Semantics and Architecture of Global Transaction Support in Workflow Environments♣

Paul Grefen, Jochem Vonk, Erik Boertjes, Peter Apers
Center for Telematics and Information Technology
University of Twente
{grefen,vonk,boertjes,apers}@cs.utwente.nl

Abstract

In this paper, we present an approach to global transaction management in workflow environments. The transaction mechanism is based on the well-known notion of sagas, but extended to deal with arbitrary process structures including cycles and savepoints that allow partial compensation. We present a formal specification of the transaction model and transaction management mechanisms in set and graph theory, providing clear, unambiguous transaction semantics. The specification is straightforwardly mapped to a modular architecture, the implementation of which is applied in the prototype of a commercial workflow management system. The loosely-coupled nature of the resulting system allows easy distribution using middleware technology.

1. Introduction

Advanced information technology support for process-centered environments like workflow management applications has widely been marked as an important field of research and development. In this context, extended transaction mechanisms are considered a prerequisite to provide high-level semantics for complex, long-running processes like workflows (see e.g. [Hsu93, Alo97, Cic98]). Most existing extended transaction models and systems implementing these models, however, have complex semantics with an operational, informal specification. This clearly limits their applicability in complex application scenarios. Also, they are mostly used in prototype implementations in research contexts only.

In this paper, we address this problem by bridging the gap between formal specification and practical application of high-level transaction management for workflow environments. The transaction model used in the presented approach features relaxed transactional properties and rollback through compensation, as required for long-living, co-operative processes. It is based on the existing saga model, but is applicable to general process structures including cycles and adds the notion of partial compensation. We present a formalization in set and graph theory of both high-level transaction model concepts and transaction management algorithms. This formalization provides clear semantics for the operational aspects of the transaction model. These semantics are not obvious from informal descriptions in complex scenarios, which are common in process-centric environments like workflow management applications. Optimization aspects as described in this paper further complicate matters semantically and thus strengthen the need for formal semantics. The formal ingredients used in the approach are of an well-accepted nature, thus allowing for practical use of the presented work. We show how the formal function specification can easily be mapped to a system architecture.

The model and mechanisms presented in this paper have been applied in the global transaction support developed in the WIDE (Workflow on Intelligent Distributed database Environment) ESPRIT project. In this project, advanced database technology is developed to support next-generation process-oriented applications like workflow management [Cer97, Gre99]. One of the major parts of the database technology developed in WIDE is a two-level transaction management subsystem, which is informally described in [Gre97, Gre99]. The upper level of the subsystem caters for global transactions as formally described in this paper. The prototype system has been applied in real-world insurance and healthcare applications.

In short, the contribution of this paper is threefold. Firstly, we extend the well-known basic saga model to deal with complex process structures and partial compensation. In doing so, we obtain an extended transaction model that is well usable in practical workflow contexts. Secondly, we show that it is possible to provide a precise though simple formal specification of corresponding

♣ *The work presented in this paper is supported by the European Commission in the WIDE project (ESPRIT No. 20280). Partners in WIDE are Sema Group and Hospital General de Manresa in Spain, Politecnico di Milano in Italy, ING Bank and University of Twente in the Netherlands.*

advanced transaction management mechanism. Thirdly, we demonstrate that an implementation of this mechanism can be integrated into a loosely-coupled workflow management architecture that allows flexible distribution.

The structure of this paper is as follows. We first discuss related work in Section 2. In Section 3, we informally discuss the WIDE transaction model as the context for the formal treatment in the sequel of this document. Section 4 discusses the definition of WIDE global transactions in terms of graphs and functions that construct these graphs during transaction execution. Section 5 presents the handling of abort situations through the generation of compensating global transactions. The architecture supporting the algorithms presented in this paper is discussed in Section 6. We present both a general, abstract architecture and a concrete architecture in the context of the FORO workflow management system. The paper is ended with conclusions and further work.

2. Related work

High-level transaction models have been given considerable attention in the past decade, see for example [Elm92] for an overview. Typical examples of advanced transaction models for long-running processes are nested transactions [Day90, Day91], multi-level transactions [Wei91], and sagas [Gar87]. General frameworks have been constructed, like ACTA [Chr94], that provide a conceptual framework for constructing or analyzing extended transaction models. Low-level mechanisms have been proposed to provide a 'tool-box' approach to advanced transaction management, e.g. the ConTracts approach [Reu95].

In the WIDE project, an orthogonal two-level transaction model is used to effectively model both long-running processes and relatively short-running subprocesses [Gre97]. In this paper, we focus on the semantics of the upper level of this model. This level is a transaction model with relaxed ACID properties using a compensation mechanism for rollback operations related to sagas as presented in [Gar87], but extended with a flexible notion of partial compensation. A hybrid transaction model is also discussed in [Che97], in which transaction hierarchies are described that contain flat structured transactions. Dependencies between hierarchies are supported by cross-hierarchy failure handling. In the WIDE approach, nested processes with flat, structured levels are supported in the lower level of the transaction model. Dependencies between nested constructs are represented in the upper level of the transaction model, consisting of arbitrary process graphs. Apart from differences in the transaction model itself, the main difference between the work in [Che97] and that in this paper, is that we aim at a formal specification of the semantics of transaction mechanisms, instead of using text and pseudo-code descriptions.

An advanced transaction compensation mechanism is discussed in [Kry96] in the context of a multi-level transaction model. The emphasis is on determining the horizon (dynamic applicability) of compensation in nested structures, whereas we concentrate on constructing compensation patterns for arbitrary process graphs. Our approach to partial compensation can be used to bound the effects of compensation. Again, our work contrasts to the work in [Kry96] in the fact that we provide a complete formal specification of compensating transaction management mechanisms, whereas most other work relies on informal descriptions.

Formal specification of transaction semantics has been addressed by a number of researchers. In [Kor90], a formal treatment of compensating transactions is given. The focus is on the correctness of individual compensating transactions. The work we present in this paper focuses on the construction of complex compensating graphs (global transactions) consisting of predefined compensating transactions. As such, it can be seen as complementary to the work in [Kor90].

As it has been widely recognized that transactional semantics are an important aspect of workflow management, transaction mechanisms dedicated for workflow environments have been studied in recent years. A number of proposals is discussed in [Hsu93] and [Cic98]. A characterization of transactions in workflow contexts is given in [Alo97], stressing that advanced transaction management is indeed required, but not yet offered by existing systems. Recent work that focuses on high-level transaction management for workflow environments has been performed in the Exotica project [Alo96]. Like the global transactions discussed in this paper, the Exotica approach uses compensation to perform rollback operations, as originally described by the saga model [Gar87]. The compensation mechanisms in Exotica are of a static nature, however, and lack a formal specification as given in this paper. Handling of compensation is also considered in the OpenPM project at Hewlett-Packard [Dav95], but a formal background is not given.

Formal specification of transaction mechanisms contributes to the assessment of the correctness of these mechanisms and of the applications using these mechanisms. This paper addresses the aspect of compensation semantics in process-centered applications like workflow management. As such, it can be used to formally assess the correctness of transactional aspects of workflow systems using the transaction management approach developed in the WIDE project [Gre97]. More general observations with respect to correctness issues in workflow management are presented in [Kam96] in an informal fashion.

3. Context

In this section, we present the context in which the research described in the sequel of this paper has been conducted. We first discuss the overall two-layer transaction model as it has been adopted in the WIDE project. Then we focus on the upper level of this model by informally describing global transaction management. A formal specification of global transaction management presenting precise semantics follows in the sequel of this paper.

3.1 Two-layer transaction model

In the WIDE transaction model [Gre97], two orthogonal transactional layers are identified to deal with the different requirements of high-level (long-living) and low-level (relatively short-living) business processes. The model has been designed to cater for process-centric applications like workflow management, where transactions of long duration and a high level of cooperativeness are required.

The bottom layer of the WIDE transaction model provides *local transactions* with strict transactional (ACID) requirements [Boe98]. Local transactions coincide with business transactions in the business process application, i.e., parts of a process that have atomic behavior from an application-oriented view. The set of business transactions in an application forms a partition of the complete process. Details of local transactions are not relevant in this paper.

The top layer provides *global transactions* with relaxed transactional properties. In the global transaction layer, local transactions are used as black box atomic processes (steps in the global transaction). Relaxation of transactional properties is reflected in relaxed notions of isolation and atomicity. This relaxation caters for the needs of cooperative workflow processes above the business transaction level. Isolation in the global transaction is relaxed by making intermediate results in between steps visible to the context of the global transaction (i.e. local transactions commit their results to the shared database). To obtain relaxed atomicity, rollback operations in the global transaction layer should have application-specified semantics instead of the database-oriented semantics of the local transaction model. For these reasons, we have chosen a global transaction model that is heavily based on the saga transaction model [Gar87], extended with a flexible mechanism for partial rollback.

As in the saga model [Gar87], relaxed atomicity is obtained by using a compensation mechanism to provide rollback functionality. Rollback of global transactions is performed by executing a compensating global transaction that consists of compensating local transactions. A compensating local transaction is inserted for each local transaction that has been committed in the failing global transaction. Running, not-yet-committed steps can simply be aborted, as they are atomic local transactions. Operations in compensating steps are application-dependent and have to be specified by the application designer.

Steps in a workflow can be marked as *savepoints*. A savepoint is a step in a workflow from where forward recovery can be safely started after a global abort situation (comparable to compensation points in the OpenPM approach [Dav95]) and hence a point where compensation can end. As such, savepoints provide ways to flexibly specify partial rollback strategies dealing with abort situations occurring in different parts of a global transaction. Unlike savepoints in the saga model [Gar87], global transaction savepoints do not require making checkpoints. Like the functionality of compensating steps, placement of savepoints in a global transaction is fully application-dependent.

3.2 Global transaction model

A WIDE global transaction specification consists of a rooted directed graph of global transaction steps (local transactions). The *specification graph* is rooted as it can have only one starting step. It can have an arbitrary number of ending steps. It can contain various types of and/or-splits, and/or-joins, and cycles to cater for complex process structures as found in workflow applications (conforming to the WIDE conceptual workflow model [Cas96, Gre99]). The graph represents the possible execution orders of the steps in the application process.

An example specification graph[1] from a travel agency application is shown in the top of Figure 1. The graph models a process for selling and invoicing trips. Start of the process is local transaction 'sales', in which a trip is selected and configured. From there, a trip can either be cancelled or booked (or-split). After booking, two subprocesses proceed in parallel (and-split). The financial department calculates, files, invoices, and checks for incoming payment. The travel department prepares tickets and vouchers and sends them to the customer. Invoicing and payment checking may have to be iterated when a payment has not yet been received. Sending the travel documents cannot take place before payment has been received. Local transaction 'sales' has been specified as a savepoint.

Instantiations (executions) of a global specification graph are specified in an *execution graph* of a global transaction. As we can have or-splits and cycles in a global transaction specification, the specification graph and the execution graph of a global transaction are different in general: paths that are not executed in an or-split are

[1] *For reasons of clarity, we use a simplified version of the WIDE process notation [Cas96, Gre99], in which a diamond with a '1'-symbol indicates an or-split or or-join, and a diamond with an 'n'-symbol an and-split or and-join. An 's'-symbol in a process step denotes a savepoint.*

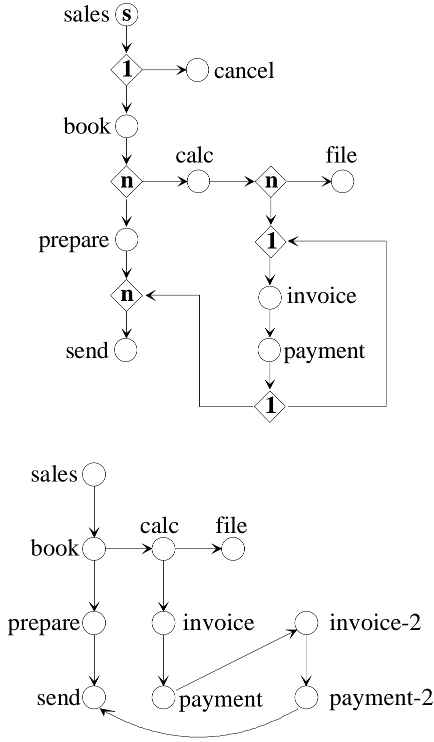


Figure 1: Example specification and execution graphs

not in the execution graph and cycles are replaced by the instantiation of the iteration. Execution graphs are thus rooted directed acyclical graphs (RDAGs).

The bottom of Figure 1 shows a completed execution graph of the example specification graph. In this execution, the 'cancel' local transaction has not been executed and the 'invoice-payment' iteration has been executed twice. To reason about the dynamic properties of a global transaction in execution, the execution graph is considered, not the specification graph.

In the next section, we present a formalization of global transactions and their execution. This formalization serves as the basis for the compensation algorithms presented in the sequel of this paper.

4. Transaction definition and execution

This section formalizes the definition and execution of global transactions. Important elements are the definition of global transaction execution graphs, basic predicates and functions defined on these graphs, and construction functions that modify the graphs during transaction execution. The treatment in this chapter is the basis for the rollback (compensation) algorithms in the next chapter.

We start with describing local transactions, which are the 'building blocks' for global transactions.

4.1 Local transactions

In the WIDE transaction model, local transactions are atomic units of execution [Gre97, Boe98]. As such, instantiations of local transactions form the elementary steps in global transactions. In the sequel of this paper, we use the term 'local transaction' to denote 'instantiation of a local transaction'. Local transactions are defined in the domain T_{loc}, the set of node labels in the execution graph (constructed by suffixing node labels from the specification graph with local transaction instantiation numbers; for clarity, we omit these numbers in the sequel of this paper).

A number of unary predicates is defined on the domain of local transactions. Predicates *started* and *committed* indicate the state of a local transaction. They denote whether the execution of a local transaction has begun, respectively, has completed. Predicates *safe* and *dummy* denote semantic properties of a local transaction. Predicate *safe* tests whether a local transaction is a global savepoint; predicate *dummy* tests whether a local transaction has dummy semantics (i.e., does not have any effect). So we have:

$$started(v) : T_{loc} \rightarrow bool$$
$$committed(v) : T_{loc} \rightarrow bool$$
$$safe(v) : T_{loc} \rightarrow bool$$
$$dummy(v) : T_{loc} \rightarrow bool$$

Two binary predicates are defined on the domain of local transactions. Predicate *trig* denotes that the first transaction has dynamically triggered the second transaction. Predicate *equal* denotes that two local transactions have equal semantics (but are not necessarily in the same execution state):

$$trig(v,w) : T_{loc} \times T_{loc} \rightarrow bool$$
$$equal(v,w) : T_{loc} \times T_{loc} \rightarrow bool$$

The state-related predicates are not independent. A transaction that is committed is started as well. Two local transactions can only have a triggered relationship if the first transaction is committed and the second transaction is started:

$$committed(v) \Rightarrow started(v)$$
$$trig(v,w) \Rightarrow committed(v) \wedge started(w)$$

Function *comp* returns the compensating counterpart of the local transaction given as its argument or a dummy transaction if the compensating counterpart does not exist:

$$comp(v) : T_{loc} \rightarrow T_{loc}$$

As remarked above, compensating counterparts of local transactions have to be specified by an application designer. This reduces function *comp* to a simple table

lookup function. Below, we use local transactions as components (atomic steps) in global transactions.

4.2 Execution graph definition

An execution graph of a global transaction models its execution history. It is a directed graph consisting of a set of vertices corresponding to all started local transactions and a set of edges corresponding to the triggering relationship between these local transactions:

$$G_{exec} = \langle V_{exec}, E_{exec} \rangle$$
$$V_{exec} = \{ v \in T_{loc} \mid started(v) \}$$
$$E_{exec} = \{ \langle v, w \rangle \in V_{exec} \times V_{exec} \mid trig(v, w) \}$$

We introduce a number of basic operations on execution graphs that we require in the sequel of this paper. Starting points of an execution graph are nodes without incoming edges. Ending points are nodes without outgoing edges:

$$start(G) = \{ v \in G.V \mid \neg(\exists w \in G.V)(\langle w, v \rangle \in G.E) \}$$
$$end(G) = \{ v \in G.V \mid \neg(\exists w \in G.V)(\langle v, w \rangle \in G.E) \}$$

Function *preds* calculates the direct predecessors of a subgraph, i.e., the set of vertices that have outgoing edges ending in starting points of the subgraph:

$$preds(G, S) = \{ v \in G.V \mid trig(v, w) \wedge w \in start(S) \}$$

The active transactions in an execution graph are the local transactions that have not yet been committed. The active edges are the edges ending in nodes corresponding to active transactions:

$$activev(G) = \{ v \in end(G) \mid \neg committed(v) \}$$
$$activee(G) = \{ \langle v, w \rangle \in G.E \mid \neg committed(w) \}$$

As discussed in Section 3.1, cycles in process specification graphs are rolled out in execution graphs, so execution graphs are acyclic. We require that a correct execution graph has exactly one starting point and at least one ending point. These constraints on the structure of compensation graphs can be expressed as shown below (where *card* is the set cardinality function):

$$(\forall v \in G.V)(\neg transtrig(G.V, v, v))$$
$$transtrig(V, v, w) \Leftrightarrow$$
$$trig(v, w) \vee (\exists x \in V)(trig(v, x) \wedge transtrig(V, x, w))$$
$$card(start(G)) = 1 \wedge card(end(G)) \geq 1$$

Having completed the preliminaries, we can turn to constructing execution graphs during global transaction execution.

4.3 Execution graph construction

During the execution of a global transaction, its execution graph has to be maintained to properly reflect the status of the execution. Basically, there are four operations on execution graphs corresponding with events in the lifecycle of a global transaction:

1. Creation of a new empty graph when a new global transaction is started.
2. Addition of a new vertex (and corresponding edges) to a graph when a new local transaction is started.
3. Replacement of a vertex (and corresponding edges) in the graph when a running local transaction is completed.
4. End of a global transaction.

These four operations are described formally below, using the concepts introduced in the previous section.

Starting a global transaction. Starting a new global transaction corresponds to creating an empty execution graph:

$$startgt = \langle \emptyset, \emptyset \rangle$$

After a global transaction has been created, local transactions can be started in its context.

Starting a local transaction. Starting a new local transaction corresponds to adding a new vertex *w* to the graph and connecting it to its set of predecessors *P*. The predecessors correspond to the completed local transactions that triggered the new local transaction.

$$startlt(G, P, w) = \langle addv(G.V, w), adde(G.E, P, w) \rangle$$
$$addv(V, w) = V \cup \{w\}$$
$$adde(E, P, w) = E \cup \{ \langle v, w \rangle \mid v \in P \}$$

Ending a local transaction. Ending a local transaction means replacing the corresponding vertex *v* in the graph by a new vertex *w* that is equal except for its state[2]:

$$endlt(G, v) = \langle changev(G.V, v), changee(G.E, v) \rangle$$
$$changev(V, v) = \left\{ \begin{array}{l} w \in T_{loc} \mid (w \in V \wedge w \neq v) \vee \\ (equal(w, v) \wedge committed(w)) \end{array} \right\}$$
$$changee(E, v) = \{ \langle x, y \rangle \in E \mid y \neq v \} \cup$$
$$\left\{ \langle x, w \rangle \in T_{loc} \times T_{loc} \middle| \begin{array}{l} \langle x, v \rangle \in E \wedge \\ equal(w, v) \wedge committed(w) \end{array} \right\}$$

---

[2] *Note that it is not possible to simply update the state of a vertex, given our declarative approach to algorithm specification.*

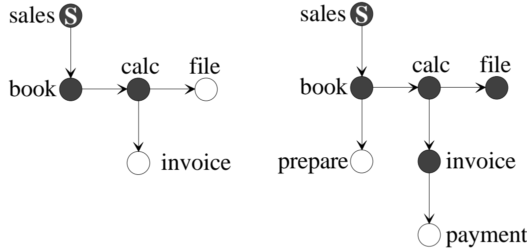


Figure 2: Partial execution graphs

Figure 2 shows two partial execution graphs, resulting from the execution of the specification graph in Figure 1. In the left-hand side graph, three steps have been completed (indicated by gray shading). Steps 'file' and 'invoice' are currently being executed. This graph has been constructed by one *startgt*, five *startlt*, and three *endlt* operations as specified above. In the right hand side graph, these two steps have been completed and steps 'prepare' and 'payment' are being executed (two more *startlt* and two more *endlt* operations have been executed).

Ending a global transaction. Ending a global transaction does not change the execution graph:

$$endgt(G) = G$$

The operations discussed in this section are used in normal global transaction processing, i.e. without the occurrence of global aborts. Now we turn our attention to handling global abort situations.

5. Global transaction compensation

In this section, we present the algorithms used for compensating global transaction when a global abort situation arises. We start with an informal introduction to global transaction compensation. Then, we formally discuss the generation of complete and partial compensation graphs, as required to perform complete, respectively, partial rollback (abort) of global transactions.

In the formal treatment, we first present the compensation driver, i.e., the high-level function used to invoke a global compensation. Next, we present the algorithms for the construction of complete and partial compensation graphs. Finally, we show how compensation graphs can be made more efficient by filtering out unnecessary steps.

5.1 Informal introduction

An example of a global transaction execution requiring global rollback is shown in the top of Figure 3. Here we see an execution graph corresponding to the specification graph in Figure 1, at a point where the global transaction has been partly been completed. The grayed steps have been committed; two steps are being executed. Local

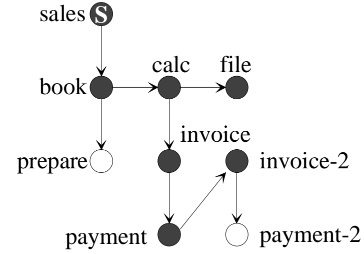


Figure 3: Partial execution and compensation graphs

transaction 'sales' has been specified to be a savepoint. Now assume that running local transaction 'payment' raises an error that requires global rollback. Then all running local transactions ('prepare' and 'payment-2') are aborted (using the local transaction mechanism). Next, the execution graph needs to be compensated from the point where the error occurred until a savepoint is encountered (to the start of the graph if none is found). This means that compensation is performed by executing the dynamically constructed global transaction depicted in the bottom of Figure 3. In this figure, the prefix 'c' for a local transaction indicates its compensating counterpart. The details of the construction of this example compensating transaction are discussed in the sequel of this paper. Note that a very simple example is chosen for reasons of clarity. In general, compensating global transactions can have a complex structure consisting of many local transactions (a more complex example follows in this paper).

5.2 Compensation driver

A compensation request is invoked by function *abort*, which is parameterized with the requested abort mode *m* (complete or partial), the identifier *n* of the global transaction to be aborted, and the identifier *v* of the global transaction step that caused the rollback. The function returns the name of the compensating global transaction and the list of restart points in the original global transaction. Restart points are points in an execution graph from where forward execution can take place after compensation. Function *abort* performs the following steps:

1. It retrieves the execution graph of the aborted global transaction from persistent storage.
2. It computes the compensating graph plus the restart points in the original graph.
3. It generates a name for the compensation graph and stores the specification of the graph into persistent storage.

So we have:

$$abort(m,n,v) =$$
$$storespec(gcomp(m, getexec(n), v), newid(n))$$

Function *gcomp* selects between complete and partial compensation based on the value of parameter *m*. In case of a complete compensation, there are no restart points (the entire original transaction has to be redone):

$$gcomp(m,G,v) = \begin{cases} \langle ccomp(G), \varnothing \rangle & \text{if } m = complete \\ pcomp(G,v) & \text{if } m = partial \end{cases}$$

Functions *ccomp* and *pcomp* are discussed in detail below.

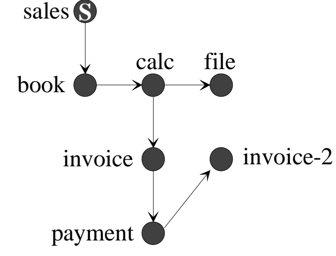
5.3 Complete compensation

When rollback of a global transaction is required, a compensating global transaction has to be constructed. This compensating global transaction is based on a compensating counterpart of the execution graph of the global transaction. In this section, we discuss calculating complete compensation graphs, i.e., compensation graphs that 'cover' the complete execution graph. A complete compensation graph is constructed from an execution graph in four steps:

1. The active transactions are removed from the graph; they have been rolled back by the local transaction mechanism and are of no concern to the global transaction mechanisms.
2. The vertices in the graph are replaced by their compensating counterparts to obtain the functional elements for the compensating global transaction.
3. The edges in the graph are reversed to obtain the correct flow control for the compensating global transaction (the inverse of the flow control of the 'original' global transaction).
4. If the graph resulting from the previous steps contains multiple starting points, a unique starting point is added to the graph.

This four-step process is reflected in the formula below. Each of the steps is described in detail in the sequel.

$$ccomp(G) = addstart(compe(compv(strip(G))))$$



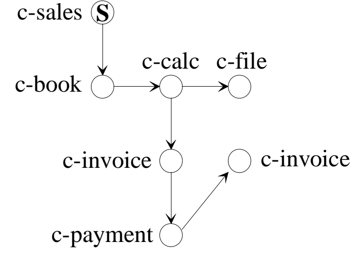


Figure 4: Steps 1 and 2 in complete compensation graph construction

Stripping an execution graph. An execution graph is 'stripped' from its active transactions by removing the vertices corresponding to active local transactions plus edges ending in these vertices:

$$strip(G) = \langle G.V - activev(G), G.E - activee(G) \rangle$$

The stripped version of the execution graph from Figure 3 is shown in the top of Figure 4.

Compensating vertices. Vertices are compensated by exchanging 'original' vertices by their compensating counterparts and reorganizing the edges in the graph to point to the new vertices. Function *compv* implements this functionality:

$$compv(G) = \langle exchangev(G.V), exchangee(G.E) \rangle$$
$$exchangev(V) = \{ v \in T_{loc} \mid v = comp(w) \wedge w \in V \}$$
$$exchangee(E) =$$
$$\left\{ \langle v,w \rangle \in T_{loc} \times T_{loc} \,\middle|\, \begin{array}{l} \langle x,y \rangle \in E \wedge \\ v = comp(x) \wedge w = comp(y) \end{array} \right\}$$

The result of applying this second step to the stripped example execution graph is shown in the bottom of Figure 4. This graph contains the required compensating actions, but not the required flow control.

Inverting edges. Edges in a graph are inverted by simply exchanging their start and end points:

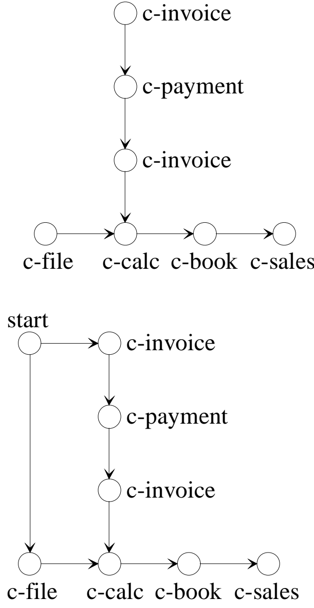


Figure 5: Steps 3 and 4 in complete compensation graph construction

$$compe(G) = \langle G.V, invert(G.E)\rangle$$
$$invert(E) = \{\langle v, w\rangle | \langle w, v\rangle \in E\}$$

The effect of applying edge inversion to the graph in Figure 4 is depicted in the top of Figure 5 (note that the graph has been graphically reordered to obtain the usual top-left to bottom-right process flow). This graph contains both the required functionality and flow control, but lacks a unique starting point.

Ensuring a single starting point. A single starting point for the graph is ensured by adding a new vertex if the 'original' graph has multiple starting point. Edges are added between the new vertex and the 'original' starting points. The new starting point has dummy semantics, represented by the empty step $t_\varnothing$.

$$addstart(G) = \langle G.V \cup addv(G), G.E \cup adde(G)\rangle$$
$$addv(G) = \begin{cases} \varnothing & \text{if } card(start(G)) = 1 \\ \{t_\varnothing\} & \text{if } card(start(G)) > 1 \end{cases}$$
$$adde(G) =$$
$$\begin{cases} \varnothing & \text{if } card(start(G)) = 1 \\ \{\langle t_\varnothing, v\rangle | v \in start(G)\} & \text{if } card(start(G)) > 1 \end{cases}$$

The graph in the top of Figure 5 contains two starting points ('c-file' and c-invoice'). Therefore, a single starting point is added as shown in the bottom of the figure. The graph represents the complete compensating global transaction.

5.4 Partial compensation

Partial compensation of a global transaction requires compensation of a part of the execution graph, starting from a rollback point and delimited by the proper savepoints in the graph. A simple example has already been presented in Figure 3, where task 'sales' of the execution graph in not compensated in the compensation graph because it is a savepoint.

As execution graphs can be arbitrarily complex, the situation is usually not as simple as depicted in Figure 3. The problem is finding the proper subgraph of the execution graph to be compensated, taking into account savepoints and forward and backward dependencies between tasks in the graph. This section presents the algorithms required to calculate the appropriate subgraph of the execution graph.

Calculating a partial compensation graph. A partial compensation graph is constructed by first calculating the proper subgraph and next using the complete compensation algorithm of Section 5.3:

$$pcomp(G, v) =$$
$$\langle ccomp(sgraph(G, v)), preds(sgraph(G, v))\rangle$$
$$sgraph(G, v) = extend(\langle\{v\}, \varnothing\rangle, G)$$

Calculating a subgraph to be compensated. The subgraph to be compensated is calculated from the vertex where the partial abort originated from. From this vertex, we first construct a subgraph consisting of predecessors of the vertex until savepoints are encountered (extending the subgraph backward). Next, we extend this subgraph forward by including all vertices reachable from the subgraph.

$$extend(S, G) = extforw(extback(S, G), G)$$

Extending a subgraph backward is performed in a recursive fashion until the subgraph has reached a stable size, i.e., doesn't grow anymore:

$$extback(S, G) =$$
$$\begin{cases} S & \text{if } S = backstep(S, G) \\ extback(backstep(S, G), G) & \text{otherwise} \end{cases}$$
$$backstep(S, G) = \langle backstepv(S, G), backstepe(S, G)\rangle$$
$$backstepv(S, G) =$$
$$\left\{ v \in G.V \middle| v \in S.V \vee \begin{pmatrix} \langle v, w\rangle \in G.E \wedge \\ w \in S.V \wedge \neg safe(v) \end{pmatrix} \right\}$$
$$backstepe(S, G) = \{\langle v, w\rangle \in G.E | w \in S.V \wedge \neg safe(v)\}$$

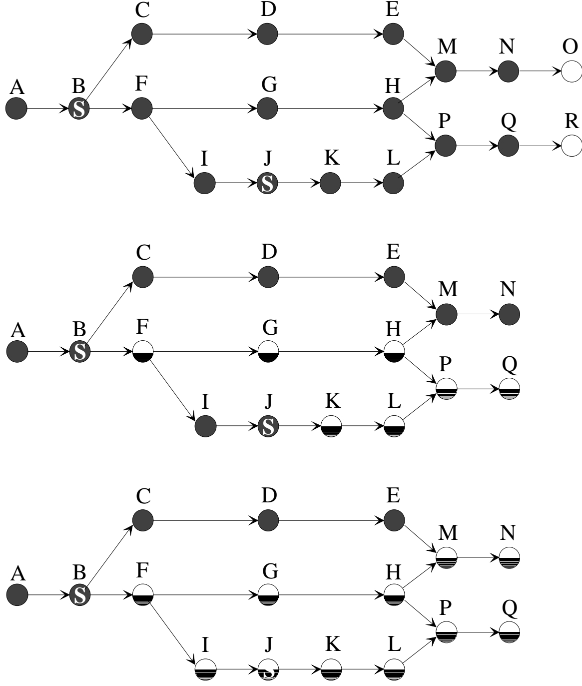


Figure 6: Execution graph, backward extension, and forward extension

Extending a subgraph forward is performed in a recursive fashion until the subgraph has reached a stable size:

$$extforw(S,G) =$$

$$\begin{cases} S & \text{if } S = forwstep(S,G) \\ extforw(forwstep(S,G),G) & otherwise \end{cases}$$

$$forwstep(S,G) =$$

$$\langle forwstepv(S,G), forwstepe(S,G)\rangle$$

$$forwstepv(S,G) =$$

$$\{v \in G.V \mid v \in S.V \vee (\langle w,v\rangle \in G.E \wedge w \in S.V)\}$$

$$forwstepe(S,G) = \{\langle v,w\rangle \in G.E \mid v \in S.V\}$$

Figure 6 shows an example of subgraph calculation. In the top of the figure, an execution graph is depicted. Steps B and J are savepoints (indicated by the S symbols) and steps O and R are currently being executed, i.e. started but not yet committed. Now suppose step R invokes a global rollback operation. Then first, steps O and R are aborted. Next, backward extension takes place from step Q (being the direct predecessor of step R), as depicted in the middle graph of the figure by the half-grayed steps. Informally, backward extension means searching for all predecessors of a given step until savepoints are encountered. Finally, forward extension takes place as shown in the bottom graph. Informally, forward extension means finding all successors of a given subgraph. Note that the subgraph to be compensated includes savepoint J, as this point is covered by forward extension. Figure 7 shows the final compensation graph, obtained by applying the algorithms of Section 5.3 to the calculated subgraph. In this figure, a step X' denotes the compensating counterpart of step X.

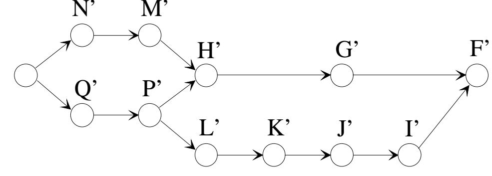


Figure 7: Compensation graph corresponding with Figure 6

5.5 Compensation graph filtering

Compensation graphs constructed as discussed above can be made more efficient by filtering out steps that are semantically unnecessary. Two typical classes of unnecessary steps are steps with dummy semantics and steps with idempotent effects in sequences. Below, we elaborate the first class; the second class can be handled in a similar manner. More advanced types of filtering are possible too, e.g., replacing sequences of compensating steps by composite compensation steps (steps that undo the effects of multiple 'original' steps in a more efficient manner).

Local transactions may not have a compensating counterpart because an inverse transaction has not been specified by the application designer (or simply does not exist). In constructing a compensating graph as discussed above, these transactions are replaced by empty (dummy) compensating transactions. For reasons of efficiency in compensation execution, these empty compensation transactions can be removed from the constructed compensation graph by contracting it with respect to the nodes corresponding to empty compensation actions (dummies). This functionality is specified as follows:

$$filter(G) = \langle filterv(G), filtere(G) \cup newe(G)\rangle$$

$$filterv(G) = \{v \in G.V \mid \neg dummy(v)\}$$

$$filtere(G) =$$

$$\{\langle v,w\rangle \in G.E \mid \neg dummy(v) \wedge \neg dummy(w)\}$$

$$newe(G) =$$

$$\left\{ \langle v,w\rangle \in G.V \times G.V \;\middle|\; (\exists x \in G.V) \begin{pmatrix} \langle v,x\rangle \in G.E \wedge \\ \langle x,w\rangle \in G.E \wedge \\ dummy(x) \end{pmatrix} \right\}$$

We base an example on the compensation graph of Figure 7. Assume that steps P and J do not have compensating counterparts, i.e., P' and J' are empty actions. Then these empty actions can be removed from the graph, resulting in the compensation graph shown in Figure 8.

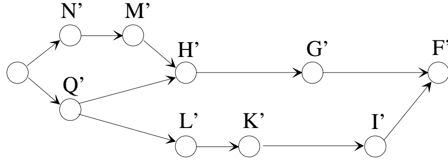


Figure 8: Reduced compensation graph of Figure 7

Function *filter* can easily be applied in the compensation driver discussed in Section 5.2, resulting in the following specification of function *abort*:

$$abort(m,n,v) =$$
$$storespec(filter(gcomp(m, getexec(n), v), newid(n)))$$

6. Architecture

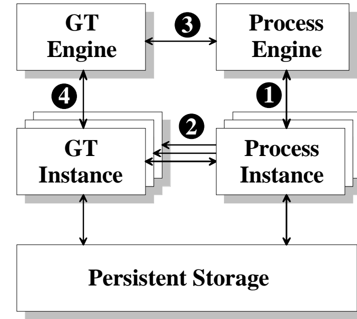
In this section, we present a system architecture designed to support the transaction mechanisms discussed in the previous sections. We discuss the architecture in both an abstract and a concrete version. The global transaction support (GTS) subsystem is designed to serve in general process-oriented systems requiring high-level transactional semantics. This approach is reflected below by discussing an abstract system architecture supporting global transaction management. Next, the specific implementation in the FORO workflow management system is discussed, as realized in the WIDE project. FORO is a commercial WFMS [For98] marketed by Sema Group.

6.1 Abstract architecture

The abstract architecture of the GTS and its environment are depicted in Figure 9a. The left side of the figure depicts the GTS system that serves as a 'transaction semantics server'. The right-hand side of the figure shows the client process enactment system that uses the GTS system. At the bottom is the persistent storage that holds non-volatile information like global transaction specification and execution graphs; this may be the same storage for GTS and client system, but not necessarily so.

The client system consists of a process engine and a number of process instance objects. The process engine interprets a process specification and performs scheduling among process instances. Each process instance object represents a separate invocation of a process specification. It is controlled by the process engine using interface ❶ (see figure). The object holds all relevant status information of the process instance. Process instance objects are created and deleted dynamically at process invocation, respectively, process termination.

The GTS system consists of a GT engine and a number of GT instance objects. The engine provides global rollback functionality as described above. Each GT instance object represents a running global transaction and holds all relevant status information, most importantly the execution graph of the global transaction. Like process instance objects, GT instance objects are created and deleted dynamically.



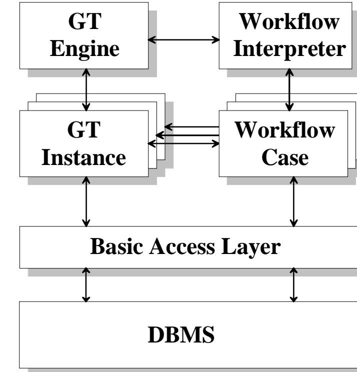


Figure 9 a and b: Abstract GTS and FORO GTS Architectures

Process instance and GT instance objects are coupled one-to-one, as a process instance corresponds with a global transaction instance. During its life cycle, a process instance object informs its GT instance object of all relevant process events through interface ❷, e.g. the start of a process step and the end of a process step. These events are used by the GT instance object to update the status of the global transaction, as described by functions *startgt*, *startlt*, *endlt*, and *endgt* in Section 4.3.

When the process engine signals a global abort for a process instance, the GT engine is informed about this through interface ❸. This corresponds with invoking function *abort* as specified in Section 5.2. Next, the GT engine retrieves the execution graph of the global transaction from the corresponding GT instance object, calculates the required compensating global transaction, and stores the specification of this transaction through the GT instance object (using interface ❹ twice). It then informs the process engine about the name of the compensating transaction and the restart points in the original transaction using interface ❸. This corresponds with the steps specified in the compensation driver (function *abort* in Section 5.2). The process engine executes the compensating transaction and then restarts the original transaction at the indicated restart points.

Note that the efficiency of the algorithms of the GT engine (the performance of the module) is not too relevant if within reasonable bounds: the engine deals with long-running workflow processes in distributed environments. Efficient results of the GT engine algorithms (i.e., efficient compensation graphs) are relevant, however, as they determine the workload for the process engine. The former observation enables straightforward implementation of the algorithms presented in this paper, without too much attention to optimization. The latter observation is the reason why we pay attention to compensation graph filtering in the algorithms.

6.2 FORO architecture

In the context of the WIDE project, an implementation of the GTS is used in a prototype of the next generation of the FORO workflow management system architecture [Cer97, Gre98, Gre99]. FORO has been equipped with both layers of the WIDE transaction support to provide transaction management functionality with both a high level of expressiveness and a high level of flexibility, as required by complex workflow application settings.

The architecture of the GTS in the FORO context is shown in Figure 9b. This architecture is directly based on the abstract architecture in Figure 9a. The role of the process engine in the abstract architecture is taken by the FORO workflow interpreter. This module interprets workflow specifications in the FORO process description language. Workflow case objects take the role of the process instance objects. Each case object manages the process state of a workflow invocation. As in the abstract architecture, case objects send messages to GT objects to manage their transactional state.

The FORO architecture is implemented in a CORBA environment [OM95] that allows flexible distribution in the architecture [Gre98]. Both GT engine and GT objects are implemented as CORBA objects. This allows for a flexible coupling of GT engines and workflow engines: if global rollbacks are seldom, one GT engine can serve multiple workflow engines; if global rollbacks are frequent, a workflow engine may use multiple GT engines. The persistent storage consists of a Basic Access Layer (BAL) and a commercial relational DBMS. The BAL 'isolates' the workflow environment from the DBMS, such that DBMS-specific details are hidden.

The GTS system uses local transactions in a black-box fashion only. As local transactions correspond to DBMS-level transactions [Boe98], this means that the GTS can operate completely orthogonal to the transaction management mechanism of the DBMS. The local transaction mechanism of the FORO environment that is implemented on top of Oracle's transaction manager is thus invisible to the GTS. This independence combined with the isolation by the BAL mentioned above guarantees a high level of portability for the GTS.

The modular approach to transaction management with simple, high-level interfaces and well-defined semantics allows flexible system composition. As such, the resulting system architecture can be considered a federation of workflow and transaction servers, based on middleware services that hide distribution details. Protocols and architecture for this distributed global transaction system are elaborated in [Von99].

7. Conclusions

In this paper, we present the formal specification of a high-level transaction mechanism. The mechanism provides an approach to advanced transaction management in process-centric environments that fulfils requirements of real-world application contexts. The distinction between specification and execution graphs allows effective handling of cyclical process structures. The concept of partial compensation allows for flexibly bounding the effects of compensation. Efficiency aspects are taken into account in the mechanism through the inclusion of compensation graph filtering. The formal specification clearly and unambiguously describes the semantics of the mechanism, which are not obvious in complex partial rollback situations. The simple formalisms used make the approach well digestible, however.

This work shows that it is well feasible to provide formal semantics of real-world advanced transaction management systems, closely coupled to a system architecture. The approach presented is not limited to WIDE global transactions, but can be applied to other transaction models as well. The formal semantics of the transaction mechanisms presented in this paper are both useful for the developers of the transaction management system itself and for advanced users of the underlying transaction model (application designers). The compensation algorithms can be used in workflow design tools to present the designer with automated aids in analyzing the effects of specific global transaction designs (e.g., the scope of rollback in specific situations).

A prototype of the transaction mechanism specified in this paper has been realized in the WIDE project, providing both complete and partial rollback functionality as described in this paper. It has been integrated with the FORO workflow management system, resulting in a flexible, distributed architecture.

The transaction model and mechanisms described in this paper can be extended in a number of ways. Even more flexibility in rollback behavior can be obtained by using dynamic savepoints, i.e., steps that are dynamically assigned the savepoint label based on expressions over the transaction state. Global transactions that are distributed over multiple transaction engines can be supported by a distributed global transaction system, as described in [Von99].

Acknowledgements

Thanks go to Stefano Ceri of the Politecnico di Milano for his comments on the draft version of this paper. All members of the WIDE team are acknowledged for their role in the realization of the architecture described in this paper.

References

[Alo96] G. Alonso et al., "Advanced Transaction Models in Workflow Contexts", *Procs. Int. Conf. on Data Engineering*, 1996, pp. 574-581.

[Alo97] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, "Functionality and Limitations of Current Workflow Management Systems", *IEEE Expert*, Vol. 12, No. 5, 1997.

[Boe98] E. Boertjes, P. Grefen, J. Vonk, P. Apers, "An Architecture for Nested Transaction Support on Standard Database Systems", *Procs. 9th Int. Conf. on Database and Expert System Applications*, Vienna, Austria, 1998, pp. 448-459.

[Cas96] F. Casati et al., *WIDE: Workflow Model and Architecture*, CTIT Technical Report 96-19, University of Twente, 1996.

[Cer97] S. Ceri, P. Grefen, G. Sánchez, "WIDE - A Distributed Architecture for Workflow Management", *Procs. 7th Int. Workshop on Research Issues in Data Engineering*, Birmingham, UK, 1997, pp. 76-79.

[Che97] Q. Chen, U. Dayal, "Failure Handling for Transaction Hierarches", *Procs. 13th Int. Conf. on Data Engineering*, Birmingham, UK, 1997, pp. 245-254.

[Chr94] P.K. Chrysanthis, K. Ramamritham, "Synthesis of Extended Transaction Models using ACTA", *ACM Transactions on Database Systems*, Vol. 19, No. 3, 1994, pp. 450-491.

[Cic98] A. Cichocki, A. Helal, M. Rusinckiewicz, D. Woelk, *Workflow and Process Automation: Concepts and Technology*, Kluwer Academic Publishers, 1998.

[Dav95] J. Davis, W. Du, M. Shan, "OpenPM: an Enterprise Process Management System", *IEEE Data Engineering Bulletin*, Vol. 18, No. 1, 1995.

[Day90] U. Dayal, M. Hsu, R. Ladin, "Organizing Long-Running Activities with Triggers and Transactions", *Procs. 1990 ACM SIGMOD Int. Conf. on Mngmnt. of Data*, Atlantic City, USA, 1990, pp. 204-214.

[Day91] U. Dayal, M. Hsu, R. Ladin, "A Transactional Model for Long-Running Activities", *Procs. 17th Int. Conf. on Very Large Databases*, 1991, pp. 113-122.

[Elm92] A.K. Elmagarmid (Ed.), *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, USA, 1992.

[For98] FORO Web Site, http://dis.sema.es/projects/FORO/index.html, Sema Group, Spain, 1998.

[Gar87] H. Garcia-Molina, K. Salem, "Sagas", *Procs. 1987 ACM SIGMOD Int. Conf. on Management of Data*, USA, 1987, pp. 249-259.

[Gre97] P. Grefen, J. Vonk, E. Boertjes, P. Apers, "Two-Layer Transaction Management for Workflow Management Applications", *Procs. 8th Int. Conf. on Database and Expert System Applications*, Toulouse, France, 1997, pp. 430-439.

[Gre98] P. Grefen, R. Wieringa, "Subsystem Design Guidelines for Extensible General-Purpose Software", *Procs. 3rd Int. Software Architecture Workshop*, Orlando, USA, 1998, pp. 49-52.

[Gre99] P. Grefen, B. Pernici, G. Sánchez (Eds.), *Database Support for Workflow Management: The WIDE Project*, Kluwer Academic Publishers, 1999.

[Kam96] M. Kamath, K. Ramamritham, "Correctness Issues in Workflow Management", *Distributed Systems Engineering Journal*, Vol. 3, No. 4, 1996.

[Kor90] H. Korth, E. Levy, A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions", *Procs. 16th Int. Conf. on Very Large Databases*, Brisbane, Australia, 1990, pp. 95-106.

[Kry96] P. Krychniak et al., "Bounding the Effects of Compensation under Relaxed Multi-Level Serializability", *Distributed and Parallel Databases*, Vol. 4, No. 4, Kluwer Academic, 1996, pp. 355-374.

[Hsu93] M. Hsu (Ed.), *Special Issue on Workflow and Extended Transaction Systems, IEEE Data Engineering Bulletin*, June 1993.

[OM95] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Version 2.0*, Object Management Group, 1995.

[Reu95] A. Reuter, F. Schwenkreis, "ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management Systems", *IEEE Data Engineering Bulletin*, 18-1, 1995, pp. 4-10.

[Von99] J. Vonk, P. Grefen, E. Boertjes, P. Apers, "Distributed Global Transaction Support for Workflow Management Applications", to appear in: *Procs. 10th Int. Conf. on Database and Expert System Applications*, Florence, Italy, 1999.

[Wei91] G. Weikum, "Principles and Realization Strategies of Multilevel Transaction Management", *ACM Transactions on Database Systems*, Vol. 16, No. 1, 1991, pp. 132-180.