the local temporal neighborhood of the workflow does not provide such a suitable output object *o*, AGENT-

WORK inserts an activity node providing the needed data object (due to the activity node's output specification). For example, in Fig. 9 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 [35].

4.5. Workflow monitoring

Workflow monitoring is used in the context of predictive adaptation to check whether the estimated execution duration of P_{VT} matches the actually needed time. The estimates may prove inaccurate for a variety of reasons, such as additional adaptations (e.g., adding or dropping nodes), additional delays for activity executions or external data accesses (e.g., due to system failures), or inaccurate predictions for predictively handled OR-SPLIT and loop blocks (4.2). These aspects result in two mismatch types w.r.t. the time estimates, namely temporal *acceleration* and temporal *delay*.

Temporal acceleration occurs if P_{VT} is executed faster than it has been estimated. This means that workflow parts after P_{VT} (e.g., nodes 12 and 21 in Fig. 6) will also be executed during *VT* and therefore further nodes not considered so far may have to be adapted (e.g., dropped or added).

In case of a **temporal delay** parts of P_{VT} will not be executed during *VT* anymore as the execution of some nodes has taken unexpectedly long (e.g., node 20 in Fig. 6 may not be executed anymore during *VT* as the execution of node 14 may haven taken unexpectedly long). Therefore, adaptations of these parts may have to be revoked, i.e., dropped nodes have to be added again or added nodes have to be dropped for example.

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 [23,42,20,31,7,10,41,43]. However, only a few commercial systems such as PROMINAND [23], ACTION REQUEST SYSTEM [42] or LOTUS DOMINO WORKFLOW [20] provide some support for workflow adaptation. For example, ACTION REQUEST SYSTEM [42] 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 focussed on workflow recovery from system failures, typically guided by an advanced transaction model supporting compensation and forward recovery [2,21,39,47]. These approaches do not deal with changing the structure of running workflows to handle logical failures.

The TAM[6] system [31,49] 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 $\text{ADEPT}_{\text{flex}}$ system [41] 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 [43], 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 [36]. 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 [4,45,30,22,38] and cooperative agent approaches [14,24,28,39,46]. 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 provide a sophisticated way to detect and handle failures (e.g., [28]), but do not address the structural consequences of failure handling on the graph level.

---

[6] TAM = Transactional Activity composition Model

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 are currently implementing a prototype of the AGENTWORK system. As a core, we use the $ADEPT_{flex}$ workflow management system [41] for the *workflow definition and execution layer* of AGENTWORK. $ADEPT_{flex}$ 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., [18]), 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). 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 $ADEPT_{flex}$ workflow model. The adaptation itself has been realized by directly invoking the $ADEPT_{flex}$ control and data flow operators using the $ADEPT_{flex}$ API. The workflow monitoring agent has been implemented in JAVA, too.

Large parts of AGENTWORK have already been implemented and show the feasability of our failure handling approach. The implementation is currently completed within an interdisciplinary medical project at the University of Leipzig. 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". We will also evaluate the applicability of the approach in different application domains such as e-business.

Acknowlegements

We thank the Database Section (Head: Prof. Dr. Peter Dadam) of the Department of Computer Science, University of Ulm, Germany, for kindly providing the $\text{ADEPT}_{\text{flex}}$ prototype for us.

References

[1] G. Alonso and C. Mohan, WFMS: the next generation of distributed processing tools, in: S. Jajodia and L. Kerschberg, eds., Advanced Transaction Models and Architectures (Kluwer, 1997) 35-62.

[2] V. Atluri, W-K. Huang and E. Bertino, A semantic based execution model for multilevel secure workflows, Journal of Computer Security 8(1) (2000) 3-41.

[3] S. Baker, CORBA distributed objects (Addison Wesley, 1997).

[4] C. Beckstein and J.Klausner, A meta level architecture for workflow management, Journal of Integrated Design and Process Science 3(1) (1999) 15-26.

[5] J. Benthem, Temporal logic, in: D.M. Gabbay, C.J. Hogger and J.A. Robinson, Handbook of logic in artificial intelligence and logic programming, Volume 4: Epistemic and temporal reasoning (Oxford University Press, Oxford, UK, 1995).

[6] E. Bertino, D. Montesi, M. Bagnato and P. Dearnley, Rules termination analysis investigating the interaction between transactions and triggers, in: Proc. IDEAS'02 (IEEE Computer Society, 2002) 285-294.

[7] F. Casati, S. Ceri, S. Paraboschi and G. Pozzi, Specification and implementation of exceptions in workflow management systems, ACM TODS 24 (1999) 405-451.

[8] S. Chakravarthy, V. Krishnaprasad, E. Anwar and S.-K. Kim, Composite events for active databases: semantics contexts and detection, in: Proc. VLDB'94 (Morgan Kaufmann, 1994) 606-617.

[9] D.K.W. Chiu, Q. Li and 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: Proc. of ER 2001, Lecture Notes in Computer Sience, Vol. 2224 (Springer, Berlin, 2001) 541-554.

[11] J. Chomicki and G. Saake, Logics for databases and information systems (Kluwer, New York, 1998).

[12] J. Chomicki and D. Toman, Temporal logic in information systems, in: J. Chomicki and G. Saake, eds., Logics for Databases and Information Systems (Kluwer, New York, 1998) 31-70.

[13] P. Dadam, M. Reichert and K. Kuhn, Clinical workflows - the killer application for process-oriented information systems? in: Proc. 4th International Conference on Business Information Systems (Springer, Berlin, 2000) 36-59.

[14] S. M. Deen, Cooperating agents for holonic manufacturing, in: Proc. Multi-Agent-Systems and Applications (2001) 119-136.

[15] J. Eder, E. Panagos and M. Rabinovich, Time constraints in workflow systems, in: Proc. CAiSE 1999 (Springer, Berlin, 1999) 286-300.

[16] E. Ehud Gudes, Martin S. Olivier and R.P. van de Riet: Modeling, specifying and implementing workflow security in cyberspace, Journal of Computer Security 7(4) (1999).

[17] E. Franconi, Description logics for natural language processing, in: F. Baader, D.L. McGuinness, D. Nardi and P.F. Patel-Schneider, eds., Description Logics Handbook (Cambridge University Press, Cambridge, 2002).

[18] F. Frohn, R. Himmeröder, P.-Th. Kandzia, G. Lausen, C. Schlepphorst, FLORID - A prototype for F-Logic, in Proc. 7th Bi-Annual German Database Conference (BTW'97) (Springer, Berlin, 1997) 100-117.

[19] D. Georgakopoulos, M. Hornick and 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.

[20] G. Giblin and R. Lam, Programming workflow applications with Domino (R&D Publications, 2000).

[21] P.W.P. J. Grefen, J. Vonk and P.M.G. Apers, Global transaction support for workflow management systems: from formal specification to practical implementation, VLDB Journal 10(4) (2001) 316-333.

[22] K.J. Hammond, Explaining and repairing plans that fail, Artificial Intelligence 45 (1990) 173-228.

[23] IABG, Reference manuals of ProMInanD (IABG Company, Munich, 1999).

[24] N.R. Jennings, P. Faratin, T.J. Norman, P. O'Brien and B. Odgers, Autonomous agents for business process management, International Journal of Applied Artificial Intelligence 14 (2) (2000) 145-189.

[25] E. Kafeza and K. Karlapalem, Temporally constrained workflows, in: Proc. ICSC 1999, Lecture Notes in Computer Science, Vol. 1749 (Springer, Berlin, 1999) 246-255.

[26] M. Kifer, G. Lausen and J. Wu, Logical foundations of object-oriented and frame-based languages, Journal of the ACM 42 (1995) 741-843.

[27] B. Kiepuszewski, A.H.M. ter Hofstede, C. Bussler, On structured workflow modelling. Proc. CAiSE'2000, Lecture Notes in Computer Science, Vol. 1789 (Springer, Berlin): 431-445.

[28] M. Klein and C. Dellarocas, A knowledge-based approach to handling exceptions in workflow systems, Journal of Computer-Supported Collaborative Work 9(3/4) (2000) 399-412.

[29] A. Lazcano, H. Schuldt, G. Alonso and H.-J. Schek, WISE: process based e-commerce, IEEE Data Engineering Bulletin 24(1) (2000) 46-51.

[30] C. Liu and R. Conradi, Automatic replanning of task networks for process model evolution in EPOS, in: Proc. ESEC'93 (1993) 434–450.

[31] L. Liu and C. Pu, Methodical restructuring of complex workflow activities, in: Proc. ICDE 1998 (IEEE Computer Society Press, 1998) 342-350.

[32] Z. Manna and A. Pnueli, The temporal logic of reactive and concurrent systems (Springer, Berlin, 1992).

[33] O. Marjanovic and M.E. Orlowska, On modeling and verification of temporal constraints in production workflows, Knowledge and Information Systems 1 (1999) 157-192.

[34] I. Motakis and C. Zaniolo, Temporal aggregation in active database rule, in: Proc. SIGMOD 1997, SIGMOD Record 26(2) (1997) 440-451.

[35] R. Müller, Event-oriented dynamic adaptation of workflows, Ph..D. Thesis, University of Leipzig, 2002.

[36] R. Müller and B. Heller, A petri net-based model for knowledge-based workflows in distributed cancer therapy, in: Proc. EDBT'98 Workshop on Workflow Management Systems (1998) 91-99.

[37] R. Müller and E. Rahm, Dealing with logical failures for collaborating workflows, in: Proc. CoopIS 2000, Lecture Notes in Computer Science, Vol. 1901 (Springer, Berlin, 2000) 210-223.

[38] K. Myers, Towards a framework for continuous planning and execution, in: Proc. of the AAAI Symposium on Distributed Continual Planning, (1998).

[39] K. Nagi, J. Nims and P.C. Lockemann, Transactional support for cooperation in multiagent-based information systems, in: Proc. vertIS2001 (2001) 177-191.

[40] N. Paton, ed., Active rules in database systems (Springer, 1999).

[41] M. Reichert and P. Dadam, $\text{ADEPT}_{\text{FLEX}}$ - supporting dynamic changes of workflows without losing control, Journal of Intelligent Information Systems 10 (1998) 93-129.

[42] Remedy Corporation, Action request system 4.0 reference manuals (Remedy Corporation, 2000).

[43] S.W. Sadiq, W. Sadiq and M.E. Orlowska, Pockets of flexibility in workflow specification, in: Proc. ER 2001, Lecture Notes in Computer Science, Vol. 2224 (Springer, 2001) 513-526.

[44] A. Sheth, K. Kochut et al., Supporting state-wide immunization tracking using multi-paradigm workflow technology, in: Proc. VLDB 1996 (Morgan Kaufmann, 1996) 263-273.

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

[46] W. de Vries, F.S. de Boer, W. van der Hoek and J.-J. Ch. Meyer, A truly concurrent model for interacting agents, in: Proc. PRIMA 2001, Lecture Notes in Computer Science, Vol. 2132 (Springer, 2001) 16-30.

[47] H. Wächter and A. Reuter, The ConTract model, in: A.K. Elmagarmid, ed., Database Transaction Models for Advanced Applications (Morgan Kaufmann, 1992) 219-263.

[48] D. Worah and A. Sheth, Transactions in transactional workflows, in: S. Jajodia and L. Kerschberg, eds., Advanced Transaction Models and Architectures (Kluwer, 1997) 3-34.

[49] T. Zhou, L. Liu and C. Pu, TAM: a system for dynamic transactional activity management, in: Proc. SIGMOD Conference 1999 (1999) 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. In the summer term 2003, he holds a deputy professorship for databases at the Department of Computer Science of the University of Munich. His current research topics include workflow management, data integration, and bioinformatics.

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 Kaiserlautern, 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.