Design for Change: Evolving Workflow Specifications in ULTRAflow

Alfred Fent, Herbert Reiter, and Burkhard Freitag*

University of Passau, Department of Computer Science
94030 Passau, Germany
Tel. (+49) 851 509 3131, Fax (+49) 851 509 3182
{fent,reiter,freitag}@fmi.uni-passau.de

Abstract. Updating the specification of workflows on the fly in a workflow management system is currently considered an important topic in research as well as application. Yet, most approaches are either very simplistic, allowing only newly started workflows to take advantage of updated specifications, or they are complex, trying to transfer every active workflow from the old to the new schema.
In the workflow management system ULTRAflow, updates to workflow specifications are handled by using a multi-version concurrency control protocol. This is facilitated by the specification language for workflows, which is rule based and therefore provides a natural partitioning of specifications into smaller units. The proposed method allows active, running workflows to partly use new specifications if this does not conflict with already executed sub-workflows. Moreover, an architecture which is also applicable in a distributed system is presented.
While the method to update the specifications is discussed in the context of a workflow management system, it can also be applied in CORBA or EJB applications, or the now ubiquitous electronic services.

1 Introduction

Updating the specification of workflows in a workflow management system (WfMS) is currently considered an important topic in research as well as application. Of special interest are procedures to perform these updates on the fly, i.e., without disturbing the execution of active workflows in the system. However, many approaches, especially those applied in commercial systems, are very simplistic, allowing only newly started workflows to take advantage of the updated specification, while active workflows still use the original one; other approaches that try to transfer every running workflow instance from the old to the new schema often are complex and complicated [26].

The way our workflow management system ULTRAflow [10,11,12] handles evolving workflow specifications lies between these two extremes. A multi-version concurrency control protocol is employed to control access to the changed parts

* The work described in this paper has been funded by the German Research Agency (DFG) under contract number Fr 1021/3-3.

A. Banks Pidduck et al. (Eds.): CAISE 2002, LNCS 2348, pp. 516–534, 2002.

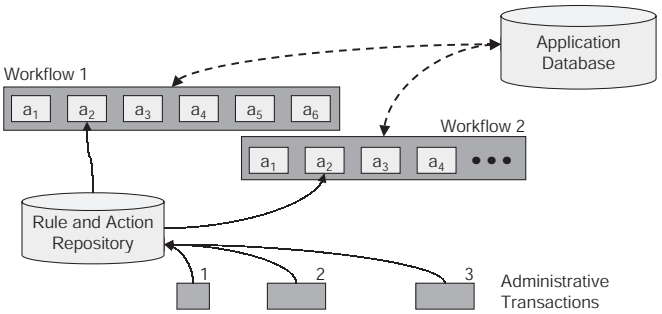

Fig. 1. Changes of specifications are also considered transactions

of a workflow specification. This is facilitated by the specification language of ULTRAflow, which is rule based [13,27,28] and therefore provides a natural partitioning of specifications into smaller units. The proposed method allows active, running workflows to partly use new specifications if this does not conflict with sub-workflows or basic operations already executed. This is not possible in known versioning schemes where a "snapshot" is taken at the beginning of the workflow. It is also applicable in a distributed environment.

The general situation is depicted in Fig. 1: several workflows are running concurrently, each consisting of steps $a_1, a_2, \ldots$ which can be basic operations or sub-workflows that are again defined in the ULTRAflow language. While a workflow is executing – which typically takes some non-negligible time, i.e. several hours, days, or weeks – it retrieves the specifications of its steps from a repository for rules and actions. In the meantime, some of the specifications of steps are updated in the repository by administrative transactions to account for new technologies, changes in business rules, new processes, etc. Access to the rule- and action repository is handled through a multi-version concurrency control protocol. This protocol protects the retrieval of steps by workflows from updates of the steps by administrative transactions and controls which of the workflows can use an updated specification, and which has to stick to the old one. This way, it eliminates the "dynamic change bug" [9,26], i.e., errors introduced by changing specifications of active workflows.

Within the transactions formed by the workflows each step can execute arbitrary actions (represented by dotted lines in the figure) on, e.g., an application database. Correctness of these actions with several workflows running concurrently is ensured by techniques well-known from databases [1] but not in the scope of this paper.

A solution for the dynamic change bug must meet the following requirements:

1. No errors or failures of active workflows are caused due to the change
2. If a new specification for a workflow is provided, other workflows that use it as a sub-workflow do not have to be modified to continue functioning

3. Updating a specification does not cause the WfMS to stop execution of running workflows, nor to delay starting new ones
4. The solution must be applicable within a system that isolates workflows with some kind of transactions and implements recovery by compensation [17]

Requirements 1 and 2 are derived from [9], while number 3 is more than reasonable in a real-life WfMS. Requirement 4 is mainly motivated by our specific setting, as ULTRAflow uses compensation for recovery, i.e., it performs semantically reverse actions to undo the effects of aborted transactions. In systems that do not rely on compensation, our solution can be implemented even more easily.

Although in this paper the method to update and control access to specifications is presented in the context of a WfMS, it can also be applied in other situations where parts of a (possibly long running) program have to be exchanged on the fly. This happens in application servers for, e.g., CORBA [20] or Enterprise Java Bean (EJB) applications [25], when new versions of the program code are deployed and integrated into the system. This is frequently referred to as "hot deployment". We performed some tests with Apache/Tomcat [2] which can be seen as a minimal application server. Today, this combination detects new versions of code, however it then delays new requests until all instances of the old code which are still active are finished, loads the new code, and continues handling requests with the new version then. The delay could be eliminated with our method. The full-fledged EJB application server Orion [21] behaved similarly in our tests; it even exchanged definitions used several times within still active transactions, thus stumbling directly into the dynamic change bug [9].

Also, the now ubiquitous electronic services (e-services) [22] can profit from our method. In e-services, it is usually assumed that at the beginning of an application the necessary services are searched and decided upon, and that they are used throughout the application. Yet, this may impose rather long setup times at the start of the application or at the first invocation of a service if the search is delayed until this moment. So, systems will begin to "cache" this information and reuse the same e-service for several applications, providing some kind of "e-service application server" which will run into the same issues as described above.

The rest of the paper is organized as follows: we start with overviews of related work and the ULTRAflow system in Sect. 2 and 3, before we detail on updating specifications in Sect. 4 and ensuring correctness in Sect. 5. Practical aspects of our architecture are presented in Sect. 6. The paper is concluded with a summary in Sect. 7.

2 Related Work

Modification of workflow specifications has been quite frequently addressed in recent research, so we will only mention a selection of the related work here. Many other articles concerning adaptive workflows can be found e.g. in [3,16].

The so-called "dynamic change bug" was introduced in [9], where also correctness criteria for workflow evolution were proposed; however, the paper does

not discuss what to do with instances that do not meet these criteria. This is done in [6], where primitive operations on workflow specifications are defined. The paper also discusses "progressive policies" dependent on the state of active instances which can either continue with the initial specification, be migrated to the new one under certain conditions, migrated to an ad hoc workflow, or be aborted. Our solution is in fact a mixture of the first two policies: parts of the workflow are migrated, and parts continue with the old version.

"Ad hoc changes", i.e., changes performed as part of the workflow itself, are the topic within $\text{ADEPT}_{\text{flex}}$ [23]. While this is important, our work deals with an evolution of the specifications driven from outside the running workflow instances. This is the typical case for planned workflows that are specified and installed by a workflow administrator. Approaches to transform running instances of an old to instances of a new specification are discussed in [8,19], and an infrastructure based on language reflection that supports such transformations is presented in [7]. Our approach, in contrast, is not based on explicit transformation, but uses new specifications automatically if they do not conflict with those already used by the workflow.

The determination of a so-called "change region", i.e., the part of a workflow that is affected by an update, is the topic of [26]. Workflow instances whose executions are currently not within this region immediately use an updated specification, while the updates are postponed for other instances until they leave the region. However, the algorithm of [26] to compute the change region has a complexity in $O(n^4(n!)^2)$, where n is the number of nodes of the workflow. Moreover, if compensation is used the change region may include the rest of the workflow, such that the anticipated advantage of minimizing the number of versions is not achieved.

Work on handling exceptions and using compensation [15] is related to our approach in so far as we also use compensation in the workflows within ULTRAflow. Yet, changes to workflow specifications or basic operations are not addressed in [15].

3 The ULTRAflow System

ULTRAflow [10,11,12] is a WfMS which is based on a rule-based specification language. It is implemented as a prototype that is integrated into a web-server as a lightweight component, thus leveraging the access to the system from almost arbitrary client systems, having HTML as the least common denominator. The overall architecture is shown in Fig. 2. Actors, i.e., usually humans interacting with the WfMS, can access the system from within a company intranet or from the internet, where in the latter case either some kind of virtual private network (VPN) or the authorization and authentication features of the web-server are used to restrict admission to the system. Part of the ULTRAflow system is the rule engine [13,27,28], which executes the workflow specifications. These specifications also include information about task assignment to different actors or access to other systems (like organisation information or production data, but

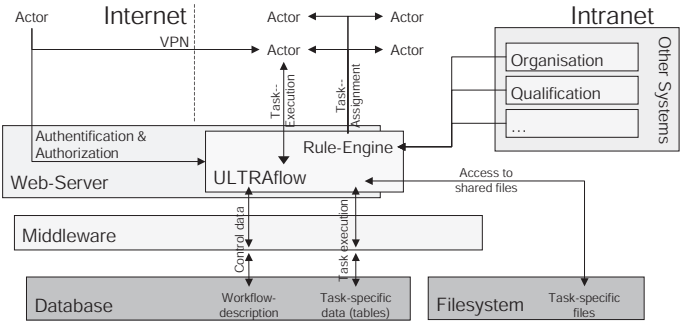

Fig. 2. The overall architecture of the ULTRAflow system

also databases etc.). General control data, as well as task-specific data (information about work progress, status, etc.) are accessed through a middleware layer. Moreover, files stored in a filesystem can be integrated into the workflow execution.

Although actors usually are human workers, other systems calling a service provided by ULTRAflow are also supported, such that the WfMS can be integrated into, e.g., an e-service infrastructure [24]. However, in this case methods for authentication and authorization specifically developed for e-services may be more appropriate [22].

The rule engine works on workflow specifications that are expressed in the action language of ULTRAflow, which itself is an instance of the general ULTRA framework [27,28]. Basic operations, like calls to other applications, posts on user- or role-specific blackboards, etc., can be composed into more complex specifications, which then can be reused in other rules. The rule language is expressive enough to describe most patterns that arise in workflow specification [10]. Features like handling of sub-workflows are inherent to the language, and formal methods known for rule languages can be applied. For instance, the dependency graph [18] reflects the workflow/sub-workflow relation.

During the execution of a workflow in the ULTRAflow system, the rules defining the workflow are "expanded" on demand in resolution-like steps [13,18]. This is in contrast to many other WfMSs where the complete workflow specification is instantiated (e.g., in the form of a graph or Petri net) when a new instance of a workflow is started.

Example 1. As an example for rule resolution, look at the following part of a simple workflow specified in the rule language of ULTRAflow:

$$send(P, D, B) \leftarrow prepare\_shipment(P) \ : \ [ship(P, D) \mid send\_bill(B)].$$
$$prepare\_shipment(X) \leftarrow collect\_items(X) : package(X).$$

The program describes sending a parcel P to delivery address D and billing address B. In ULTRAflow, a colon (":") denotes sequential composition, while concurrent execution is denoted by a vertical bar ("|"); where necessary, brack-

ets "[" and "]" indicate precedence. As usual in logic programming, variables are denoted by capital letters. To send the parcel, first the sub-workflow *prepare_shipment* is executed, then the parcel is *ship*ped to the delivery address, while concurrently the bill is sent to the billing address. The specification of sub-workflow *prepare_shipment* for a parcel X consists of the activities to *collect_items* for X that are to be sent, and to *package* them together.

If this workflow is executed, the system proceeds just as described above. For every subgoal on the right hand side of a rule, the adequate definition is looked up, which either yields another rule (like in *prepare_shipment*) or a class in Java, the implementation language of ULTRAflow, in case of a basic operation.

When errors occur during the execution of a workflow, like logical failure in rule evaluation [18], transactional conflicts, etc., compensation [17] is used to rollback the execution state to the last existing choice point. The existence of a compensation method is mandatory for basic operations and optional for rules defining sub-workflows.

In the architecture of Fig. 2 there is only one block called ULTRAflow; however, this block can be implemented as a distributed system which spreads the load of workflow execution over several nodes according to some schema based on availability of resources, physical proximity, or other distribution aspects. Whether a rule specifying a sub-workflow is executed at the same or a remote system is transparent to the caller.

4 Updating Workflow Specifications

Before we analyze how a system might evolve, we first describe workflow specifications in more detail.

4.1 Components of a Workflow Specification

As ULTRAflow uses a rule based specification language, every workflow and sub-workflow specification is represented by a number of rules. An entire workflow instance is started by issuing a (top-level) query, whereas the start of a sub-workflow is caused by a sub-query within the specification of the superordinate workflow. The call to a sub-workflow and also to a basic operation is just a name and a number of arguments. However, the WfMS itself needs additional information to be able to call the sub-workflow and to ensure correctness of the execution in the application level sense. All this information can be divided into the following parts:

Syntactic Information includes everything the WfMS needs to identify the specific basic operation or workflow which is called. In the ULTRAflow system, this is currently just the name, but additional information like name spaces or module names may be present, too.

Interface Definition describes the arguments of the sub-workflow or basic operation, their number and types, etc.

Semantical Context covers everything that the WfMS needs to ensure correctness during the execution. In the ULTRAflow system, this includes a compatibility matrix, a description of transactional features (e.g., support for two phase commit [4,5,14]), and recovery aspects for basic operations. For complex workflows, the semantical context may also contain recovery data (name of a compensating workflow), estimated execution times for sub-workflows and actions, etc. In other words, the semantical context of a step comprises all the meta data available to the WfMS.

Program Code is called when the operation or workflow is executed, i.e., it is the actual implementation. For basic operations the program code is given by the name of a Java class (cf. Sect. 6). As compensation [17] is used for recovery (see requirement 4 in Sect. 1), this class must provide forward (do) and backward (undo) methods. For complex (sub-)workflows, the program code consists of the defining rules for the predicate corresponding to the workflow; providing an undo-workflow is optional, as the effects can always be removed by compensating the basic operations. So, in short, "program code" includes everything that is needed to perform the step or to remove its effects in case of (transactional) failure.

Definition 1. *A* (specification of a) step *is a tuple* $a = (N, I, S, C)$ *consisting of name* N*, interface* I*, semantical context* S*, and code* C*. A step can be a basic operation or a sub-workflow that is defined by a rule in the* ULTRAflow *language. The set of all steps known to the system is called* P*.*

Example 2. The specification of step *prepare_shipment* of Ex. 1 is the tuple

$$\begin{aligned} N &= \text{"}prepare\_shipment\text{"} \\ I &= X \quad \text{plus adequate type information} \\ S &= \text{representation of semantical context} \\ C &= \{collect\_items(X) : package(X)\} \end{aligned}$$

The set of all steps is as follows, with tuples abbreviated to their names:

$$\begin{aligned} P = \{&(\text{"}send\text{"}, \ldots), (\text{"}prepare\_shipment\text{"}, \ldots), (\text{"}ship\text{"}, \ldots), \\ &(\text{"}send\_bill\text{"}, \ldots), (\text{"}collect\_items\text{"}, \ldots), (\text{"}package\text{"}, \ldots)\} \end{aligned}$$

4.2 Adding New Specifications

Adding new specifications to the system is relatively easy from the theoretical point of view and poses only engineering issues. We can simply define

Definition 2. *A step* $a' = (N', I', S', C')$ *is* new (in P) *if its name is not contained in* P*, i.e.,* $\forall a = (N, I, S, C) \in P : N \neq N'$*. Adding a new step* a' *to* P *is done by insertion, i.e.,* $P' = P \cup \{a'\}$ *where* P' *is the new set of steps.*

This definition which is used in the ULTRAflow system only relies on the name of the operation. Alternatively it would be possible to also take the interface

definition into account like in object-oriented languages and define a' as new if $\forall a = (N, I, S, C) \in P : (N \neq N' \wedge I \neq I')$. Yet, it does not make a difference in the following presentation which is why we stick to the simple definition based only on the names of predicates.

In the following, we identify the step $a = (N, I, S, C)$ with its name N.

4.3 Deleting an Existing Specification

It may also happen that the specification of a step is removed from the system. However, we do not discuss this here for a simple reason: requirements 1 and 2 of Sect. 1 said that the specifications of other workflows in the system must not be altered because of an update, and that no errors may occur due to it. Now, if we removed a step that is used in another workflow, this other workflow will no longer work, i.e., it will cause runtime errors. So our requirements would be violated. If, on the other hand, the step to delete is not used in any other workflow specification, it is only "dead code" and therefore can be removed, anyway.

4.4 Updating an Existing Specification

As described in the previous sections, adding new and deleting existing steps are more or less straightforward and do not pose a lot of problems. This is no longer the case when we consider updating the specification of an existing step. We then have to analyze three cases, as three parts of the step can be changed: interface, code, and semantical context.

Changing the Interface As mentioned in Def. 2 of Sect. 4.2, the interface is not used to identify the step itself, but only the predicate name. So it is possible to provide a new version with a changed interface. Yet, as one of our basic requirements is that the specifications of workflows using the updated step have to remain unchanged (see Sect. 1) this precludes a change to the interface; otherwise, runtime errors would occur. If a changed step really needs other arguments, mappings from the old to the new interface have to be provided within the implementation code, such that the visible interface stays unchanged. Otherwise, a new step with the new interface has to be added to the system.

Changing the Implementation This is the most frequent case: the overall syntactical and semantical appearance of the step stays the same, but its code is updated to reflect some change in the environment, to provide an optimized or corrected version of the implementation, etc. Recall from Sect. 4.1 that the code includes do- and undo-methods.

Formally, changing the specification of step $a = (N, I, S, C)$ to the new code C' corresponds to replacing P with the new set $P' = P \cup \{(N, I, S, C')\} \setminus \{a\}$.

The notion of "code" here does not necessarily imply some kind of programming, binary data, or compiled executable. In WfMS, for example, it is quite frequent that applications like word processors or spreadsheets are started for the user. If now a company changes its corporate design and therefore replaces the standard

style sheets, letter heads, etc. of the word processor, this must also be seen as a change to the implementation code of the corresponding operation "call word processor".

Changing Semantical Information Besides the code itself, there may be additional semantical information that is provided to the system. In ULTRAflow, this information mainly covers compatibility information to facilitate scheduling operations of concurrent workflows. In other systems semantical information may comprise arbitrary data necessary for the correct execution of workflows.
Like above, changing the specification of step $a = (N, I, S, C)$ to the new semantical information S' corresponds to replacing P with the new set $P' = P \cup \{(N, I, S', C)\} \setminus \{a\}$.

5 Ensuring Correctness

The previous section analyzed the types of changes that can occur within a running WfMS. From the various possibilities (adding, deleting, and changing any of the four components a step consists of) several were ruled out because they violate the requirements of Sect. 1. Adding and deleting steps were already described, such that we are now left with the two most interesting (and challenging) cases: changing program code and changing semantical information.

5.1 Updating Implementation Code

Updates to and usage of specifications have to be synchronized, especially if (part of) the specification of an active workflow is to be updated. This obviously resembles the classical and well-known case of concurrent access to data items in a database system, except that we do not access data, but code and rules. In fact, in ULTRAflow we reuse serializability theory [4,5,14] to ensure that changes to specifications of steps do not have negative effects on active workflows. Therefore, we will introduce – in addition to transactions that the system already may use – a second layer of transactions that protect access to the specifications of steps. In other words, we consider the implementation code as the data objects that are either read or written.

To avoid having to reinvent all the definitions and theorems from database concurrency control, we observe that the execution of a workflow only reads its steps, while changes to the specification only write to it. Due to the rule based specification language of ULTRAflow we consider a workflow specification as a structured object with each rule (step) being accessible separately. This allows us to treat the execution of a workflow as a sequence of accesses to its steps.

Definition 3. *A workflow execution which executes the steps* $a_1, a_2, \ldots, a_n$ *is a transaction* T_i *which only reads these specifications, denoted by* $r_i(a_1)\ r_i(a_2)\ \ldots r_i(a_n)$.

An administrative transactions that updates the steps $b_1, b_2, \ldots, b_m$ *is a transaction* T_j *which only writes to the specifications, denoted by* $w_j(b_1)\ w_j(b_2)\ \ldots\ w_j(b_m)$.

With these definitions, we can now easily distinguish the "productive" workflows which only read and execute the steps from the administrative transactions that update them. Note that we talk about changes and updates to the steps here, i.e., we see the set P as the database and the elements of P as data items (recall Fig. 1: P corresponds to the rule and action repository). This is independent from the read and write access that the basic operations actually perform on an application database containing production data. In other words, a workflow is a sequence of reads from P but can also write data to the application database. However, it must not write to P.

Example 3. Recall the setting of Ex. 1 and that we identify the step $a = (N, I, S, C)$ with its name N. The execution of an instance of the top-level workflow *send* corresponds to the following transaction T_i (the numbers denote times and are used for reference later):

$$\begin{array}{ll} 1: & r_i(send) \\ 2: & r_i(prepare\_shipment) \\ 3: & r_i(collect\_items) \\ 4: & r_i(package) \\ 5: & r_i(ship) \\ 6: & r_i(send\_bill) \end{array}$$

Note that after each read of the specification of a step the workflow executes it. For steps like *prepare_shipment* which are defined by a rule this results in further read operations (reads of *collect_items* and *package*). Execution of basic operations like *ship* can involve access to the application database, e.g., to record the shipment.

In database concurrency control, an interleaved execution of transactions is considered correct if it has the same effects as some serial, i.e., non-interleaved, execution of these transactions (serializability, cf. [4,5,14]). We adopt this notion here and define correctness of specification changes as follows:

Definition 4. *An interleaved execution of instances of a workflow specification and updates of its steps is correct, if it is equivalent to some serial, non-interleaved execution.*

Now that we mapped the execution of workflows and changes to their steps to classical database concurrency control, we can reuse the protocols developed there.

The most frequently used concurrency control protocols are those based on locking, with two phase locking as the most prominent representative [4,5,14]. However, locking protocols have the inherent property that they may block the execution of transactions, and if this happens to one of the productive workflows, it violates requirement 3 of Sect. 1. So locking protocols cannot be used in our context.

Optimistic protocols [4,5,14] avoid blocking, but they also cannot be used here: incorrect executions are only detected at a transaction's commit time;

erroneous transactions then are aborted and restarted. In a WfMS, this is clearly unacceptable, as all the work done within the workflow, which can be worth several weeks, would be lost.

The most suitable protocols to choose in the specific situation are in fact the multi-version concurrency control protocols [4,5]. The different versions of the data object in database theory correspond to the different implementation versions in our WfMS setting. Moreover, we can exploit a simple property that is valid in this specific setting where modification of specifications is concerned: there are only two kinds of transactions, namely the productive workflows which only read the steps, and the administrative transactions that only write them. So we have only "queries" and "updaters" (in the terminology of [4]) in the system. In this case, it is possible to use a mixture of two protocols: multi-version timestamp ordering is used to synchronize queries, i.e., access to the steps, and strict two phase locking to synchronize updaters, viz., changes of the steps. This mixed protocol, which is described and analyzed in detail in [4, Chap. 5.5] and [5, Chap. 6.6], especially has the property that a query is never forced to wait for updaters and never causes updaters to wait. The scheduler can always process a query's read without delay, i.e., execution of a workflow instance is never blocked by an update of the specification. To make this paper self contained, we repeat the protocol here:

Definition 5 (Multi-version Mixed Concurrency Control with Commit Lists [4]).
When a query begins executing, a list of all the committed update transactions is associated with it, called the commit list*. The query attaches the commit list to every read that it sends to the scheduler, essentially treating the list like a timestamp. When the scheduler receives $r_i(x)$ for a query transaction T_i, it finds the most recently committed version of x whose transaction identifier is in T_i's copy of the commit list.*

Updaters use strict two phase locking, so two transactions may not concurrently create new versions of the same data item and the order of a data item's versions (and hence the version list) is well defined.

Given this organization of versions, to process $r_i(x)$ for a query transaction T_i, the scheduler scans the version list of x until it finds a version written by a transaction that appears in the commit list associated with T_i.

So, while using concurrency control to protect access to the code guarantees that requirement 1 of Sect. 1 is fulfilled, the selection of the specific concurrency control protocol also satisfies requirement 3.

Moreover, the protocol is also implementable efficiently in distributed systems. This is important in our setting, as execution of a workflow can span several systems, either by distribution of the workflow (several concurrent parts run on different systems), or by migration from one system to another (execution is transferred to another system). Using the distributed version of the protocol, we can ensure that no matter on which system a part of a workflow is executed, it always uses the correct version of its steps.

However, the protocol of Def. 5 has a disadvantage: the starting time of a query transaction fully determines all the versions that the program sees, as the commit list is generated at this time. While this is acceptable for normal database transactions, an application to updating workflow specifications would yield the simple versioning schema that is also employed in many commercial systems [26], as it corresponds to taking a snapshot at the beginning of each workflow. However, we want active workflows also to use changes to their steps that happen after their start, if they do not conflict with the work executed *so far*.

Example 4. Recall the setting of Ex. 3 and assume that *ship* is updated by a transaction T_j between times 3 and 4. Then the history reads like this:

	$\mathbf{T_i}$	$\mathbf{T_j}$
	...	
3 :	$r_i(collect\_items)$	
3.1 :		$w_j(ship)$
3.2 :		$commit_j$
4 :	$r_i(package)$	
5 :	$r_i(ship)$	

This update can be caused by implementing a new shipment procedure that is more efficient than the old one. Now we would like the active workflow T_i to take advantage of this new, improved *ship* definition. However, as the commit list of T_i was built at its starting time, it does not contain transaction T_j, and consequently the read of step *ship* at time 5 will return the old version instead of the updated one.

Therefore, we use the following extension of the protocol of Def. 5 in ULTRAflow:

Definition 6 (Multi-version Mixed Concurrency Control with Dynamic Commit Lists). *When a query begins executing, a list of all the committed update transactions is associated with it, called the* commit list*. It attaches the commit list to every read that it sends to the scheduler. When the scheduler receives $r_i(x)$ for a query transaction T_i, it finds the most recently committed version of x whose transaction id is in T_i's commit list. Moreover, a mark is set on data item x which records the read access of T_i. We denote the set of all data items which are marked at time t by transaction T_i as $mark(t, T_i)$.*

Updater transactions use strict two phase locking, so two transactions may not concurrently create new versions of the same data item and the order of a data item's versions (and hence the version list) is well defined. We denote the set of all data items that are written by a transaction T_j as $write(T_j)$. Whenever an updater commits, its transaction id is added to the commit lists of all query transactions T_i for which $mark(c, T_i) \cap write(T_j) = \emptyset$, where c denotes commit time of T_j.

In other words, the modified protocol is like the original one, only whenever an updater commits, its transaction id is added to the commit lists of all those query transactions, which did not yet read any data item that the updater wrote.

The procedures to purge old versions and to keep the commit list short that are shown in [4] can also be applied to our modified protocol. Moreover, as updater transactions do only write and not read data (i.e., implementation code), the issues of the original protocol in a distributed environment cannot arise here.

Proposition 1. *The protocol of Def. 6 creates only multi-version serializable histories.*

Proof. A formal proof of the proposition can be found in [12]. Intuitively, Def. 6 extends the protocol of Def. 5 in one respect: the commit lists are not determined and kept unmodified during the whole lifetime of a query transaction, but may grow whenever an updating transaction commits. We have to distinguish two cases:

First, for query transactions whose commit list is not modified the protocol is correct, as it then coincides with the one of Def. 5 which is known to be correct.

Second, a query transactions T_i whose commit list is modified did not read any of the data that was written by the committed updater transaction T_j up to commit time of T_j. This is the condition for the commit list to be changed. So the interleaved execution of T_i and T_j is obviously equivalent to a serial execution of T_j before the start of T_i. In the serial case, T_j would be in the commit list of T_i. So adding it in the interleaved case is allowed, too.

Note that the argumentation relies heavily on the fact that query transactions only read data, while updater transactions only write it. This precludes workflows that modify their own specification as they are investigated, e.g., in $\text{ADEPT}_{\text{flex}}$ [23].

If there are n queries and m committed updaters, $O(nm)$ modifications of commit lists are performed. However, $O(n)$ accesses are necessary even in the original protocol (Def. 5) to initially create the commit lists. As we can assume that the workflows always outnumber the administrative transactions by far, m is small in comparison to n, and the induced overhead is acceptable.

The marks set during read operations are *not* read locks and consequently do not block writes from updating transactions. They only record access to the definitions and can be used at commit time of an updater transaction to determine the commit lists to be changed: recall that updaters use strict two phase locking. When now the locks on updated data items (i.e., specifications) are released, an *exclusion list* is built containing all ids of query transactions that marked these data items. The id of the committed updater is added to the commit lists of all query transactions not in the exclusion list. As updaters usually consist of only a few writes, the exclusion list can be built efficiently.

As a side effect, the exclusion list can also be used to support workflows that must stick to the old version of a step, e.g., for legal reasons or because usage of this old version is demanded in a contract: it suffices to add the transaction ids of such workflows to the exclusion lists and they will never use a new version during their runtime.

The protocol of Def. 6 ensures correctness, but allows some committed modifications to be visible to query transactions even after their start. In our workflow context this enables active workflows to take advantage of updates to steps they did not yet use.

Example 5. Recall Ex. 4. At the time when T_j commits T_i has not yet read the new version of *ship*. So, using the new protocol T_j is added to T_i's commit list at time 3.2, and T_i will use the new version of *ship* at time 5.

Let again T_i be a productive workflow, and T_j be the administrative transaction which now additionally updates the step *package*. Let T_h be another workflow.

	$\mathbf{T_i}$	$\mathbf{T_j}$	$\mathbf{T_h}$
	...		...
3 :	$r_i(collect\_items)$		
3.1 :			$r_h(collect\_items)$
3.2 :		$w_j(ship)$	
3.3 :		$w_j(package)$	
4 :	$r_i(package)$		
4.1 :		$commit_j$	
4.2 :			$r_h(package)$
4.3 :			$r_h(ship)$
5 :	$r_i(ship)$		

Although *ship* is again read by T_i after the commit of T_j, this time it will return the old version, because T_i already used the old version of *package* (the new version was not yet committed at this time), and consequently T_j will not be added to T_i's commit list at time 4.1. So, the erroneous situation where the old version of *package* and the new version of *ship* are combined is avoided.

However, workflow T_h can use the new definitions, as it did not access any of the updated steps before T_j's commit, and so T_j was added to T_h's commit list at time 4.1. That is, although T_i read the step *ship* after T_h it will correctly see the old version, while T_h already gets the new one.

An additional problem arises through the use of compensation [17] to rollback workflows in case of failures (cf. [15]). As backward operations heavily depend on the forward operations they have to undo, it must be ensured that the correct version is selected even if the steps have been updated between the time the forward and backward operation are called. In ULTRAflow, this is achieved by packaging forward and backward operations together. This is reflected in the set P of all steps because it does not contain special elements for the undo operations, i.e., there is no $ship^{-1}$ or $send^{-1}$. Instead, the Java objects that represent the steps in the implementation always have to provide methods for the forward and backward actions (cf. Sect. 4.1). However, from the administrative transaction's point of view these can only be changed together. The call to a compensating backward operation appears as read access to the specification for the forward operation, and consequently the multi-version concurrency control protocol ensures that the correct version of the implementation is accessed.

Table 1. Compatibility matrices before (a) and after (b) updating semantical information

a)

	a	b
a	+	+
b	+	–

b)

	a	b
a	+	–
b	–	–

5.2 Updating Semantical Information

In the previous section, we described that a multi-version concurrency control protocol can be used to change specifications of steps. Yet there is a subtlety that has to be observed when not only the implementation code but also additional semantical information are changed. Although the latter usually is associated with one specific step, its proposition may involve several steps. Compatibility information as described below is one representative of this phenomenon, but it occurs with other step-related information as well: changing its average duration may have ramifications for other steps if an overall deadline is to be met, new access rights can influence following steps, etc.

In ULTRAflow, every basic operation class contains not only the code for the forward and backward implementation, but also a compatibility matrix as semantical information to facilitate scheduling of concurrent workflows using a nested transaction approach [1,13]. The effects of changes to this matrix can even be shown using standard database conflict serialization theory [4,5,14]. Consider the following example:

Example 6. We assume two transactions T_1, T_2 executing basic operations a and b with a compatibility matrix as shown in Tab. 1a. Note that the matrix is arbitrarily chosen for the sake of the example. The history $a_1\ b_2\ b_1\ a_2\ commit_1\ commit_2$ obviously is serializable as the only operations in conflict are b_2 and b_1.

Now suppose the definition of b has been changed by another transaction. We write $\mu(b)$ to denote this event and get the new history $a_1\ b_2\ \mu(b)\ b_1\ a_2\ commit_1\ commit_2$. With our concurrency control protocol of Def. 6 transaction T_1 uses the new version of b, while T_2 uses the old one. Still, the overall execution is correct.

But what if the change of the code of b affects its compatibility behavior, i.e., if it implies a new matrix like Tab. 1b? The history $a_1\ b_2\ \mu(b)\ b_1\ a_2\ commit_1\ commit_2$ is no longer serializable, as after the update b_2 and b_1 imply the order $T_2 < T_1$, while b_1 and a_2 imply $T_1 < T_2$. However, all our correctness criteria state that the history is correct.

Apparently, the change made in the compatibility matrix of b implicitly changed also the matrix of a, and consequently the history should contain modifications of both, a and b, and look like $a_1\ b_2\ \mu(a,b)\ b_1\ a_2\ commit_1\ commit_2$. In this case it is obvious that T_1 has to execute the old version of b before the change. Then the original compatibility matrix (Tab. 1a) is used, and the history is serializable and thus correct.

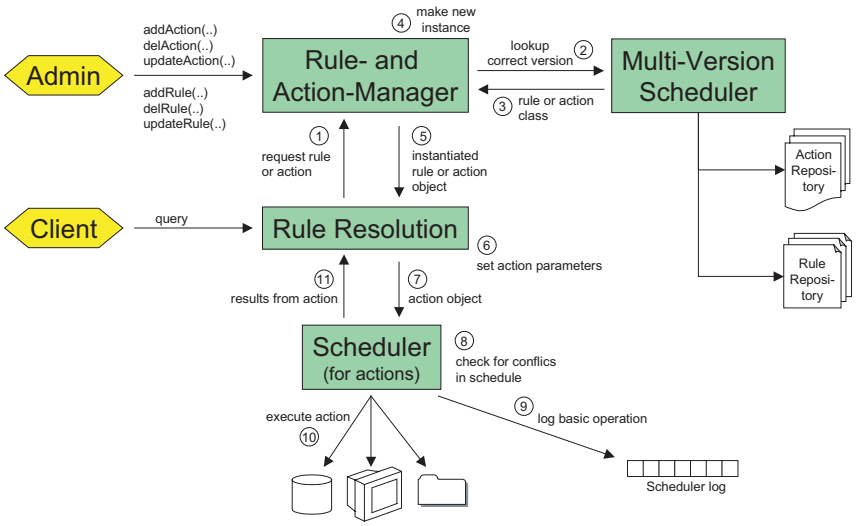

Fig. 3. Ensuring safe modification of operations by using an additional scheduler

The essence of this example is that changes of the semantical information may have effects on more than one operation and that they have to be accounted for adequately. In the case of ULTRAflow, it is enough to ensure that a change to semantical information is performed on *every* affected operation. In other systems that require other semantical information the effects of changing semantical information have to be analyzed, too.

6 Implementation within ULTRAflow: An Architecture

The architecture of Fig. 3 incorporates the multi-version scheduling for administrative transactions. The main components are as follows:

Rule- and Action-Manager: The central component of our architecture is the Rule- and Action-Manager (RAM). It manages the registered complex specifications and basic operations in a Rule- and an Action-Repository, resp., which map an action name to the appropriate implementation code (a class file). Each time a request for a rule or action is sent to the RAM, it gets the implementation of the rule or action from the repositories and then creates a new instance. Since the implementation can be changed at any time the RAM accesses the repositories not directly, but through the separate multi-version scheduler. The repositories correspond to the set P of Def. 1.

Multi-version Scheduler: This component implements the protocol as described in Sect. 5 and controls access to the Action and Rule Repositories. As required by the multi-version protocol, older implementations are kept in the repositories, too.

Rule Resolution: Evaluation of rules is done in the rule resolution component which proceeds as sketched in Sect. 3. The rule resolution requests rules and actions from the RAM and gets back instantiated Java objects. This happens repeatedly until a rule is completely resolved [13,18], while basic operations are sent directly to the scheduler.

Scheduler: The scheduler is mainly responsible for a correct execution of actions at the application level, i.e., of the productive workflows. It implements a concurrency control protocol on the basic operations [1] which is in general independent from the one the multi-version scheduler applies. When an action is sent to the scheduler it first checks whether the action conflicts with other actions in the schedule. If there is no conflict the action is logged in the scheduler log and executed afterwards. Execution can affect external systems, databases, etc. If the scheduler detects a conflict with other actions it can be necessary to rollback several actions executed earlier. In this case the log will be used to find the actions which have to be undone. To do so, the scheduler requests the corresponding action from the Rule- and Action-Manager and compensates it using its *undo* method.

Client: Clients are workflow instances that correspond to queries against the workflow rule base and are sent to the Rule Resolution (see below).

Admin: In the diagram, "Admin" stands for administrative tools that update workflow specifications and basic operations, i.e., steps. In the terminology of Sect. 5 these are the updater transactions, while the clients are the query transactions.

The chronological sequence of workflow execution is as follows: assume that a client workflow is started. This corresponds to a query "$\leftarrow q$" which is sent to the Rule Resolution. This component first requests an action or rule object from the RAM by giving the name (1), e.g., "*ship*". The RAM resolves the name and accesses the repository (2, 3) through the multi-version scheduler which returns the correct version of the requested rule or action. It then creates a new instance of the corresponding class (4) and returns this instance to the Rule Resolution (5). The Rule Resolution initializes the object by setting the calling parameters (6) that are bound during the resolution process. Complex sub-workflows are iteratively resolved, i.e., steps (1) to (6) are repeated.

Basic operations are passed to the scheduler (7). Before executing the action the scheduler checks for conflicts with other actions in the schedule (8) and makes an entry in its log (9). Provided the scheduler did not detect any conflicts the action is executed (10). This may cause for example access to a database, the file system or popup a message on the screen. Finally, results are returned to the Rule Resolution through the action object (11) which is given back after the execution.

7 Conclusion

Updating workflow specifications is an important topic in research as well as in practice. We showed how workflow specifications can evolve safely by defining

additional administrative transactions which are scheduled using a multi-version concurrency control protocol. This new protocol allows active workflows to use new specifications that were updated even after its start. Furthermore, we showed how this solution is implemented in our WfMS ULTRAflow. The proposed solution can also be applied to application servers and e-services.

Further work includes application of the schema to so-called "scientific workflows" that are only partly specified, as well as tuning of the rule resolution component and its interaction with the rule and action manager. In particular, a prefetching schema for steps is under investigation that can improve the reaction time of the system when a step in a workflow is finished and the next one is to be retrieved.

References

1. G. Alonso, D. Agrawal, A. E. Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *Proc. 12th Int. Conf. on Data Engineering*, pages 574–583, 1996. 517, 530, 532
2. Apache Software Foundation. *The Apache HTTP daemon*, 2001. http://www.apache.org. 518
3. A. Bernstein, C. Dellarocas, and M. Klein. Towards adaptive workflow systems (CSCW-98 workshop report). *ACM SIGMOD Record*, 28(3), 1999. 518
4. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987. 522, 524, 525, 526, 528, 530
5. P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997. 522, 524, 525, 526, 530
6. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. *Data and Knowledge Engineering*, 24(3):211–238, 1998. 519
7. D. Edmond and A. H. M. ter Hofstede. A reflective infrastructure for workflow adaptability. *Data and Knowledge Engineering*, 34(3):271–304, 2000. 519
8. C. Ellis and K. Keddara. ML-DEWS: Modeling language to support dynamic evolution within workflow systems. *Computer Supported Cooperative Work*, 9(3/4):293–333, 2000. 519
9. C. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *Proc. Conf. on Organizational Computing Systems*, pages 10–22, 1995. 517, 518
10. A. Fent and B. Freitag. ULTRAflow – a lightweight workflow management system. In *Proc. Int. Workshop on Functional and (Constraint) Logic Programming (WFLP2001), Kiel, Germany*, pages 375–378, 2001. 516, 519, 520
11. A. Fent and B. Freitag. ULTRAflow – Ein regelbasiertes Workflow Management System. In *Innovations-Workshop im Rahmen der Stuttgarter E-Business Innovationstage, 5.-8. November 2001, Stuttgart*. Fraunhofer IAO, 2001. 516, 519
12. A. Fent, H. Reiter, and B. Freitag. Design for change: Evolving workflow specifications in ULTRAflow. Technical Report MIP-0104, University of Passau (FMI), 2001. http://daisy.fmi.uni-passau.de/papers/. 516, 519, 528
13. A. Fent, C.-A. Wichert, and B. Freitag. Logical update queries as open nested transactions. In *Transactions and Database Dynamics*, volume 1773 of *LNCS*, pages 45–66. Springer, 2000. 517, 519, 520, 530, 532
14. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. 522, 524, 525, 530

15. C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, 2000. 519, 529
16. M. Klein, C. Dellarocas, and A. Bernstein. Special issue on adaptive workflow systems. *Computer Supported Cooperative Work*, 9(3/4), 2000. 518
17. H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In V. Kumar and M. Hsu, editors, *Recovery mechanisms in database systems*, chapter 15, pages 444–465. Prentice Hall, 1998. 518, 521, 522, 529
18. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1987. 520, 521, 532
19. N. C. Narendra. Adaptive workflow management - an integrated approach and system architecture. In *Proc. 2000 ACM Symposium on Applied Computing, Villa Olmo, Italy*, volume 2, pages 858–865, 2000. 519
20. Object Management Group. *The Common Object Request Broker Architecture, v2.0*, 1997. http://www.omg.org. 518
21. Orion Software. *The Orion J2EE Server*, 2001. http://www.orionserver.org. 518
22. T. Pilioura and A. Tsalgatidou. E-Services: Current technology and open issues. In *Proc. 2nd Int. Workshop on Technologies for E-Services (TES), Rome*, volume 2193 of *LNCS*, pages 1–15. Springer, 2001. 518, 520
23. M. Reichert and P. Dadam. $\text{ADEPT}_{\text{flex}}$ – supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, 10:93–129, 1998. 519, 528
24. G. Shegalov, M. Gillmann, and G. Weikum. XML-enabled workflow management for e-services across heterogeneous platforms. *VLDB-Journal*, 10(1):91–103, 2001. 520
25. Sun Microsystems. *Enterprise Java Beans*, 2001. http://www.javasoft.com/ejb. 518
26. W. van der Aalst. Exterminating the dynamic change bug. A concrete approach to support workflow change. *Information Systems Frontiers*, 3(3):297–317, 2001. 516, 517, 519, 527
27. C.-A. Wichert. *ULTRA – A logic transaction programming language*. PhD thesis, University of Passau, 2000. 517, 519, 520
28. C.-A. Wichert, A. Fent, and B. Freitag. A logical framework for the specification of transactions (extended version). Technical Report MIP-0102, University of Passau (FMI), 2001. http://daisy.fmi.uni-passau.de/papers/. 517, 519, 520