The Workflow Activity Model WAMO

Research paper

Johann Eder, Walter Liebhart
Institut für Informatik, Universität Klagenfurt
A-9020 Klagenfurt, Austria
email:{eder,walter}@ifi.uni-klu.ac.at

Abstract

Workflow technology has not yet lived up to its expectations not only because of social problems but also because of technical problems, like inflexible and rigid process specification and execution mechanisms and insufficient possibilities to handle exceptions. The aim of this paper is to present a workflow model which significantly facilitates the design and reliable management of complex business processes supported by an automatic mechanism to handle exceptions. The strength of the model is its simplicity and the application independent transaction facility (advanced control mechanism for workflow units) which guarantees reliable execution of workflow activities.

1 Introduction

Among cooperative information systems, workflow management is one of the key technologies for providing efficiency and effectiveness in the office. A Workflow Management System (WFMS) is a system that completely defines, manages and executes workflow processes through the execution of software whose order of execution is driven by a computer representation of the workflow process logic [1]. Of course, the work steps or tasks in a workflow process are not restricted only to the execution of some software programs or modules but also to any thinkable work unit which has to be done (e.g., to make a phone call).

Although there are more and more success stories in the area of workflow, it is generally acknowledged that workflow has not lived up to its expectation. Besides social problems (e.g., cultural resistance to change) there are still a lot of technical problems, like insufficient tools and methodologies to describe processes and exceptions. Additionally, most of the current available workflow systems do not very well support automatic operation execution, status monitoring, enforcement of consistency and concurrency control, or recovery from failure.

In this paper we want to address at least some of the above mentioned shortcomings by introducing the transaction oriented Workflow Activity Model WAMO [2] which enables the workflow designer in modeling not only correct business processes but also potential (expectable) exceptions which may arise during process execution. Exceptions will be handled automatically by the underlying workflow system which we have partially developed [3]. One of the main goals of our work is to provide mechanisms for defining and controlling long-lived activities, just like transactions in traditional DBMSs control short computations. Therefore our work is based on the ideas of *Transactional Workflows* which are a combination of workflow systems and database management systems (DBMSs) with the intention to incorporate the advantages of both technologies [4, 5].

The remainder of this paper is organized as follows: in the rest of this section we focus on exceptions and failures and we present related research activities in the area of workflow specification and execution. In section 2 we describe the workflow activity model WAMO with the help of a workflow metamodel. In section 3 we define the semantics of model and the workflow activity description language WADL which is used to describe workflow processes according to WAMO. Additionally, we analyze safety questions of modelled processes and we present a small example. The example shows a workflow for the arrangement of a trip reservation, including dependencies between work units (e.g., payment may not start before the flight reservation is done, the car - room reservation may only be started when the flight reservation terminates with a positive result) and compensation units for exception handling (e.g., the client cancels the flight and therefore the flight reservation must be undone). Section 4 concludes this paper.

1.1 Exceptions and Failures

Exceptions and failures are basic characteristics of cooperative information systems.

type	level	examples
unexpected exceptions	process definition level	the structure of the modelled process cannot handle a special case (e.g., change of order processing for only one but very important client)
expected exceptions	WAMO level	one of the process work steps fails (e.g., the client cannot pay the bill, a flight cannot be booked because it is already booked-up)
application failures	application level	program failures, constraint violations
basic failures	system level	system crash, deadlocks, connection problems, printer breakdown

Table 1: Classification of failures and exceptions

Therefore the adequate and efficient treatment of exceptions and failures in cooperative information systems, especially in workflow systems is a critical success factor. We have identified four different types of failures and exceptions associated with the corresponding levels where these failures and exceptions can be handled. The classification is summarized in table 1 and more precisely explained in the remaining part of this subsection.

Basic failures: Failures of this kind are handled at system level by the corresponding components of the underlying system (e.g., the DBMS, the operating system, the network software). Typical failures in the database area are deadlocks, connection problems or media failures. There are well known techniques (e.g., rollback, rollforward, redo, reread of a record) to manage such failures in a consistent way. If a failure like a system crash happens, the WFMS is responsible for storing all necessary *context information* (process information) in order to automatically restart the interrupted processes as soon as the underlying DBMS is in a consistent state and available again.

Application failures: Such failures mainly comprise programming failures (e.g., because of unexpected intput) in components (tasks) of the application program. The WFMS will interrupt the current process execution with the faulty task and wait until the human expert has eliminated the failures. Ideally, the workflow administrator should have some possibilities to handle such failures temporarily in order to enable process progress (to perform the task manually, to skip the faulty task if this does not lead to inconsistent states, etc.) until the failure is corrected by a corresponding person. In any case, the process must be restarted by the workflow administrator.

Expected exceptions: We characterize expected exceptions as something which does not represent the "normal" case but still may arise frequently and therefore special mechanisms are available to handle such *special cases*. Workflows typically consist of tasks which depend on the progress (success or failure) or on the result of other tasks. In WAMO, the workflow designer has the possibility to specify - during process definition - how the process should behave, if a particular task terminates *negatively* (e.g, a flight cannot be booked because it is already booked-up). In general, the aim is to make forward progress but if a positive execution of the task is expected definitely but not possible, it may be necessary to undo some previous computations (especially to compensate previous tasks) in order to reach a consistent state again and then to try to continue process execution by executing an alternative path in order to reach the end of the overall process.

We want to emphasize that such situations occur *frequently* and therefore it is necessary to support the *modeling* and *automation* of such exceptions! As already mentioned, WAMO offers an adequate and simple mechanism to handle such exceptions by the usage of special transaction specific constructs during process definition (see subsection 2.2). The main advantage of this concept is that the workflow designer needs not to specify all possible process execution alternatives - which would lead to a complex and incomprehensible process description - because they are computed and executed reliably by the system during runtime execution.

Unexpected exceptions: An important class of exceptions concern the necessity to change the current process structure of a defined workflow during runtime because of totally new, unexpected requirements (e.g., change of order processing for only one but very important client). Of course, it is

neither recommended nor feasible to model *all* possible exceptions but the WFMS should be open to handle such situations adequate at runtime (e.g., to skip an activity; to change the order of two activities). Exceptions of this level are more detailed described in [6, 7].

1.2 Related Work

Important areas of related work are the research activities concerning advanced transaction models. A first step in the evolution of the traditional (flat) transaction model was the development of (closed) *Nested Transactions* [8]. The main advantages of the nested transaction model are the support of *modularity* (decomposition of transactions), *failure handling* at the granularity of subtransactions and *intra-transaction parallelism*. An interesting extension in the area of nested transactions are *Open Nested Transactions* [9]. They relax the isolation requirements of nested transactions and make the results of committed subtransactions visible to other concurrently executing nested transaction. A new dimension in the evolution of transaction models was opened by the concept of compensation, used for example in the *Saga Model* [10]. Sagas are long-lived transactions that can be interleaved in any order with subtransactions of other sagas. A saga requires that either all subtransactions complete execution or compensating transactions are run to undo the effects of partial execution.

A lot of related work is also done in the area of distributed transaction management, as for example the development of the *DOM Transaction Model* [11] which allows closed nested and/or open nested transactions. The *Flexible Transactions Model* [12] is based on the nested transaction model and has been proposed as a transaction model suitable for a multi-database environment. The objective of multidatabase systems [13] is to integrate autonomous software systems (legacy systems) and is therefore very close to the objective of workflow systems which try to integrate local services of an institution (e.g., mailers, text systems) and also services of autonomous institutions. Based on the work on multidatabase systems there are also interesting research activities in defining general purpose work flow languages [14].

Important issues related to transactional workflow models besides [10, 11, 12], for example, were addressed in the work of *Long-Running Activities* [15], *On Transactional Workflows* [5] or *The ConTract Model* [16]. In [15] a *Long-Running Activity* consists recursively of multiple application steps each of which is either an activity or a (nested) transaction. Control flow and data flow of an activity may be specified statically in the activity's script or dynamically by ECA-rules. The model includes compensation, communication between steps and exception handling. In [5] a precise overview of *transactional workflows*, including a description of an abstract model of a task (by a state machine) is introduced. The *ConTract Model* [16] tries to provide the formal basis for defining and controlling long-lived, complex computations. *ConTracts* can be seen as a mechanism for grouping transactions into a multi-transaction activity. A *ConTract* consists of a set of predefined actions (with ACID properties) called steps, and the execution plan called a script.

However, the main difference to this related work is that our model is much easier to work with as it does not require skilled programmers and it is more flexible in modeling complex business processes because of the possibility to use selectively *easy* but *expressive* control structures and intuitive simple transaction specific features. Additionally, our model supports the *automatic* handling of exceptions and it does not presume the existence of a database system with an *advanced* transaction mechanism.

2 The Workflow Activity Model WAMO

WAMO enables the workflow designer to *easily* model complex business processes in a simple and straightforward manner. The basic idea is to decompose a complex business process into smaller work units (activities) which themselves consist of ideally *preexisting* tasks and to guarantee *reliable* flow control with automatic exception handling by using control structures and special transaction parameters which are input to the workflow scheduler.

2.1 The Conceptual WAMO Architecture

Up to now there is no general accepted *workflow metamodel* but there are a lot of efforts in defining such a model [17]. We have developed a metamodel which incorporates traditional workflow modeling features as well as transaction specific features. The metamodel presented in Figure 1 is adopted to the purpose of this paper and does therefore not contain *all* necessary components of a workflow metamodel.

A workflow typically consists of multiple activities, forms and agents. *Activities* represent any abstract description of work units in the business process. A *form* is a data container or folder which stores process and application relevant data. Forms are passed between activities because they are necessary for the communication. The *agent* or processing entity is responsible for the execution of activities. In WAM0, the agent can more precisely be modelled with the help of users and roles.

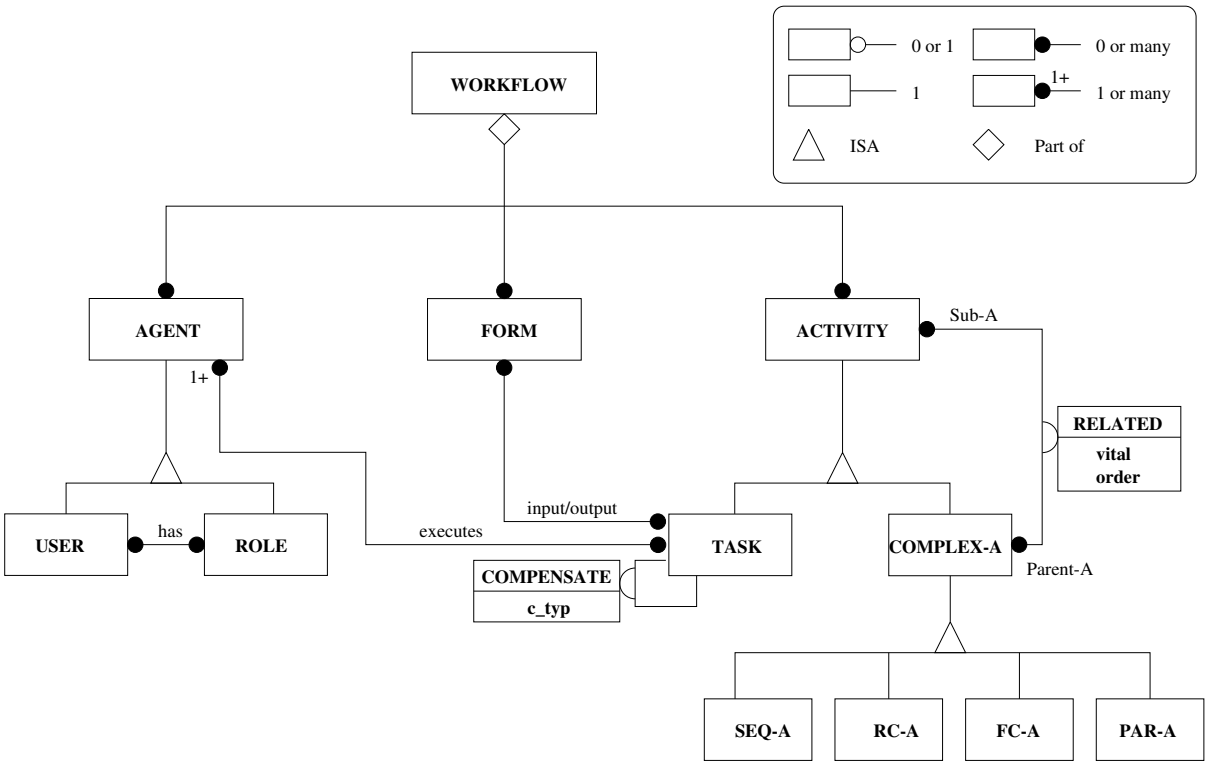

Figure 1: WAMO Metamodel

A characteristic feature of our approach is that activities may consist of other activities, representing *subprocesses*. Furthermore, it is possible that a certain activity takes part in several other activities, especially several times in an other activity. On the design level the *related-association* between activities is many-to-many. This allows that new workflows can easily be composed using predefined activities. Subprocesses which occur several times in different processes have to be designed only once. And for maintenance, it is only necessary to change such a subprocess once, and it is changed for all workflows where it appears.

For the definition of the dependencies between activities, an occurrence of an activity in a workflow has to be aware of its process context. Therefore, we transform the design information contained in the metamodel into *activity trees* - one tree for each workflow. In such activity trees different occurrences of the same activity are unambiguously distinguished, such that we can define the dependencies between activities on basis of these activity occurrences. In the following we do not distinguish between activities and activity occurrences, unless it is necessary for clarity.

Additionally, there exist *complex activities* in the model to specify control structures (behavior) over activities. These control structures are illustrated by the edges of nested activities in the *activity tree*. The leave activities of the activity tree are *tasks* which are the elementary units of work and not further decomposable (e.g., database transactions, application programs, software modules, autonomous services in a network or human interactions). We want to emphasize that we treat tasks like *black boxes* which are not developed by the workflow designer and which must have a corresponding interface in order to communicate with them (e.g., start a task, data exchange). Additionally, every task in the activity tree has a corresponding *parent activity*.

*Control structure*s are structuring mechanisms to support the decomposition of business processes into smaller work units and to define the flow of control between these work units in the activity tree. There are five simple but powerful control structures:

- *Sequence:* In a sequence activities are executed strictly sequential. This means that activity A_i cannot begin execution until activity A_{i-1} has terminated.
- *Ranked Choice:* This construct enables the modeling of alternative (contingency) activities. An alternative activity is executed only if the immediate previous activity fails (commits unsuccessfully).
- *Free Choice:* The free choice construct is similar to the ranked choice but the activation order of the alternative activities in a free choice list is

computed dynamically (at run time).

- *Parallel:* A parallel control structure enables activities and tasks to run concurrently.
- *Nesting:* The decomposition of activities is supported by the concept of nesting. An activity can be further decomposed into smaller, less complex subactivities.

2.2 Execution of Activities and Tasks

At run time activities are associated with unique identifiers and the activity tree defines the execution order of activities and tasks in the *activity execution tree* (AET). Activities and tasks have different *execution states* during execution, like startable, active and so forth. Additionally, activities and tasks are able to react on events, like start, abort or commit. All activities and tasks in the AET are executed under the control of an advanced transaction manager. Main characteristics of the underlying transaction model are:

- *Relaxed Atomicity:* Each application may have its own application dependent failure atomicity. A workflow may survive and make forward progress although some of its tasks do not terminate successfully.
- *Relaxed Serializability:* It is not possible to execute an entire workflow as a single isolated transaction to achieve (data) consistency. In our approach consistency is guaranteed by user defined *semantic serializability* [18] between concurrent and interleaving workflows (inter-workflow dependencies) and correct execution of each individual workflow (intra-workflow dependencies). Nevertheless, the traditional conflict serializability criterion will be necessary for traditional DB-transactions within tasks.
- *Relaxed Isolation:* Isolation will be relaxed which means that activities may externalize uncommitted results and release resources to achieve a higher degree of concurrency. This "dangerous" feature is complemented by the concept of compensation which enables a semantic undo of already committed activities.

Many of the advanced transaction features are very easy to use and control by the workflow designer during process specification (construction of the AET), as for example:

- By *control structures*: Control structures are simple but expressive mechanisms to handle activity coordination requirements (intra-workflow dependencies).
- By *transaction specific features*: Tasks can be specified more detailed by the STORNO-TYPE and FORCE parameter. Additionally, activities which are not essential for a successful termination of the corresponding parent activity can be defined as NON VITAL (NV).

The STORNO-TYPE and FORCE parameters of a task are necessary for eventual compensation transactions. With the STORNO-TYPE the workflow designer may specify how a specific task behaves in case of compensation. There are four possibilities:

- *none:* The committed task does not need to be compensated because it is not relevant from an application point of view.
- *undoable:* The committed task can be undone by the corresponding compensation task without any side-effects. Let S be the database state at some time t, T the original task and TC the compensation task. Then the database state S' after executing T and TC in sequence equals the previous state S if in the between time no other operation has been executed. (e.g. a client makes a flight reservation - later he cancels the reservation without paying a cancellation fee; an expert attaches a notice to a document which may be undone later by simply removing it).
- *compensatable:* The committed task can be *semantically* undone by the corresponding compensation task but there are side-effects. Let again S, S' be database states, T the original task and TC the compensation task. Then the database state S' after executing T and TC in sequence may not be equal to the previous state S, regardless whether in the between time other operations have been executed or not (e.g., a client makes a flight reservation - later he cancels the reservation but now he has to pay a cancellation fee, money transfer and back transfer with transfer fees).
- *critical:* The task cannot be undone or compensated afterwards because there exists no compensation task to undo the already committed effects (e.g., drilling a hole, mailing a sensitive information, etc.).

Some tasks in real world situations are always expected to terminate successfully (e.g., open an account, print a document). This natural feature may also be demanded from tasks in our model by using the parameter FORCE during specification of a task. If a forcable task terminates negatively (e.g. because of a connection problem or something similar) then it is repeated and re-executed several times (specified by the workflow designer) until a positive acknowledgement is achieved. Otherwise, process execution stops and the workflow administrator has to intervene manually (e.g. fix the defect and restart the task; change the state of the task).

Besides forcable activities and tasks there may also exist activities within a workflow AET which are *not essential* for the parent activity to terminate successfully. For such parent-child relations we have introduced the transaction specific parameter NON VITAL. If a non vital activity fails, the workflow can continue and make forward progress without any compensation actions (e.g., the travel agency has booked a flight but it is not possible to rent a car (which has been specified as a non vital activity) - now it is not necessary to interrupt or compensate the whole trip reservation). Normally, all activities within a workflow AET are essential and therefore vital for the parent activity. In any case, if a *vital* activity fails then the compensation mechanism is activated. For example: if a vital activity in a sequence fails then the whole sequence fails which means that all previous successful committed activities and tasks in the sequence have to be compensated.

3 Definition of the Workflow Activity Description Language WADL

For the formalization of a complex business process we have developed the simple to use and high-level *Workflow Activity Description Language WADL*. As already explained, the basic elements of our model are activities, tasks, control structures and transaction specific parameters. We now introduce WADL with the following syntactic sketch (a complete definition in Backus-Naur Form can be found in [19]):

DEFINITION ACTIVITY A_ID

SEQUENCE [NV] A {[NV] A} END-SEQUENCE OR

RANKED CHOICE A {A} END-RANKED-CHOICE OR

FREE CHOICE A {A} END-FREE-CHOICE OR

PARALLEL [NV] A {[NV] A} END-PARALLEL OR

TASK

END-ACTIVITY-DEFINITION

DEFINITION TASK T_ID [STORNO-TYPE] [FORCE]

ACID-Transaction | ApplicationProgram | HumanInteraction

INVERSE_TASK T_ID

% INVERSE_TASK is necessary if STORNO-TYPE is (COMPENSATABLE or UNDOABLE)

END-TASK-DEFINITION

With the usage of WADL, the workflow designer is able to *decompose* complex business processes into smaller and hence less complex work units (subprocesses). The semantics of the language, especially the workflow coordination requirements (control- and data flow) between work units can be described *formally* by *dependency rules* based on valid state transitions between nodes (activities and tasks) in the AET. The possible state transitions mainly depend on execution states (e.g., a flight reservation was possible and therefore the task commits successfully) and output values of other activities or tasks.

The main idea to implement this language is to define the necessary dependencies with the help of the ACTA Transaction Metamodel [20] and then to map these specifications into *production rules* [21]. These rules will be integrated into our existing prototype *PantaRhei*[3] which is based on an active database system. Currently we are moving to an active object-relational DBMS.

3.1 A Short Overview of ACTA

ACTA is a transaction framework which facilitates the formal description of properties of extended transaction models. It allows the specification of transaction types, whereby a transaction type intentionally describes a set of transaction instances that share *structure* and *behavior*. With ACTA, the effects of transactions on other transactions (intertransaction dependencies) and also their effects on objects (visibility of and conflicts between operations on objects) through constraints on histories can be formally specified. Transaction instances issue events, mainly *transaction events* (i.e., begin, commit) and *object events* (i.e., data manipulation events). An event causes a unit of work to switch to a particular state. The following concepts are important:

- A *history H* of the concurrent execution of a set of transactions T contains all the events associated with the transactions in T and indicates the (partial) order in which these events occur.
- The predicate $e \rightarrow e'$ is true if event e precedes event e' in history H. It is false, otherwise.
- $(e \in H) \Rightarrow Condition_H$, where $\Rightarrow$ denotes implication, specifies that the event e can belong to history H only if $Condition_H$ is satisfied. In other words, $Condition_H$ is necessary for e to be in H. $Condition_H$ is a predicate involving the events in H.
- $Condition_H \Rightarrow (e \in H)$ specifies that if $Condition_H$ holds, e should be in the history H. In other words, $Condition_H$ is sufficient for e to be in H. Typically, (parts of) the semantics of one transaction type depend on its relationships to other types which may be expressed by *dependencies*. A dependency is an implication that constrains the occurrence or order of events of two transactions.

Dependency	Definition	Meaning
Abort Dep. t_j AD t_i	$(Abort_{ti} \in H) \Rightarrow (Abort_{tj} \in H)$	if t_i aborts then t_j aborts
Commit -on-Termination Dep. t_j CTD t_i	$(e \in H) \Rightarrow (e \rightarrow Commit_{tj})$ where $e \in \{Commit_{ti}, Abort_{ti}\}$	if t_i terminates then t_j commits
Serial Dep. t_j SD t_i	$(Begin_{tj} \in H) \Rightarrow (e \rightarrow Begin_{tj})$ where $e \in \{Commit_{ti}, Abort_{ti}\}$	t_j cannot begin execution until t_i either commits or aborts
Begin Dep. t_j BD t_i	$(Begin_{tj} \in H) \Rightarrow (Begin_{ti} \rightarrow Begin_{tj})$	t_j cannot begin execution until t_i has begun
Begin-on-Commit Dep. t_j BCD t_i	$(Begin_{tj} \in H) \Rightarrow (Commit_{ti} \rightarrow Begin_{tj})$	t_j cannot begin execution until t_i commits
Begin-on-Abort Dep. t_j BAD t_i	$(Begin_{tj} \in H) \Rightarrow (Abort_{ti} \rightarrow Begin_{tj})$	t_j cannot begin execution until t_i aborts
Force-Commit-on-Abort Dep. t_j CMD t_i	$(Abort_{ti} \in H) \Rightarrow (Commit_{tj} \in H)$	if t_i aborts, t_j commits

Table 2: Examples for ACTA-Dependencies

Dependencies can arise due to *structure* (e.g., dependencies between a parent and child transactions) or due to *behavior* (operations on objects). Some examples of structural dependencies which are used in WADL are summarized in table 2.

3.2 Semantics of WADL

In this subsection we will briefly describe parts of the semantics of WADL with the help of structural ACTA dependencies. First we define the essential *events* and *states* of our model, presented in Figure 2.

An activity can be started if it is in the initial state *startable*. By triggering the event *start*, the activity changes its state from startable to *active*. After the corresponding child-activities have finished, the activity either *succeeds* or *fails*, depending on the execution result(s) of its child(s). These two events have the following semantics:

- Succeed is the same as commit successfully or commit with a positive result.
- Fail corresponds to commit unsuccessfully, commit with a negative result or abort semantically.

The corresponding termination states are called *committed successfully* or *committed unsuccessfully*.

An important feature of WAMO is its compensation concept. Normally, a compensation is initiated if a vital activity fails. Then the general solution is to compensate all previous *successful committed* activities in the current control structure in inverse order. There are two important strategies in compensating activities:

- A compensation of an successful committed activity is propagated to its child activities until the task level is reached. At task level, the corresponding task is compensated according to its STORNO-TYPE. As soon as all childs of the activity are compensated successfully, the compensating activity changes its state to *compensated*. If the compensation fails, an error is raised and a manual intervention is necessary.
- As soon as all relevant activities in the current control structure are compensated, the parent activity of the control structure changes its state to *committed unsuccessfully*. Depending on the current situatiuon, the system now can make forward progress (e.g., to try the next alternative in a choice) or again start a new compensation process on the next higher level.

The above mentioned compensation process shows, that only successful committed activities can be compensated. Now, in WAMO it is possible to *abort* an active activity by the user or by the system (e.g., if the client for whom the trip reservation is arranged, falls ill then the whole reservation should be interrupted). It must be mentioned, that an abort is not always possible because it may endanger the safety of the overall process (see subsection 3. 3). A nice feature of WAMO is the reusability of the compensation concept for an abortion and vice versa. The abortion of an *active* activity roughly consists of the following steps:

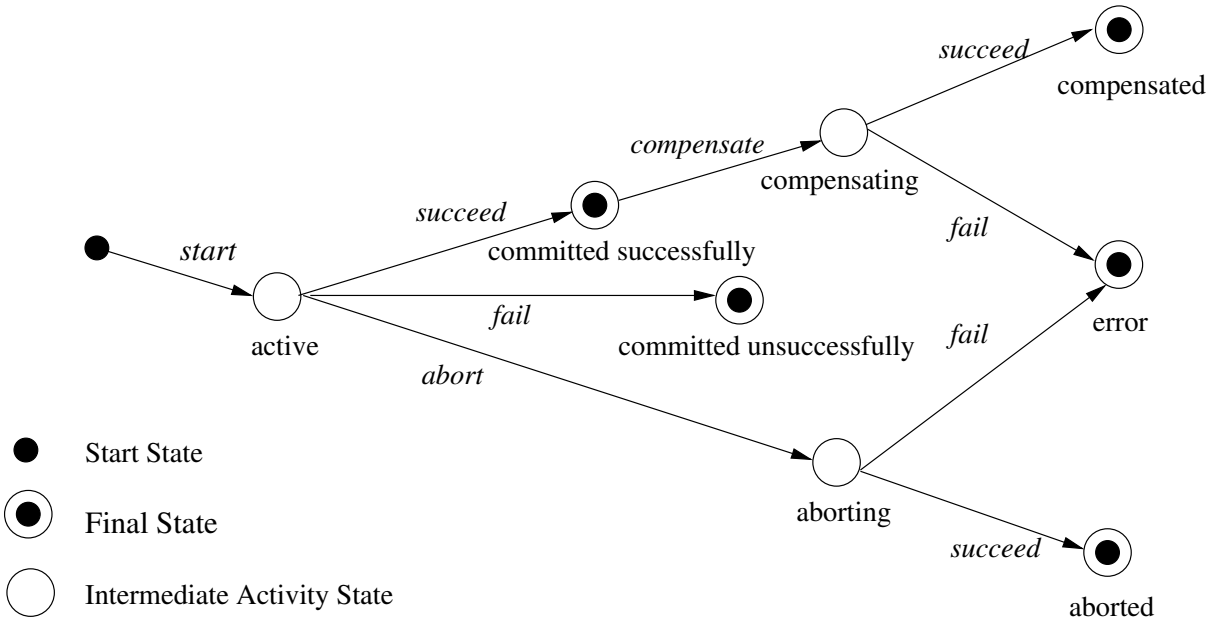

Figure 2: Event-state diagram for activities

- The active activity changes its state to *aborting* and propagates the abortion to its child activities until the tasks are reached. The currently *active* task is aborted and the state of the parent activity of the task is changed to aborted which corresponds to committed unsuccessfully.
- The remaining steps which are necessary to complete the abortion are executed like an ordinary compensation: Compensate previously successful committed activities in the same control structure. Then change the state of the parent activity to *aborted* which corresponds a commit unsuccessfully and continue with the compensation until the *initial aborted* activity is reached. At this point, all active childs of the activity are aborted and all successful committed childs are compensated. Finally the state of the aborting activity is changed to *aborted* which again corresponds an unsuccessful commit. Now the abortion process is finished but now there may be further compensations necessary if the aborted activity was a vital activity.

The abortion of a task demands some additional actions: First it is necessary to distinguish in which state the task currently is and whether the task has a corresponding mechanism to handle an abort-event or not. We cannot a priori demand such a functionality from a task because we treat tasks as black boxes. Therefore there are two possibilities to handle an abort:

a) The task is *active* and it is an *abortable* task. The task reacts immediately to the abort event and executes a rollback which leads to the state *aborted* which corresponds to committed unsuccessfully.
b) The task is *active* but it is a *not abortable* task which means that there is no direct mechanism to execute an internal, task specific rollback. Therefore, it is necessary to wait until the task has finished execution. Depending on the termination state, the task now already is in the correct state if it failed. Otherwise, the task is compensated according its STORNO-TYPE immediately. Finally the state of the task is changed to aborted.

Besides the previous mentioned termination events for activities and tasks there are two additional abort events for tasks. A task may abort because of a *failure at system level* or because of a *failure at program level*. If there is a failure at system level the task will be restarted automatically several times - this can be configured at process specification time - until the task succeeds. If this is not possible, the task state is changed to *committed unsuccessfully* in case it is not a forcable task. If it is a *forcable* task, then a manual intervention by the workflow administrator is necessary because of safety reasons. Such a situation is very serious and neither expected by the workflow designer nor by the system (e.g., an urgent request for payment must be sent out but the file with the necessary data is corrupted).

In the same way, a manual intervention is necessary, if a task aborts because of a failure at program level. Manual intervention depends on the current situation and means to correct the failure, to restart the task or to change the execution state of the task in order to support process progress.

In the rest of this section we analyze structural dependencies of activities in a sequence. The following abbreviations are used:

PA:	Parent Activity
CA:	Child Activity
VCA:	Vital Child Activity; the relation between the CA and its PA is vital
CompA:	Compensation Activity; because of implementation aspects, each activity has a corresponding compensation activity which controls the compensation
n:	number of last activity in the sequence

- *Structural dependencies between activities in a sequence:*

(a) CA_1 *BD PA* % Begin Dependency
(b) *PA CTD* CA_n % Commit-on-Termination Dep.
(c) *PA AD* VCA_j % Abort Dependency
(d) CA_i *SD* CA_{i-1} *:* $1 < i \leq n$ % Serial Dependency
(e) CA_i *BCD* VCA_{i-1}*:* $1 < i \leq n$
%Begin-on-Commit Dep.

The semantics of these logical clauses is:
(a) The first child activity CA_1 of the sequence cannot be started before its parent activity PA has been started.
(b) PA succeeds as soon as the last CA in the sequence terminates (either succeeds or fails)
(c) If there is a vital child activity *VCA* and this activity fails then the whole sequence and PA fails.
(d) If the sequence consists of several child activities, then the activity CA_i in the sequence is started as soon as the previous activity CA_{i-1} has terminated.
(e) If the previous child activity is a vital activity then the next child activity CA_i in the sequence is started only if VCA_{i-1} succeeds.

- *Structural compensation dependencies between activities in a sequence:*

If a vital child activity (VCA) fails then the compensation process is activated! All previous executed activities which have committed successfully are compensated in inverse order. At last the parent activity fails.

(a) $CompA_i$ *BAD* VCA_j %Begin-on-Abort Dep.
(b) $CompA_j$ *CMD* VCA_i % Force-Commit-on-
% Abort Dep.
(c) $CompA_i$ *BCD* A_i % Begin-on-Commit Dep.
(d) $CompA_{j-1}$ *BCD* $CompA_j$
% Begin-on-Commit Dep.

The semantics of these logical clauses is:
(a) The compensation activity $CompA_i$ of the immediately previous *succeeded* (successful committed) activity of the vital activity VCA_j cannot be started before VCA_j has *failed*.
(b) If a vital child activity VCA_i fails then the compensation activity $CompA_j$ of the immediately previous successful committed activity *must* succeed.
(c) An activity can only be compensated if it has *succeeded* before.
(d) If there are several successful committed activities in the sequence then the compensation activity $CompA_{j-1}$ of the successful committed activity A_{j-1} cannot be started before $CompA_j$ (of the successful committed activity A_j) has committed successfully.

A complete definition of structural dependencies of the workflow activity language WADL can be found in [19].

3.3 Safety of defined processes

After the workflow designer has defined a complex business process according to the concepts of WAMO it would be very helpful to compile the specified process in order to check whether the specified process is *unsafe* or not. We define a process as unsafe if during process execution a compensation of a critical task can become necessary to proceed with process execution. However, this is a pessimistic approach because we declare processes as unsafe as soon as we detect a *possible* activation order of activities and tasks which may lead to an unsafe process but the point is that we do not forbid such process specifications but we only want to inform the workflow designer of this fact. The detection of unsafe processes is influenced by the following parameters:

- The existence of different control structures in the process definition.
- The usage of the special transaction parameters *vital* and *non vital* for activities. The concept of compensation is the reason why at all safety computations are required and a compensation is initiated only if a vital activity terminates negatively.

- The usage of the task specific transaction parameter *storno-type*. Tasks with the storno-type *critical* play a central role for the computation of safety because it must be checked whether they can come into an compensation process or not.
- The usage of the task specific transaction parameter *force*. Forceable tasks help to improve the safety of a process because they always terminate positively.

As stated above, the task specific parameters *critical* and *force* are crucial for safety computations. They are inherited upwards to its parent activities which leads to critical activities and forcable activities.

An activity is a *critical activity*, iff one of the following conditions holds:
(a) it is a critical task
(b) it is the parent of a sequence of activities and at least one activity in the sequence is a critical activity
(c) it is the parent of a ranked / free choice of activities and at least one activity in the choice is a critical activity
(d) it is the parent of a set of parallel activities and at least one activity of the parallel activities is a critical activity

An activity is a *forcable activity*, iff one of the following conditions holds:
(a) it is a forcable task
(b) it is the parent of a sequence of activities and all *vital* activities in the sequence are forcable activities
(c) it is the parent of a ranked / free choice of activities and at least one activity in the choice is a forcable activity
(d) it is the parent of a set of parallel activities and all *vital* activities in the parallel set are forcable activities

We want to emphasize that the semantics of a forcable activity is slightly different to the semantics of a forcable task. To achieve the same semantics, it is necessary that all forcable tasks are safe or critical safe (explained in the rest of this subsection).

Now we are able to compute the safety of activities which is done bottom up, from the leave nodes of the AET to the root node of the AET. We distinguish between three different safety states: safe, critical safe or unsafe.

- An activity is *safe* if there is no critical task specified in the process or the critical task is the last task in the process.
- An activity is *critical safe* if there are critical tasks in the process definition but it can be guaranteed that during process execution no compensation of a critical task is attempted.

  There are some important rules for critical safe activities:

  The parent activity of a *sequence* of activities is critical safe, iff the following conditions holds:
  (a) all critical activities in the sequence are critical safe and
  (b) all vital activities after a critical activity in the sequence are forcable activities

  The parent activity of a *ranked / free choice* is critical safe, iff the following condition hold:
  (a) all critical activities are critical safe

  The parent activity of a set of *parallel* activities is critical safe, iff the following conditions hold:
  (a) all critical activities are critical safe and
  (b) all vital activities are forcable
- An activity is *unsafe* if it is neither safe nor critical safe.

With the possibility of safety computations the workflow designer can be warned at compile time that there are unsafe states in the process specification. Additionally, there are possibilities to transform automatically unsafe process definitions into safe process definitions but this is still topic of further research.

3.4 A Small Example

In this subsection we present a simplified and incomplete business process example which emphasizes the most important features of our model. The example is illustrated graphically (see Figure 3) and also more formal by the usage of WADL:

```
ACTIVITY Trip_Reservation
    SEQUENCE
        Flight_Reservation
        Car_Room_Reservation (NV)
        Payment
        Document Handling
  END SEQUENCE
    ACTIVITY Flight_Reservation
        SEQUENCE
            prepare
            exec_FR
        END SEQUENCE
    END ACTIVITY Flight_Reservation
```

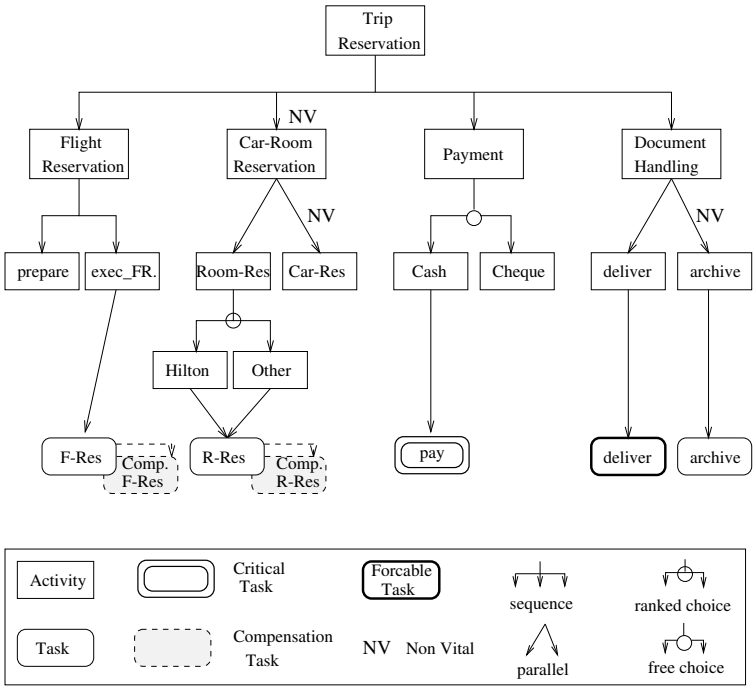

Figure 3: Trip Reservation

```
        ACTIVITY exec_FR
            TASK F_Res
        END ACTIVITY F_Res
    ACTIVITY Car_Room_Reservation
        PARALLEL
            Room_Res
            Car_Res (NV)
        END PARALLEL
    END ACTIVITY Car_Room_Reservation
    ACTIVITY Payment
        FREE CHOICE
            Cash
            Cheque
        END FREE CHOICE
    END ACTIVITY Payment
    ACTIVITY Document Handling
        PARALLEL
            deliver
            archive
        END PARALLEL
    END ACTIVITY Document Handling

    TASK F_Res (COMPENSATABLE)
        F_Res
        INVERSE_TASK Comp_F_Res
    END TASK F_Res
    ...
END ACTIVITY Trip_Reservation
```

A *Trip Reservation* consists of the activity sequence *Flight Reservation - Car-Room Reservation - Payment* and *Document Handling*. *Car-Room Res.* is not vital (NV) for the parent-activity *Trip Res*., whereas the other activities in the sequence are vital (per default). This means, if one of these activities terminates negatively, a compensation process is started automatically and the whole trip reservation will fail. If for example the vital activity *Payment* terminates negatively then the activities *Car-Room Res.* and *Flight Res.* have to be undone (compensated).

Flight Res. is further decomposed into two activities whereby the activity *exec_FR* is realized by the task *F-Res* and the corresponding compensation task *Comp. F-Res*.

Car-Room Res. consists of two parallel executable activities: *Room-Res* and *Car-Res*. *Room-Res* is modelled as a ranked choice consisting of two alternatives whose activation order is from left to right (if there is no room at Hilton then the system attempts a different hotel). If the non vital *Car-Res* fails, the parent activity *Car-Room Reservation* still may terminate successfully.

Payment is modelled as a free choice which means that the activation order of the alternatives depends on information which is computed at execution time. If the choice is *cash* then the critical task *pay* is executed and this means that a compensation of this task (e.g., to return the money back to the client) later on is not possible.

Document Handling consists of two activities which may be executed in parallel. Additionally the activity *deliver* must succeed because it is forcable.

4 Conclusions

We have defined the Workflow Activity Model WAMO which supports the workflow designer in modeling complex business processes. The model is based on the concepts of transactional workflows in order to guarantee reliable execution of workflow activities by an advanced transaction management facility. In particular, we have developed a workflow metamodel which incorporates traditional workflow modeling features as well as transaction specific features and we have presented the high-level *Workflow Activity Description Language WADL* to model such transactional workflows. The strength of the language mainly is based on simple structuring mechanisms and the possibility to express application specific transactional requirements. Typically, workflow activities are of long duration, highly concurrent and of a cooperative nature. Therefore, our transaction facility enables the relaxation of atomicity, serializability and isolation as well as, for example, the possibility to compensate activities.

One of the main advantages of WAMO is the exception handling mechanism. The workflow designer needs not to model all possible process execution alternatives (exceptions), instead he only needs to specify whether special tasks are essential for the workflow and whether there are compensation tasks or not. During process execution the system will automatically control the reliable execution of exceptions and failures.

As we have to be very carefully in compensating tasks (not all tasks are compensatable) it is possible to compute the safety of modelled processes and to warn the workflow designer in case he has modelled unsafe processes.

The proposed transaction-oriented workflow activity model will be integrated in our prototype system based on an *active* database management system [3]. We are also developing a graphical application development interface to design such workflows with the possibility to generate WADL-code automatically.

References

[1] Workflow Management Coalition Members: Glossary - A Workflow Management Coalition Specification. November 1994.

[2] Eder J., Liebhart W.: A Transaction-Oriented Workflow Activity Model. In: Proc. of the Ninth Int. Symposium on Computer and Information Sciences, Antalya, Turkey, Nov. 1994.

[3] Eder J., Groiss H., Nekvasil H.: A Workflow System Based on Active Databases. 9th Austrian-Hungarian Informatics Conference, Austria, 1994.

[4] McCarthy D.R., Sarin S.K.: Workflow and Transactions in InConcert. Bulletin of the Technical Committee on Data Engineering., Vol. 16, No. 2, June 1993.

[5] Rusinkiewicz M., Shet A.: On Transactional Workflows. Bulletin of the Technical Committee on Data Engineering, Vol. 16, No. 2, 1993.

[6] Saastamoinen, H.T.: Rules and Exceptions. In Information Modeling and Knowledge Bases IV: Concepts, Methods and Systems, H. Kangassalo et al. Eds. IOS Press, Amsterdam, 1993, pp. 271-286

[7] Saastamoinen, H.T. et al.: Exception Handling in Office Information Systems, In Proc. of the Third International Conference on Dynamic Modeling of Information Systems, Noordwijkerhout, 1992.

[8] Moss J.E.B.: Nested Transactions: An Approach to Reliable Distributed Computing. MIT Press, Cambridge, MA, 1985.

[9] Weikum G., Scheck H.-J.: Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In: [22].

[10] Garcia-Molina H., Salem K.: SAGAS. Proc. of ACM SIGMOD Conf. on Management of Data, 1987.

[11] Buchmann A. et al.: A Transaction Model for Active Distributed Object Systems. In: [22].

[12] Elmagarmid A.K., Leu Y., Litwin W. Rusinkiewicz M.: A Multidatabase Transaction Model for InterBase. Proc. of the 16th VLDB Conf. Brisbane, Australia 1990.

[13] Bright M. W., Hurson A. R., Pakzad S. H.: A Taxonomy and Current Issues in Multidatabase Systems. IEEE Computer, March, 1992.

[14] Bukhres O., Kuehn E., Forst A.: General Purpose Work Flow Languages. To appear in: Int. Journal on Parallel and Distributed Databases, 1995.

[15] Dayal U., Hsu H., Ladin R.: A Transactional Model for Long-Running Activities. Proc. of the 17th Int. Conf. on VLDBs, Barcelona, Sept. 1991.

[16] Wächter H. et al.: The Contract Model. In [22].

[17] The Workflow Reference Model. Workflow Management Coalition Specification Document, Version 0.6, June 1993.

[18] Breitbart Y., et al.: Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows. SIGMOD RECORD, Vol. 23, No. 3, Sept. 1993.

[19] Liebhart W.: A Formal Description of the Workflow Activity Description Language WADL. Tech. Report, University of Klagenfurt, Austria, 1994.

[20] Chrysanthis P. K., Ramamritham K.: ACTA: The Saga Continues. In: [22].

[21] Geppert A., Dittrich K.: Rule-Based Implementation of Transaction Model Specifications. In Paton N., Williams M. (eds.): Rules in Database Systems. Workshops in Computing, Springer, 1993.

[22] Elmagarmid A. K.: Database Transaction Models for Advanced Applications. Morgan Kaufmann, 1992.