The VLDB Journal 10: 316–333 (2001) / Digital Object Identifier (DOI) 10.1007/s007780100056

Global transaction support for workflow management systems: from formal specification to practical implementation

Paul Grefen, Jochem Vonk, Peter Apers

Computer Science Department, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands;
E-mail: {grefen,vonk,apers}@cs.utwente.nl

Edited by P. Bernstein. Received: 16 November 1999 / Accepted 29 August 2001
Published online: 6 November 2001 – © Springer-Verlag 2001

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 compensation, but extended to deal with both arbitrary process structures to allow cycles in processes and safepoints to allow partial compensation of processes. We present a formal specification of the transaction model and transaction management algorithms in set and graph theory, providing clear, unambiguous transaction semantics. The specification is straightforwardly mapped to a modular architecture, the implementation of which is first applied in a testing environment, then in the prototype of a commercial workflow management system. The modular nature of the resulting system allows easy distribution using middleware technology. The path from abstract semantics specification to concrete, real-world implementation of a workflow transaction mechanism is thus covered in a complete and coherent fashion. As such, this paper provides a complete framework for the application of well-founded transactional workflows.

Keywords: Transaction Management – Long-running transaction – Compensation – Workflow management

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. [31, 2, 11, 45]). 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 academic 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. Relaxed transactional properties are required for long-living, co-operative processes like workflows to avoid complete undo of performed work and to facilitate sharing of intermediate results of processes. Compensation allows rollbacks in relaxed transactional processes. Our transaction model is based on the existing saga model [19], but is applicable to general process structures including cycles and adds the notion of partial compensation through the use of safepoints. Support for cycles is an important feature, as many practical workflow applications contain cyclical process structures, e.g., to obtain a business goal in an iterative way or to retry a specific business function. Partial compensation is important in practice to have a flexible means to control the scope of a process rollback.

In this paper, 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 a 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 modular 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-

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. A short and earlier version of this paper has appeared in the proceedings of the CoopIS'99 conference [26].

oriented applications like workflow management [8, 25]. One of the major parts of the database technology developed in WIDE is a two-level transaction management subsystem, which is informally described in [23, 25]. The upper level of the subsystem caters for global transactions as formally described in this paper. The lower level caters for individual business transactions. The transaction management subsystem has been integrated with the commercial FORO workflow management system developed by Sema Group [18]. The resulting WIDE transactional workflow management 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 the advanced transaction management mechanism implementing this model, thus giving a complete operational semantics of transaction management in the context of this model. Thirdly, we demonstrate that an implementation of this mechanism can be integrated into a modular, loosely coupled architecture of a commercial workflow management system. The resulting architecture allows flexible distribution of transaction management. Altogether, we provide a complete framework for the application of advanced, compensation-based transaction management to practical workflow management, thereby bridging the gap existing between these two domains.

1.1 Structure of this paper

The structure of this paper is as follows. In Sect. 2, we informally discuss the WIDE transaction model as the context for the formal treatment of transaction processing in the sequel of this paper. The notion of global transaction as defined in the WIDE model is central in this paper. Sect. 3 discusses the formal specification of global transactions in terms of specification graphs. In Sect. 4, we show how execution graphs are dynamically constructed from specification graphs during transaction execution. Execution graphs model the execution state of a global transaction. Section 5 presents the formal specification of algorithms for handling workflow process rollbacks. These algorithms describe the dynamic generation of compensating global transactions from execution graphs. We end this section with a short discussion of the practical applications of the formalism. One of these applications is using the specification of the algorithms in the implementation of a transaction support subsystem. The architecture that provides the overall structure for this implementation is discussed in Sect. 6. We present a general abstract architecture, a concrete architecture in the context of the FORO workflow management system, and a distributed architecture. In doing so, we show how the formal specification can be directly mapped onto the architecture, thus providing a complete path from formal semantics to practical implementation of transaction management. We provide a discussion of related work in Sect. 7, subdivided into four related areas of research. The paper is ended with conclusions and a short discussion of possible extensions to the presented work.

2 Context

In this section, we present the background of the transaction management approach elaborated in this paper. 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. We introduce an example that is used in the next sections to illustrate algorithms. A formal specification of global transaction management presenting precise semantics follows in the next sections of this paper.

2.1 Two-layer transaction model

In the WIDE transaction model [23], 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. In most applications, both types of processes exist, as low-level processes are subprocesses of high-level processes. The WIDE model has been designed to cater for process-centric applications like workflow management, where complex 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 [5]. Local transactions operate on persistent data (both workflow and application data), using traditional transaction mechanisms to enforce the ACID properties, most notably atomicity and isolation. 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) that comply with the ACID properties – as guaranteed by the local transaction layer. The two-layer approach thus provides a good basis for a separation of concerns in complex transaction management. Relaxation of transactional properties at the global transaction layer 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 model is relaxed by making intermediate results of steps visible to the context of the global transaction (i.e., local transactions commit their results to the shared database), such that they are accessible to other global transactions. In line with this shared database approach, there is no explicit data flow and consequently no data-oriented synchronization within or between workflow instances at the global transaction level. Note, however, that the local transaction layer takes care of traditional concurrency control.

To obtain relaxed atomicity, rollback operations in the global transaction layer should have application-specific semantics instead of the database-oriented semantics of the local transaction model. For these reasons, we have chosen a global transaction model that is based on the saga transaction model

[19], extended with a flexible mechanism for partial rollback. As in the saga model, 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 included 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 *safepoints*. A safepoint is a step in a workflow from where forward recovery can safely be started after a global rollback has been completed (comparable to compensation points in the OpenPM approach [14]) and hence a point where compensation can end. As such, safepoints provide ways to flexibly specify partial rollback strategies dealing with abort situations occurring in different parts of a global transaction. Unlike safepoints in the saga model [19], global transaction safepoints do not require making checkpoints. Like the functionality of compensating steps, placement of safepoints in a global transaction is fully application-dependent.

The WIDE workflow model also includes exception primitives to model non-standard behavior of applications [7, 25]. An exception is specified as an Event-Condition-Action (ECA) rule, stating when and under which conditions a separately specified subflow is triggered by a workflow process. A decoupled execution model [54] is used for exceptions in the WIDE approach, i.e., the clauses of an ECA rule are evaluated in separate transactions. Consequently, exception handling is completely orthogonal to transaction handling. For this reason, we will not discuss exceptions explicitly in this paper.

2.2 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 [7, 25]). 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 left-hand side of Fig. 1. The graph models a high-level view of 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 safepoint. This means that in case of a process rollback, the effects of 'sales' do not need to be undone and process re-execution can start from 'sales'.

Instantiations (executions) of a specification graph are described 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 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 right hand side of Fig. 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. Note that a single specification graph can lead to many different execution graphs. 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 next sections.

3 Transaction specification

This section formalizes the specification of global transactions as outlined in the previous section. Global transactions are directed graphs built from workflow process elements and sequence relations between them. These two ingredients of global transactions are discussed in the subsections below. After that, we show how they are used together in the specification of global transactions.

3.1 Workflow process elements

Workflow processes contain two types of elements: workflow tasks that represent activities to be performed and workflow connectors that provide the primitives to construct complex process structures from these workflow steps.

Workflow tasks are specifications of atomic local transactions in our model, as outlined in the previous section. A local transaction specification t_{spec} is taken from the domain T_{spec} and has as attributes a name from the domain *ID*, a task specification from the domain P (the domain of task programs), the identifier of its compensating counterpart from the domain *ID*, and an indication whether the local transaction is a safepoint:

$$T_{spec} = \langle name : ID, task : P, comp : ID, safep : bool \rangle$$

In practice, workflow specification models and languages offer different types of tasks to specify automatic tasks performed by a system, manual tasks performed by a human actor, or

[1] For reasons of clarity, we use a simplified version of the WIDE workflow process notation [7, 25], in which a diamond with a '1'-symbol indicates an or-split or or-join, and a diamond with an 'n'-symbol indicates an and-split or and-join. An 's'-symbol in a process step denotes a safepoint.

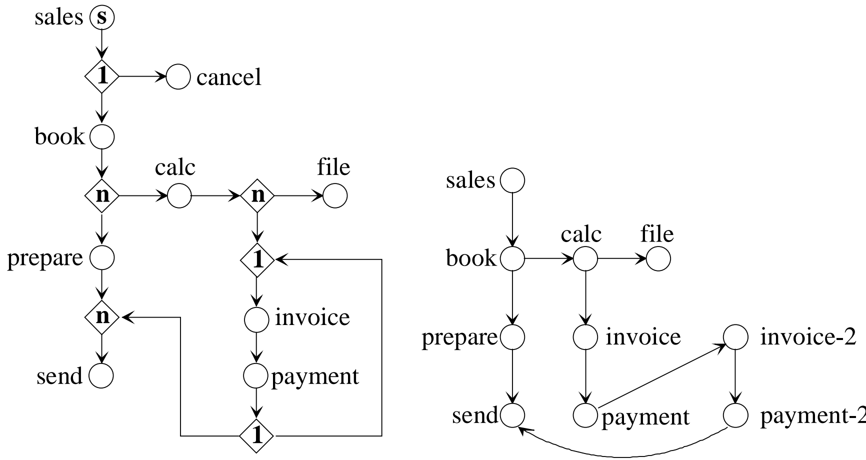

Fig. 1. Example specification and execution graphs

wait tasks to model explicit delays (wait states) in process execution. The various types of tasks included in the WIDE workflow model are described in detail in [7, 25]. In this paper, we abstract from the specifics of tasks, as we view them as atomic units of work.

Workflow connector elements represent the various types of splits and joins found in workflow process definitions. A split connector cs_{spec} is taken from the domain CS_{spec}, a join connector cj_{spec} from the domain CJ_{spec}. CS_{spec} and CJ_{spec} are both subtypes of the domain of general connectors C_{spec}:

$$C_{spec} = CS_{spec} \cup CJ_{spec}$$

Apart from its identification, the only relevant attribute of a connector is the indication whether it is total (i.e., it is an and-split or and-join as opposed to an or-split or or-join):

$$C_{spec} = \langle name : ID, total : bool \rangle$$

The selection expressions associated with an or-join are included in the specification of sequence elements, as discussed below. The WIDE workflow model offers a number of variations on the basic connector types, like iterative splits and joins, and partial (k-out-of-n) joins [7, 25]. These can, however, all be handled in a similar way as the basic types – hence they are not explicitly discussed in this paper.

Now we can define the set of workflow process node (vertex) specifications V_{spec} as the supertype of the local transaction specifications and the connector specifications:

$$V_{spec} = T_{spec} \cup C_{spec}$$

3.2 Sequence elements

Where the workflow process elements defined above represent the nodes in a workflow specification graph, the workflow sequence elements represent the edges in a specification graph. A sequence element (edge) specification e_{spec} is taken from the set E_{spec} defined as follows:

$$E_{spec} = \langle orig, dest : ID, cond : X_{sel} \rangle$$

Each sequence element has a workflow process element from the domain V_{spec} as origin and destination, identified by their identifiers from the domain *ID*. Further, a sequence element has a condition identification from the domain X_{sel} that specifies when the sequence element is active. The condition is used to select between outgoing paths from an or-join – in other cases the condition yields *TRUE* by default. The condition is evaluated by the workflow management system during workflow enactment, usually on the basis of case data associated with a global transaction instance. The details of condition evaluation are not relevant in the context of this paper.

3.3 Global transaction specification graph

Now we can define a global transaction specification graph G as a tuple consisting of a set of workflow process elements and a set of workflow sequence elements:

$$G_{spec} = \langle V_{spec}, E_{spec} \rangle$$

The example global transaction specification in the left-hand side of Fig. 1 consists of a set of 15 process elements and 16 sequence elements.

As explained above, a global transaction specification is an abstraction of a complete workflow specification, obtained by removing all elements from the workflow specification that are not relevant to transaction processing as described in this paper. Although the global transaction specification model is relatively simple, it is based on an advanced, full-fledged workflow model [7, 25]. A comparable approach to workflow abstraction is used in [44], where only nodes and edges of a process model are the basis for the specification of the transaction model.

We introduce a number of basic operations on specification graphs that we require in more complex functions. Starting points of a specification graph are nodes without incoming edges. Ending points are nodes without outgoing edges:

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

Functions *predv* and *succv* calculate the set of direct predecessors, respectively direct successors of a vertex v in a graph G:

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

We require that a correct specification graph have exactly one starting point and at least one ending point. These constraints are expressed as follows (where *card* is the set cardinality function):

$$card(start(G)) = 1$$
$$card(end(G)) \geqslant 1$$

We assume that all connectors are modeled explicitly in specification graphs, i.e., all local transactions have at most one incoming and one outgoing edge. This model can, however, be easily adapted to cater for other models, e.g., specification in UML activity diagrams [6] in which or-joins are implicit.

4 Transaction execution

This section formalizes the execution of global transactions, based on the specification defined in the previous section. Put shortly, global transaction execution takes a global transaction specification graph as input and incrementally generates a global transaction execution graph that is a representation of the execution history of the global transaction.

Important elements of this section are the definition of global transaction execution graphs, basic predicates and functions defined on these graphs, and construction functions that modify execution graphs during transaction execution. The treatment in this section is the basis for the rollback (compensation) algorithms in the next section. We start with formalizing 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 [23, 5]. As such, instantiations of local transactions form the elementary steps in instantiations of global transactions. For reasons of brevity, we use the term 'local transaction' to denote 'instantiation of a local transaction'. Local transactions are defined in the domain T_{loc}, the domain of local transaction identifiers. These identifiers are constructed by suffixing the name attribute from local transaction specifications in the global transaction specification graph with local transaction instantiation numbers. Instantiation numbers are necessary to distinguish between multiple instantiations of the same local transaction specification (as generated by cycles in a specification graph). For clarity, we omit these numbers in this paper.

We first define a function that yields the local transaction specification that a local transaction is an instantiation of:

$$spec : T_{loc} \to T_{spec}$$

A number of unary predicates is defined on the domain of local transactions. Predicates *started* and *committed* indicate the execution state of a local transaction. They denote whether the execution of a local transaction has begun, respectively, has completed:

$$started : T_{loc} \to bool$$
$$committed : T_{loc} \to bool$$

Predicates *safe*, *dummy*, and *idempotent* denote semantic properties of a local transaction – we need these predicates in the construction of compensating global transaction specification graphs. Predicate *safe* tests whether a local transaction is a safepoint. Predicate *dummy* tests whether a local transaction has dummy semantics (i.e., does not have any effect). Predicate *idempotent* tests whether a sequence of executions of a local transaction has the same semantic effect as one single execution. So we have:

$$safe : T_{loc} \to bool$$
$$dummy : T_{loc} \to bool$$
$$idempotent : T_{loc} \to bool$$

The implementation of these predicates relies on the local transaction specification an instantiation is based on:

$$safe(t) = spec(t).safep$$
$$dummy(t) = empty(spec(t).task)$$
$$idempotent(t) = idempotent(spec(t).task)$$

The idempotent predicate defined on tasks can be evaluated automatically in simple cases, but will have to be indicated by the task designer for the general case.

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

$$trig : T_{loc} \times T_{loc} \to bool$$
$$equal : T_{loc} \times T_{loc} \to bool$$

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:

$$(\forall v \in T_{loc})(committed(v) \Rightarrow started(v))$$
$$(\forall v, w \in T_{loc})(trig(v, w) \Rightarrow committed(v) \wedge started(w))$$

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:

$$comp : T_{loc} \to T_{spec}$$
$$comp(t) = \begin{cases} spec(t).comp & \text{if } spec(t).comp \neq null \\ dummy & \text{if } spec(t).comp = null \end{cases}$$

As remarked above, compensating counterparts of local transactions have to be specified by an application designer. Below, we use local transactions as components (atomic steps) in global transactions.

4.2 Execution graph definition and basic operations

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:

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

We introduce a number of basic operations on execution graphs that we require in the sequel of this paper. Like for specification graphs, starting points of an execution graph are nodes without incoming edges and ending points are nodes without outgoing edges:

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

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

$$preds(G, S) = \{ v \in G.V \,|\, \langle v, w \rangle \in G.E \wedge w \in start(S) \}$$

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:

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

As discussed in Sect. 2.1, cycles in process specification graphs are rolled out in execution graphs, so execution graphs are acyclic. This constraint on the structure of execution graphs can be expressed as shown below:

$$(\forall v \in G.V)\ (\neg path(G, v, v))$$
$$path(G, v, w) \Leftrightarrow (\langle v, w \rangle \in G.E) \vee$$
$$(\exists x \in V)\ (\langle v, x \rangle \in G.E \wedge path(G, x, w))$$

Having completed the preliminaries, we can now 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. The details of the maintenance follow from the specification of the global transaction. Basically, there are four types of operations on execution graphs corresponding with events in the lifecycle of a global transaction:

1. Creation of a new empty execution graph G_{exec} when a new global transaction instance is started on the basis of a specification graph.
2. Addition of a new vertex (and corresponding edges) to G_{exec} when a new local transaction is started as specified in G_{spec}.
3. Replacement of a vertex (and corresponding edges) in G_{exec} 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. A new local transaction is started when a new task in the specification graph is instantiated. Note that the execution of connector elements does not change the execution graph itself, but can influence the execution path through the specification graph (based on the conditions in sequence elements).

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

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

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]:

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

Figure 2 shows two partial execution graphs, resulting from the execution of the specification graph in Fig. 1. In the left-hand side graph, three steps have been completed (indicated by 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. A global transaction is ended after the last task conforming to the global transaction specification graph has been ended. 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.

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

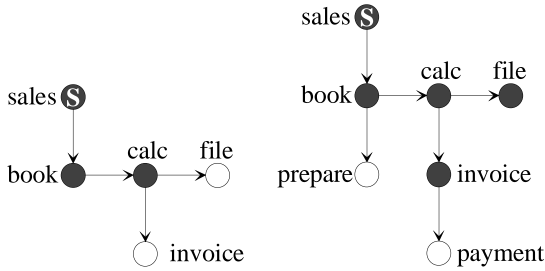

Fig. 2. Partial execution graphs

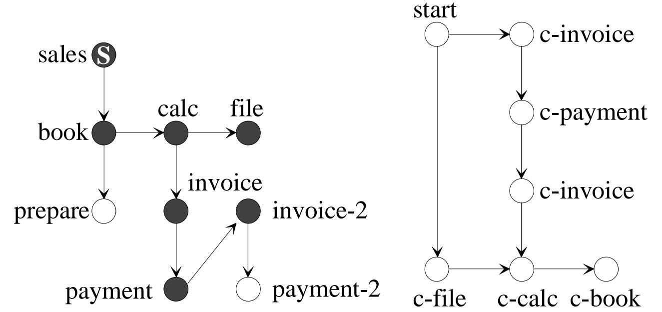

Fig. 3. Partial execution and compensation graphs

5 Global transaction compensation

In this section, we present the algorithms used for compensating global transaction when a global abort situation arises. Put shortly, compensation is performed by dynamically generating and executing a compensating specification graph, based on the analysis of the execution graph of a global transaction.

We start this section 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 left part of Fig. 3. Here, we see an execution graph corresponding to the specification graph in Fig. 1, at a point where the global transaction has partly been completed. The grayed steps have been committed; two steps are being executed. Local transaction 'sales' has been specified to be a safepoint. 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 safepoint 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 right-hand side of Fig. 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 section. 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 specification and the list of restart points in the original global transaction execution graph. 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 specification 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:

$$
\begin{aligned}
&abort(m, n, v) \\
&\quad = storespec(gcomp(m, getexec(n), v), newid(n))
\end{aligned}
$$

Function *getexec* retrieves the execution graph from persistent storage based on the global transaction identifier n and returns it as its result. Function *storespec* takes a compensation graph plus identifier generated by *newid* as input and stores the graph as specification graph in persistent storage; it produces no result.

Function *gcomp* is used to calculate the compensation specification graph for a given abort mode, execution graph, and failure point. It distinguishes between complete compensation and partial compensation. In case of a partial compensation, the restart points in the execution graph have to be calculated. Hence, the result type of *gcomp* is a pair of compensation graph and set of local transactions.

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

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 specification has to be constructed. This compensating global transaction is based on the execution graph of the global transaction that has to be rolled back. 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 the following five steps:

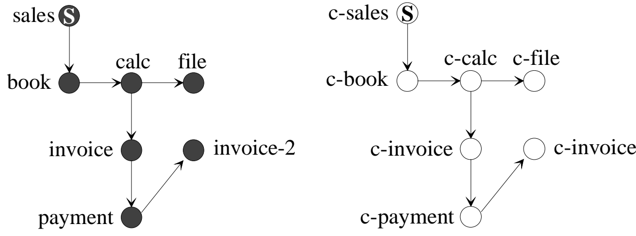

Fig. 4. Steps 1 and 2 in complete compensation graph construction

1. The active local 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. Note that this step effectively transforms an execution graph into an 'intermediate form' specification graph. This form is independent from the concrete process specification model (the latter is dealt with in step 5 below).
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 and connected to the 'original' starting points.
5. Explicit connectors are added to the specification graph to make it compliant with the concrete process specification model defined in Sect. 3.

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

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

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

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

The stripped version of the execution graph from Fig. 3 is shown in the left-hand side of Fig. 4.

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

$$compv(G) = \langle exchangev(G.V), exchangee(G.E) \rangle$$
$$exchangev(V) = \{v \in T_{loc} \,|v = comp(w) \wedge w \in V\}$$
$$exchangee(E) = \{\langle v, w \rangle \in T_{loc} \times T_{loc} | \langle x, y \rangle \in E \wedge \\ v = comp(x) \wedge w = comp(y)\}$$

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

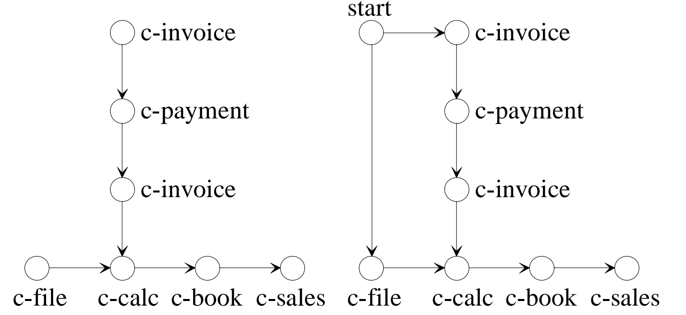

Fig. 5. Steps 3 and 4 in complete compensation graph construction

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

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

The effect of applying edge inversion to the graph in Fig. 4 is depicted in the left-hand side of Fig. 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 (i.e., the compensating tasks) and flow control, but lacks a unique starting point.

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

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

The graph in the left-hand side of Fig. 5 contains two starting points ('c-file' and 'c-invoice'). Therefore, a single starting point is added as shown in the right hand side of the figure. The graph represents the complete compensating global transaction, but without explicit connectors.

Inserting explicit connectors. The intermediate form specification graph obtained through the previous steps does not contain any explicit connectors yet. And-split and and-join connectors are implicit in nodes that have multiple outgoing, respectively, multiple incoming edges (note that a compensating process does neither contain or-splits nor or-joins):

$$multiout(G) = \{v \in G.V \,|card(succv(v)) > 1\}$$
$$multiin(G) = \{v \in G.V \,|card(predv(v)) > 1\}$$

Function *insconn* computes a new specification graph in which the set of nodes is extended with the set of required explicit

connectors and the set of edges is computed by leaving out the edges involved in implicit connectors and adding the edges required for the explicit connectors:

$$insconn(G) = \langle G.V \cup newconn(G), (G.E \backslash deledges(G)) \cup newedges(G) \rangle$$

The set of required explicit connectors is easily established as follows (note that we give the explicit connectors the same ID as the nodes involved in the implicit splits and joins to 'connect' them easily):

$$newconn(G) = newsplit(G) \cup newjoin(G)$$
$$newsplit(G) = \{\langle n, \mathit{TRUE} \rangle \in CS_{spec} | (\exists v \in multiout(G))(v.name = n)\}$$
$$newjoin(G) = \{\langle n, \mathit{TRUE} \rangle \in CJ_{spec} | (\exists v \in multiin(G))(v.name = n)\}$$

The superfluous edges in the new graph are also easily determined:

$$deledges(G) = \{\langle v, w \rangle \in G.E | v \in multiout(G) \vee w \in multiin(G)\}$$

The new edges are determined per explicit connector in the following way:

$$newedges(G) = \bigcup_{v \in multiout(G)} newedgeslocs(G, v) \cup \bigcup_{v \in multiin(G)} newedgeslocj(G, v)$$

$$newedgeslocs(G, v) = \{\langle v, w \rangle | w \in newsplit(G) \wedge v.name = w.name\} \cup \{\langle x, w \rangle | x \in newsplit(G) \wedge v.name = x.name \wedge w \in succv(v)\}$$
$$newedgeslocj(G, v) = \{\langle v, w \rangle | v \in newjoin(G) \wedge v.name = w.name\} \cup \{\langle x, w \rangle | w \in newjoin(G) \wedge v.name = w.name \wedge x \in predv(v)\}$$

The application of the above function of the intermediate compensation graph depicted in the right hand side of Fig. 5 results in the completed compensation graph depicted in Fig. 6.

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 safepoints in the graph. A simple example has already been presented in Fig. 3, where task 'sales' of the execution graph is not compensated in the compensation graph because it is a safepoint.

As execution graphs can be arbitrarily complex, the situation is usually not as simple as depicted in Fig. 3. The problem is finding the proper subgraph of the execution graph to be compensated, taking into account safepoints and forward

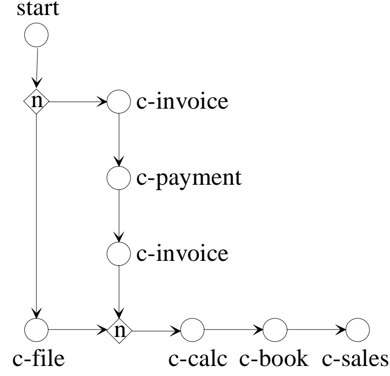

Fig. 6. Step 5 in complete compensation graph construction

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 of the execution graph to be compensated and next using the complete compensation algorithm of Sect. 5.3. The direct predecessors of the subgraph to be compensated become restart points. Function *pcomp* is thus specified as follows:

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

Function *extend* is used to calculate the subgraph starting from the local transaction that caused the abort; it is specified below.

Calculating a subgraph to be compensated. The subgraph to be compensated is calculated from the vertex where the partial abort originated. From this vertex, we first construct a subgraph consisting of predecessors of the vertex until safepoints 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 by function *extback* in a recursive fashion until the subgraph has reached a stable size, i.e., does not grow anymore. Function *extback* uses function *backstep* to extend a graph one step:

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

Extending a subgraph forward is performed in a similar manner: function *extforw* extends a subgraph in a recursive fashion until the subgraph has reached a stable size:

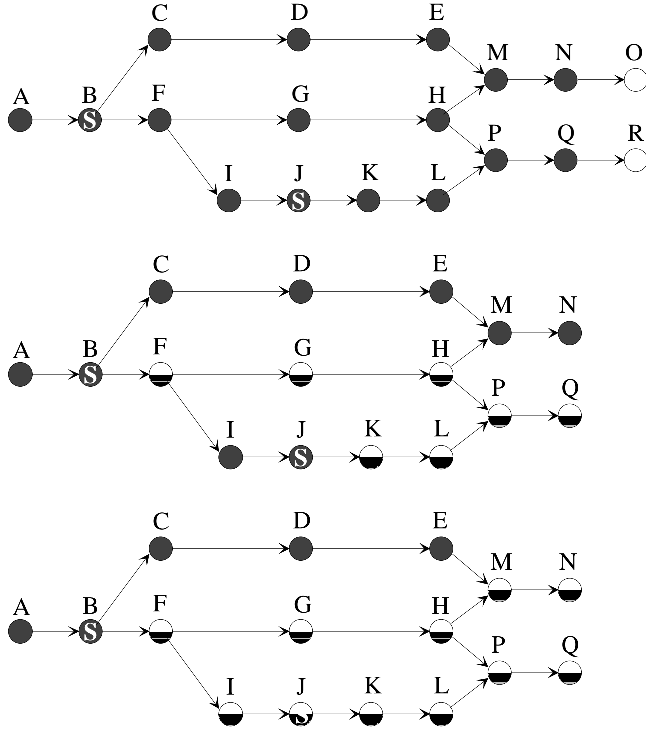

Fig. 7. Execution graph, backward extension, and forward extension

$$
\begin{aligned}
&extforw(S,G)\\
&= \begin{cases} S & \text{if } S = forwstep(S,G) \\ extforw(forwstep(S,G),G) & \text{otherwise} \end{cases}\\
&forwstep(S,G) = \langle forwstepv(S,G), forwstepe(S,G)\rangle\\
&forwstepv(S,G)\\
&= \{v \in G.V \,|\, v \in S.V \vee (\langle w,v\rangle \in G.E \wedge w \in S.V)\}\\
&forwstepe(S,G) = \{\langle v,w\rangle \in G.E \,|\, v \in S.V\}
\end{aligned}
$$

Figure 7 shows an example of subgraph calculation. In the top of the figure, an execution graph is depicted. Steps B and J are safepoints (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 rollback operation. Then first, running steps O and R are aborted using the atomicity control functionality of the underlying local transaction mechanism. Next, compensation processing is initiated by determining the subgraph to be compensated. Backward extension as described above takes place from step Q (being the direct predecessor of step R that caused the rollback), 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 safepoints 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 safepoint J, as this point is covered by forward extension. Figure 8 shows the final compensation graph, obtained by applying the algorithms of Sect. 5.3 to the calculated subgraph. In this figure, a step X^{-1} denotes the compensating counterpart of step X.

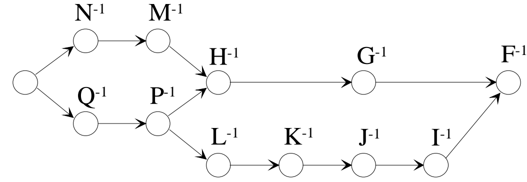

Fig. 8. Compensation graph corresponding with Fig. 7

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. 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), or filtering of not strictly necessary steps based on actual system load.

A second-order function *filter* is used to construct a functional composition of various functions that each perform one of the filtering algorithms, e.g., the ones mentioned above. This function takes a specification graph and a list of filter functions as its arguments:

$$
\begin{aligned}
&filter(G, < filter_1, \ldots, filter_n >)\\
&\quad = (filter_1 \circ \cdots \circ filter_n)(G)
\end{aligned}
$$

Function *filter* thus provides the function of a filter driver, allowing easy addition of new filtering functionality or selection of filters for specific application classes.

Function *filter* can easily be applied in the compensation driver discussed in Section 5.2, resulting in the following specification of function *abort* (where f is the list of filter functions):

$$
\begin{aligned}
&abort(m,n,v)\\
&\quad = storespec(addstart(filter(gcomp(m,\\
&\qquad getexec(n),v),f)),newid(n))
\end{aligned}
$$

Note that we have to reapply function *addstart* (as defined in Section 5.3) above, as the filtering might remove the starting point of a compensation graph. It is now used twice to keep a clear separation of concerns between compensation graph generation and compensation graph filtering.

Below, we show how filtering dummy steps and idempotent steps can be specified as two filter functions using the formalism introduced in this paper. Before, we first introduce a general filtering function.

Filtering steps. Function *filterf* specified below is used to remove steps from a compensation graph that satisfy a predicate f, where f describes a characteristic of a step in the context of its compensation graph. Removing these steps implies removing the corresponding vertices from the graph, removing all edges connected to these vertices, and inserting new edges to connect the vertices that were disconnected by the removals:

$$
\begin{aligned}
filterf(G,f) = \langle & filterfv(G,f), filterfe(G,f)\\
& \cup newfe(G,f)\rangle
\end{aligned}
$$

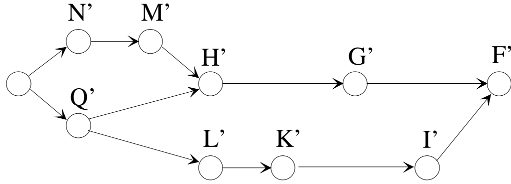

Fig. 9. Reduced compensation graph of Fig. 8

$$
\begin{aligned}
filterfv(G,f) &= \{v \in G.V \,|\, \neg f(G,v)\} \\
filterfe(G,f) &= \{\langle v,w\rangle \in G.E \,|\, \neg f(G,v) \wedge \neg f(G,w)\} \\
newfe(G,f) &= \{\langle v,w\rangle \in G.V \times G.V | \\
&\qquad fconn(G,f,v,w)\} \\
fconn(G,f,v,w) &= (\exists x \in succv(v))(f(G,x) \wedge \\
&\qquad (\langle x,w\rangle \in G.E \vee fconn(G,f,x,w)))
\end{aligned}
$$

Removing dummy steps. 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. An inverse transaction may not have been specified, because undoing the transaction has no added value in a workflow process. Transactions for which an inverse does not exist should be handled with great care. For all local transactions that cause a relevant state change in a business process, inverse transaction should normally be specified. The inverse may have a quite different implementation than the 'original' – for example, the inverse of giving out cash at an ATM may be sending an invoice to the customer.

In constructing a compensating graph as discussed above, transactions without compensating counterpart 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 quite easily specified in function *filterd* using function *filterf* introduced above (note that predicate *dummy* is context-free and thus does not require the compensation graph as an argument):

$$
\begin{aligned}
filterd(G) &= filterf(G, dummyf) \\
dummyf(G,v) &= dummy(v)
\end{aligned}
$$

We base an example on the compensation graph of Fig. 8. Assume that steps P and J do not have compensating counterparts, i.e., P^{-1} and J^{-1} are empty actions. Then these empty actions can be removed from the graph, resulting in the compensation graph shown in Fig. 9.

Removing idempotent steps. An idempotent compensation step is a step that produces the same effect no matter how many times it is executed in sequence. If we have an application step that modifies an application variable, a compensation step could set this variable to a default value. Clearly, the compensation step is idempotent: if in the workflow the application step is executed several times in sequence, the compensation needs to be executed only once. For this purpose, we introduce function *filteri* that removes unnecessary idempotent steps from a compensation graph. This predicate

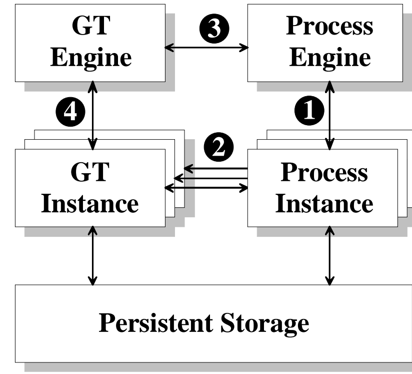

Fig. 10. Abstract GTS architecture

idemf identifies idempotent steps that have only direct predecessors with equal semantics (assuming the basic predicate *idempotent* defined on local transactions).

$$
\begin{aligned}
filteri(G,v) &= filterf(G, idemf) \\
idemf(G,v) &= idempotent(v) \wedge (\forall w \\
&\qquad \in predv(G,v))(equal(v,w))
\end{aligned}
$$

5.6 Application of the formalism

In the current and the previous sections, we have described a complete, formal specification of algorithms for global transaction management as informally introduced in Sect. 2.2. This formal specification can be practically used in three ways:

- Firstly, the specifications provide a complete and unambiguous functional specification for the design of a transaction management subsystem. The high-level functions completely specify the interfaces to the subsystem and the lower-level functions completely specify the internals.
- Secondly, the specifications provide the basis for formal analysis of transactional behavior of workflows during workflow design. Given the unambiguous nature of the algorithms, effects of transactional primitives like abort can be statically analyzed in detail.
- Thirdly, the algorithms can be used as the basis for a simple tool that provides on-the-fly insight in the effect of global aborts during the execution of a workflow. Using partial abort is practical in real-world business processes, but end users invoking an abort should have the means to check what part of a complex process will be affected by an abort.

In the next section, we move from formal algorithm specification to the design of software architectures implementing the algorithms. In doing so, we focus on the first above use of the formal specifications. More concretely, we show in Sect. 6.1 in detail how the formal functions can be mapped onto software components.

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

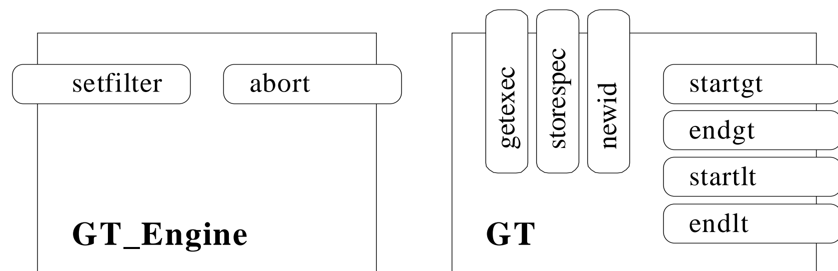

Fig. 11. GT_Engine and GT class interfaces

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 concrete implementation of the GTS is discussed, which is applied in a stand-alone test environment. The integration of the GTS in the FORO workflow management system is discussed, as realized in the WIDE project. FORO is a commercial WFMS [18] marketed by Sema Group. Finally, we show how the architecture can be extended to deal with distributed global transactions, i.e., global transactions that span multiple process engines.

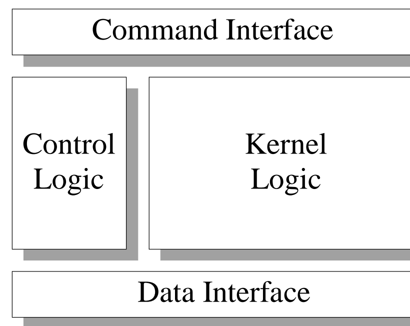

Fig. 12. Internal GT engine architecture

6.1 Abstract architecture

The abstract architecture of the GTS and its environment are depicted in Fig. 10. The left-hand 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 the 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 workflow process specification (i.e., a global transaction specification graph as defined in Sect. 3) 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 Fig. 10). 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 as defined in Sect. 4. Like process instance objects, GT instance objects are created and deleted dynamically. GT objects implement the functions related to execution graph maintenance defined in Sect. 4.3.

Process instance and GT instance objects are coupled one-to-one, as a process instance corresponds to 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 to update the state of the global transaction.

When the process engine signals a global abort for a process instance, the GT engine is informed about this through interface ❸. This corresponds to invoking function *abort* as specified in Sect. 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. These steps correspond with the function calls specified in the compensation driver (function *abort* as defined in Sect. 5.2). The process engine executes the compensating transaction and then restarts the original global transaction at the indicated restart points.

The high-level functions for global transaction management that we have introduced in Sections 4 and 5 of this paper can be mapped directly to the abstract software architecture shown in Fig. 10. We have depicted this mapping in Fig. 11. On the left, we see the interfaces of the GT_Engine object class, of which a GT engine is instantiated. From Sect. 5, we only have the interface to function *abort*. We have added an interface to a function *setfilter* here, to illustrate how the filter behavior of the GT engine can be influenced. This function sets a complex state variable in the GT engine the value of which is used as an argument for function *filter* as introduced in Sect. 5.5. On the right side of the figure, we see the interfaces of the GT object class, of which GT instances are created. Horizontally, we see the interfaces used by the client system to the functions defined in Sect. 4 (through interface ❷ in Fig. 10). Vertically, we see the interfaces to the functions required by the GT engine as described in Sect. 5.2. The interfaces of the classes correspond to methods in the object-oriented paradigm.

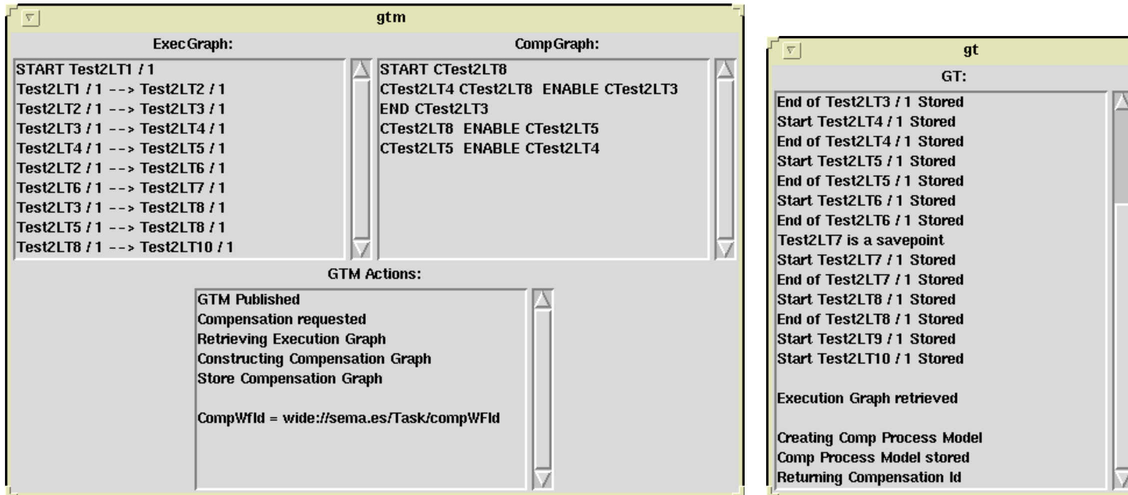

Fig. 13. GTS front end

Note that the efficiency of the algorithms of the GT engine (the performance of the module) is not too important, since 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 GTS software architecture

A GTS implementation has been realized in the WIDE project. Below, we first discuss the internal architecture of the GT engine. Then we show how the implemented GTS has been used in a stand-alone test environment. In Sect. 6.3, we will turn to the integration in a WFMS architecture.

Internal GT engine architecture. The internal architecture of the GT engine is based on the standard software module architecture chosen in the WIDE project, as shown in Fig. 12.

The GT engine communicates with its context through two interfaces: the command interface to a process engine and the data interface to GT instances. The command interface covers the 'transaction semantics server' functionality discussed above. The data interface covers the connections to services that the GT engine relies on. Both interfaces use a CORBA mechanism [46] for communication with the environment (we will address this further when discussing the integrated architecture below).

The internal logic of the GT engine is separated into control logic and kernel logic. The former controls the internal operation of the engine and drives the interface logic. The latter contains the algorithms for complete and partial compensation as described in this paper. In the current implementation, the GT engine is single-threaded, i.e., it can handle one abort request at a time. A multi-threaded version would easily be feasible, however: this would mainly require changes to the control logic and would leave the internals of the kernel logic unaffected.

In the previous subsection, we have seen that the GTS module mainly implements function *abort*. To show how the internal architecture of the GTS maps to the functional specification in the previous chapter, we recall the compensation driver function from Sect. 5.2:

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

Function *abort* is implemented in the command interface as shown in Fig. 12. The command interface will perform some input checking and delegate the actual execution of the function to the control logic. The control logic actually implements the compensation driver, as specified in the function above. From this function, the functionality of *gcomp* is allocated entirely to the kernel logic. The kernel logic is stateless between invocations of *gcomp*. The data logic interfaces to the *getexec*, *newid*, and *storespec* functions, which are allocated to the GT instance objects, as shown in Fig. 11.

Test environment. Because of the modular architecture depicted in Fig. 10, the implementation of the GTS could easily be tested in a stand-alone environment with mock-up process engine and process instances. This stand-alone environment allows software testing in completely controlled conditions. It also allows to easily feed the GTS with specific rollback situations to test its compliance with the algorithms outlined in the previous sections of this paper.

To provide an easy-to-use user interface to the test environment, a graphical front-end has been constructed using the TCL/TK toolkit [47]. This front-end is shown in Fig. 13. The first window displays the operations on the execution and compensation graphs and the communication actions of the GT engine (in the window called 'GTM' for Global Transaction Manager). The second window displays the communication actions of a GT instance (in the window called 'GT'). Using the output in these windows, the dynamic behavior of the GTS can be monitored.

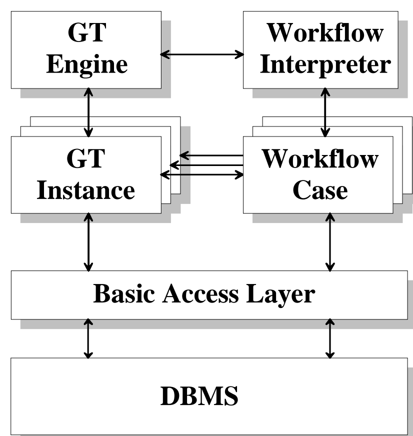

Fig. 14. FORO GTS architecture

6.3 Integrated WFMS architecture

In the context of the WIDE project, the implementation of the GTS is used in a prototype of the next generation of the FORO workflow management system [8, 25]. FORO has been equipped with both layers of the WIDE transaction support (as discussed in Sect. 2.1) 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 module in the FORO context is shown in Fig. 14. This architecture is directly based on the abstract architecture in Fig. 10. 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 [46] that allows for flexible distribution in the architecture [24]. Both GT engine and GT instances 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 hides DBMS-specific details, such that easy portability between DBMS platforms is achieved.

The modular approach to transaction management with simple, high-level interfaces and well-defined semantics allows for 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.

6.4 Distributed GT architecture

The GTS architecture can be easily extended to deal with global transactions that span multiple process engines. The need for this functionality arises if multiple process engines execute one overall process that requires transactional behavior. This situation occurs for instance if multiple organizations or independent parts of organizations enact processes that cross their boundaries.

In the distributed GT architecture, each GTM computes compensation graphs for its part of the execution graph, using the algorithms specified in this paper. Control over the overall compensation process is achieved through communication protocols between the GTMs. To facilitate this communication, a second peer-to-peer command interface can be added to the internal GT engine architecture as depicted in Fig. 12. The overall architecture of the distributed GTS is depicted in Fig. 15. In this figure, ❺ is the peer-to-peer interface between process engines used to coordinate distributed workflows. Interface ❻ is the interface between GT engines used to coordinate distributed rollbacks. Further details on the architecture and communication protocols can be found in [51].

A variation on the distributed GT architecture is used in the CrossFlow project. In this project, an architecture is designed to deal with cross-organizational workflows in dynamic virtual enterprise environments [27, 28]. In this context, distributed global transactions exist that span the workflows of two organizations that have dynamically integrated their workflow processing [52]. To support these transactions, GT engines located in the workflow environments of two separate, autonomous organizations have to cooperate.

7 Related work

In this section, we provide a discussion of work related to this paper. We have divided the related work into four topic areas: general work on advanced transaction models, specific work on approaches to compensation in transaction management, work on transactional workflows (or workflow transactions), and formal approaches to transaction semantics. Note that some of the work discussed below can be categorized in multiple areas – in these cases we have chosen the area that we considered most appropriate in painting a complete overview of the field of work related to this paper.

7.1 Advanced transaction models and environments

Advanced (or extended) transaction models have been given considerable attention in the past decade; see for example [17, 34] for overviews. Typical examples of advanced transaction models for long-running processes are nested transactions [15, 16], multi-level transactions [53], sagas [19], and nested sagas [20, 21]. General frameworks have been constructed, like ACTA [10], 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. The best-known example in this category is probably the ConTracts approach [48, 49]. ConTracts are not an extended transaction model, but an environment that provides the basis for reliable execution of long-lived computations. As such, the Contracts approach have been used for the realization of transactional workflows.

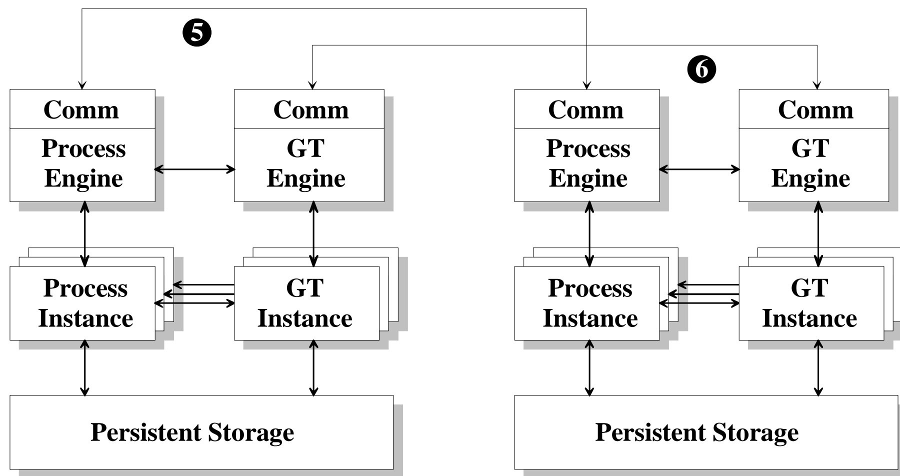

Fig. 15. Distributed GT architecture

In the WIDE project, an orthogonal two-level transaction model is used to effectively model both long-running processes and relatively short-running subprocesses [23]. In this paper, we focus on the semantics of and support for 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 [19]. Our approach to compensation is more comprehensive, however, in a number of ways. The separation of specification and execution graphs provides a natural way to handle cycles in processes. The concept of safepoint provides a flexible notion of partial compensation. The optimizations discussed in Sect. 5.5 provide possibilities to reduce the cost of performing compensations. The WIDE model is based on a single-level process model, so it does not cover nested sagas [20, 21], but can easily be extended in this direction.

A hybrid transaction model is also discussed in [9], 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 [9] and that in this paper, is that we aim at a formal specification of the semantics of transaction mechanisms, instead of using text and pseudocode descriptions.

7.2 Compensation approaches

Compensation is nowadays generally considered a proper way to handle application rollback, not only in workflow contexts. The position of [4], for example, is that ‘every well-designed TP application should include a compensating transaction type for every type of transaction’. Compensation is not only used for ‘direct’ rollbacks, but also as an ingredient to higher-level notions. In the OPERA system, for example, compensation is used in a flexible exception handling approach [30].

An advanced transaction compensation mechanism is discussed in [41] 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 [41] in the fact that we provide a complete formal specification of compensating transaction management mechanisms, whereas most other work relies on informal descriptions.

Compensation is an important ingredient of the ConTracts approach [48, 49]. The main difference with our work is the fact that ConTracts rely on the specification of compensating blocks to compensate groups of steps [49]. In our approach, compensation graphs are dynamically constructed from compensating steps for this purpose. Given the semantic structure of the ConTracts model, partial compensation has to be ‘applied with great care’ [49]. In the WIDE model, partial compensation is usually the default approach. This is possible because the consistency of the WIDE model is not as strictly defined as those of the ConTracts model – it relies more on application semantics.

The Coyote approach [12] also relies on compensation to provide rollback functionality. In Coyote, compensation is used in a conversational model for long-running transactions. The work in the Coyote project is focused on application style and system architecture issues – a formal background on the compensation approach is not given.

7.3 Transactional workflows

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 early proposals is discussed in [31] and a more recent overview is given in [11]. A characterization of transactions in workflow contexts is given in [2], stressing that advanced transaction management is indeed required, but not yet offered by existing systems. The importance of transactional aspects of workflows in production contexts is stressed in [45].

Work that focuses on high-level transaction management for workflow environments has been performed in the Exotica

project [1]. Like the global transactions discussed in this paper, the Exotica approach uses compensation to perform rollback operations, as originally described by the saga model [19]. The compensation mechanisms in Exotica are of a static nature, however, and lack a formal specification as given in this paper.

In the FlowBack project [37], an approach similar to the Exotica approach has been followed. Both in Exotica and FlowBack, compensation plans are generated based on the workflow specification. In the approach presented in this paper, the actual workflow execution is the basis for compensation plan generation.

An approach to compensation in the context of IBM's FlowMark WFMS [43] is discussed in [44]. Where our approach to partial rollback is based on the notion of safepoints in the workflow specification, the approach in [44] is based on the notion of 'spheres of joint compensation'. As such, in our approach process annotations are made that impact a complete workflow specification, whereas in the FlowMark approach annotations are made that impact parts of workflow specifications. The work in [44] presents an approach of how to integrate compensation into FlowMark. Complete specification of algorithms is not provided. At the time of writing this paper, compensation functionality has not been realized in FlowMark or its successor MQSeries Workflow [33]. A prototype extension to provide basic support for compensation functionality has been realized in the CrossFlow project, however [28, 52].

Compensation is an ingredient of the METEOR approach [39]. The work includes specification of task compensation in the WFSL workflow specification language and the enactment of compensation. Complete algorithms for specification and a formal semantics are not given, however. Handling of compensation is also considered in the OpenPM project at Hewlett-Packard [14], but detailed algorithms or a formal background are not given in this work. Their notion of 'compensation points' is comparable to our notion of safepoints. The semantics of partial compensation are, however, specified very informally in [14] – we provide a precise, unambiguous specification based on safepoints. The work on the Virtual Transaction (VT) model at Hewlett-Packard [40] continues the work in the context of OpenPM. A compensation model is used of which 'compensation end points' are comparable to our safepoints and of which a 'compensation strategy' distinguishes between 'all' and 'reachable' compensation modes – comparable to our complete and partial compensation modes. The forward rollback mechanism used after compensation provides three alternative strategies (redo, alternate path, and terminate) that are not included in the mechanism presented in this paper but that can be added easily to our approach. Semantics of the VT model are specified only informally, however, and mapping to supporting systems is not dealt with in detail in [40].

The ObjectFlow approach to rollback on the business process level in workflows relies on restoring process states [32]. This approach relies on persistence of data and can be considered orthogonal to the compensation approach: the former approach relies on reinstalling a previous state whereas the latter approach tries to recompute a state equivalent to a previous state. Equivalence in this context is an application-dependent concept. In [32], extension of the ObjectFlow approach with limited compensation functionality is considered, however.

The TSME approach aims at supporting transactional workflows by providing a programmable transaction management environment for workflow management [22]. As such, the aim of TSME is comparable to that of the ConTracts approach mentioned above. The dependency descriptors of TSME can be compared to those of ACTA – details are included in [22].

A flexible approach for compensating workflows, called opportunistic compensation, has been developed in the context of the CREW project at the University of Massachusetts [36]. In this approach, execution and compensation dependencies between workflow steps are explicitly specified in the LAWS workflow specification language. Although the explicit specification of dependencies provides more flexibility than our approach (in which compensation dependencies follow directly from control flow in a process), it can also lead to very complex situations with unclear semantics, e.g., in the case of parallel compensation and (re)execution. The semantics of the LAWS primitives are, however, only addressed informally in [36].

7.4 Formal specification of transaction semantics and mechanisms

Formal specification of transaction semantics has been addressed by a number of researchers. In [38], 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 [38]. 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 [23].

A formal approach to complex transactions in composite systems is presented in [3]. This work describes the semantics of complex transaction environments in terms of three basic system compositions (stacks, forks, and joins). Sagas are placed in this framework by analyzing their concurrency characteristics on a high abstraction level. The work is not aimed at analyzing the operational semantics of sagas, however. As such, [3] can be considered a high-level framework, into which the more detailed work in this paper can be placed. Related work is presented in [50], in which formal correctness criteria for concurrency control and recovery in transactional process management are given for contexts with transactional subsystems.

More general observations with respect to correctness issues in workflow management are presented in [35] in an informal fashion.

8 Conclusions

In this paper, we present the formal specification of a complete 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 with relaxed transactional requirements. The mechanism can easily be coupled to a low-level transaction mechanism to obtain two-level recovery functionality. In the WIDE project, this coupling has been demonstrated with a low-level nested transaction model.

The distinction between specification and execution graphs in our approach allows effective handling of cyclical process structures. The abstraction of workflow processes on the one hand and separation of workflow definition time and enactment time formalisms on the other hand also provides an independence of specific workflow specification languages. 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. We believe that the integration of a formal specification and operational approach provides a step towards the practical use of advanced transaction mechanisms.

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 useful for the developers of a transaction management subsystem: essentially, it provides a complete and unambiguous functional specification of the system. If 'wrapped' in the right tools, the formal semantics can be of use for advanced users of the underlying transaction model – both application designers and advanced end users. The formal semantics can be the basis for static formal validation of transactional behavior of complex workflow processes. The compensation algorithms can be used in workflow analysis tools to present the designer or end user with automated aids in analyzing the effects of specific global transaction designs (e.g., the scope of partial 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 prototype system has been employed in two demonstrator environments, one in the insurance and one in the medical domain [25].

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 safepoints, i.e., steps that are dynamically assigned the safepoint 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 outlined in this paper and described in more detail in [51].

An application of the work presented in this paper to heterogeneous federated environments is one of the topics of the CrossFlow project [27, 28, 29]. In this project, compensation algorithms are applied in a three-level process model supporting cross-organizational workflows [52]. Heterogeneous federated environments are typically found in electronic commerce scenarios, where tightly coupled transactional interaction is often not appropriate [13].

Acknowledgements. Thanks go to Stefano Ceri of the Politecnico di Milano for his comments on an earlier version of this work. Wijnand Derks of KPN Research is acknowledged for his feedback on parts 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

1. G. Alonso et al. (1996) Advanced Transaction Models in Workflow Contexts; Procs. Int. Conf. on Data Engineering; New Orleans, Louisiana, USA, pp. 574–581
2. G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan (1997) Functionality and Limitations of Current Workflow Management Systems; IEEE Expert; Vol. 12, No. 5
3. G. Alonso, A. Fessler, G. Pardon, H-J. Schek (1999) Transactions in Stack, Fork, and Join Composite Systems; Procs. 7th Int. Conf. on Database Theory; Jerusalem, Israel, pp. 150–168
4. P. Bernstein, E. Newcomer (1997) Principles of Transaction Processing; Morgan Kaufmann
5. E. Boertjes, P. Grefen, J. Vonk, P. Apers (1998) An Architecture for Nested Transaction Support on Standard Database Systems; Procs. 9th Int. Conf. on Database and Expert System Applications; Vienna, Austria, pp. 448–459
6. G. Booch, J. Rumbaugh, I. Jacobson (1999) The Unified Modeling Language User Guide; Addison-Wesley
7. F. Casati et al. (1996) WIDE: Workflow Model and Architecture; CTIT Technical Report 96-19; University of Twente
8. S. Ceri, P. Grefen, G. Sánchez (1997) WIDE – A Distributed Architecture for Workflow Management; Procs. 7th Int. Workshop on Research Issues in Data Engineering; Birmingham, UK, pp. 76–79
9. Q. Chen, U. Dayal (1997) Failure Handling for Transaction Hierarches; Procs. 13th Int. Conf. on Data Engineering; Birmingham, UK, pp. 245–254
10. P.K. Chrysanthis, K. Ramamritham (1994) Synthesis of Extended Transaction Models using ACTA; ACM Transactions on Database Systems, 19–3, pp. 450–491
11. A. Cichocki, A. Helal, M. Rusinckiewicz, D. Woelk (1998) Workflow and Process Automation: Concepts and Technology; Kluwer Academic Publishers
12. A. Dan, F. Parr (1997) The Coyote Approach for Network Centric Service Applications: Conversational Service Transactions, a Monitor and an Application Style; Procs. High Performance Transaction Processing Workshop; Asilomar, CA
13. A. Dan c.s. (2000) Business to Business Integration with TpaML and a B2B Protocol Framework (BPF); IBM Research Report RC 21863; IBM Research, USA
14. J. Davis, W. Du, M. Shan (1995) OpenPM: an Enterprise Process Management System; IEEE Data Engineering Bulletin, Vol. 18, No. 1, pp. 27–32
15. U. Dayal, M. Hsu, R. Ladin (1990) Organizing Long-Running Activities with Triggers and Transactions; Procs. 1990 ACM SIGMOD Int. Conf. on Management of Data; Atlantic City, USA, pp. 204–214
16. U. Dayal, M. Hsu, R. Ladin (1991) A Transactional Model for Long-Running Activities; Procs. 17th Int. Conf. on Very Large Databases; Barcelona, Spain, pp. 113–122

17. A.K. Elmagarmid (Ed.) (1992) Database Transaction Models for Advanced Applications; Morgan Kaufmann; USA
18. FORO Web Site (2001) http://dis.sema.es/projects/FORO/foro.html;Sema Group,Spain
19. H. Garcia-Molina, K. Salem (1987) Sagas; Procs. 1987 ACM SIGMOD Int. Conf. on Management of Data; San Francisco, California, USA, pp. 249–259
20. H. Garcia-Molina et al. (1991) Modeling Long-Running Activities as Nested Sagas; IEEE Data Engineering Bulletin, Vol. 14, No. 1, pp. 14–18
21. H. Garcia-Molina et al. (1998) Coordinating Multitransaction Activities with Nested Sagas; in [42]; Ch. 16, pp. 466–481
22. D. Georgakopoulos, M. Hornick, F. Manola (1996) Customizing Transaction Models and Mechanisms in a Programmable Environment Supporting Reliable Workflow Automation; IEEE Trans. on Knowledge and Data Engineering, Vol. 8, No. 4, pp. 630–649
23. P. Grefen, J. Vonk, E. Boertjes, P. Apers (1997) Two-Layer Transaction Management for Workflow Management Applications;Procs. 8th Int. Conf. on Database and Expert System Applications; Toulouse, France, pp. 430–439
24. P. Grefen, R. Wieringa (1998) Subsystem Design Guidelines for Extensible General-Purpose Software; Procs. 3rd Int. Software Architecture Workshop; Orlando, USA, pp. 49–52
25. P. Grefen, B. Pernici, G. Sánchez (Eds.) (1999) Database Support for Workflow Management: The WIDE Project; Kluwer Academic Publishers
26. P. Grefen, J. Vonk, E. Boertjes, P. Apers (1999) Semantics and Architecture of Global Transaction Support in Workflow Environments; Procs. 4th IFCIS Int. Conf. On Cooperative Information Systems; Edinburgh, Scotland, pp. 348–359
27. P. Grefen, Y. Hoffner (1999) CrossFlow: Cross-Organizational Workflow Support for Virtual Organizations; Proceedings International Workshop on Research Issues in Data Engineering; Sydney, Australia, pp. 90–91
28. P. Grefen, K. Aberer, Y. Hoffner, H. Ludwig (2000) CrossFlow: Cross-Organizational Workflow Management in Dynamic Virtual Enterprises; Int. Journ. of Computer Systems Science & Engineering, Vol. 15, No. 5, pp. 277–290
29. P. Grefen, K. Aberer, H. Ludwig, Y. Hoffner (2001) CrossFlow: Cross-Organizational Workflow Management for Service Outsourcing in Dynamic Virtual Enterprises; IEEE Data Engineering Bulletin, Vol. 24, No. 1, pp. 52–57
30. C. Hagen, G. Alonso (1998) Flexible Exception Handling in the OPERA Process Support Systems; Procs. 18th Int. Conf. on Distributed Computing Systems; Amsterdam, Netherlands, pp. 526–533
31. M. Hsu (Ed.) (1993) Special Issue on Workflow and Extended Transaction Systems; IEEE Data Engineering Bulletin, Vol. 16, No. 2
32. M. Hsu, C. Kleissner (1998) ObjectFlow and Recovery in Workflow Systems; in [42]; Ch. 18, pp. 505–527
33. MQSeries Workflow Web Site (2001) http://www-4.ibm.com/software/ts/mqseries/workflow/; IBM, USA
34. S. Jajodia, L . Kerschberg (1997) Advanced Transaction Models and Architectures; Kluwer Adademic Publishers
35. M. Kamath, K. Ramamritham (1996) Correctness Issues in Workflow Management; Distributed Systems Engineering Journal; Vol. 3, No. 4, pp. 213–221
36. M. Kamath, K. Ramamritham (1998) Failure Handling and Coordinated Execution of Concurrent Workflows; Procs. 14th Int. Conf. on Data Engineering; Orlando, Florida, USA, pp. 334–341
37. B. Kiepuszewski, R. Muhlberger, M. Orlowska (1998) FlowBack: Providing Backward Recovery for Workflow Management Systems; Procs. 1998 ACM SIGMOD Int. Conf. On Management of Data; Seattle, USA, pp. 555–557
38. H. Korth, E. Levy, A. Silberschatz (1990) A Formal Approach to Recovery by Compensating Transactions; Procs. 16th Int. Conf. on Very Large Databases; Brisbane, Australia, pp. 95–106
39. N. Krishnakumar, A. Sheth (1995) Managing Heterogeneous Multi-system Tasks to Support Enterprise-wide Operations; Distributed and Parallel Databases, Vol. 3, No. 2
40. V. Krishnamoorthy, M. Shan (2000) Virtual Transaction Model to Support Workflow Applications; Procs. 2000 ACM Symposium on Applied Computing; Como, Italy, pp. 876–881
41. P. Krychniak et al. (1996) Bounding the Effects of Compensation under Relaxed Multi-Level Serializability; Distributed and Parallel Databases, Vol. 4, No. 4; Kluwer Academic, pp. 355–374
42. V. Kumar, M. Hsu (eds.) (1998) Recovery Mechanisms in Database Systems; Prentice Hall
43. F. Leymann, D. Roller (1994) Business Process Management with FlowMark; Procs. 39th IEEE Computer Society Int. Conf.; San Francisco, USA, pp. 230–234
44. F. Leymann (1995) Supporting Business Transactions via Partial Backward Recovery in Workflow Management Systems; Procs. Datenbanksysteme in Büro, Technik und Wissenschaft; Dresden, Germany, pp. 51–70
45. F. Leymann, D. Roller (2000) Production Workflow – Concepts and Techniques; Prentice Hall PTR
46. Object Management Group (1995) The Common Object Request Broker: Architecture and Specification, Version 2.0; Object Management Group
47. J. Ousterhout (1994) Tcl and the Tk Toolkit; Addison-Wesley
48. A. Reuter, F. Schwenkreis (1995) ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management Systems; IEEE Data Engineering Bulletin, Vol. 18, No. 1, pp. 4–10
49. A. Reuter, K. Schneider, F. Schwenkreis (1997) Contracts Revisited; in [34]; Ch. 5, pp. 127–151
50. H. Schuldt, G. Alonso, H-J. Schek (1999) Concurrency Control and Recovery in Transactional Process Management; Procs. 18th ACM Symp. on Principles of Database Systems; Philapdelphia, USA, pp. 316–326
51. J. Vonk, P. Grefen, E. Boertjes, P. Apers (1999) Distributed Global Transaction Support for Workflow Management Applications; Procs. 10th Int. Conf. on Database and Expert System Applications; Florence, Italy, pp. 942–951
52. J. Vonk, W. Derks, P. Grefen, M. Koetsier (2000) Cross-Organizational Transaction Support for Virtual Enterprises; Procs. 5th Int. Conf. on Cooperative Information Systems; Eilat, Israel, pp. 323–334
53. G. Weikum (1991) Principles and Realization Strategies of Multilevel Transaction Management; ACM Transactions on Database Systems, Vol. 16, No. 1, pp. 132–180
54. J. Widom, S. Ceri (Eds.) (1996) Active Database Systems: Triggers and Rules for Advanced Database Processing; Morgan Kaufmann