Beyond the Black Box: Event-based Inter-Process Communication in Process Support Systems[*]

Technical Report No. 303
ETH Zürich, Department of Computer Science

Claus Hagen Gustavo Alonso
Information and Communication Systems Research Group
Institute of Information Systems, Swiss Federal Institute of Technology (ETH)
ETH Zentrum, CH-8092 Zurich, Switzerland
{hagen,alonso}@inf.ethz.ch

Abstract

Existing workflow management systems encapsulate the data and behavior of a process within its execution scope, preventing other processes from accessing this information until the process terminates. There are, however, many situations in which allowing running processes to exchange information would be a great help to speed up processing time, improve service quality, and increase customer satisfaction. This paper describes the concept and implementation of an inter-process-communication facility based on the exchange of events between concurrently running processes. In contrast to previous approaches based on shared data stored in a common database, our approach has the advantage of being platform independent and providing straightforward support for distribution. In addition, we also explore the problem of process atomicity and consistency when events are revoked due to the abort of processes. The paper presents a family of recovery protocols that allow to control the effects of aborted processes.

1 Introduction

Process Support Systems for business applications (workflow tools) have attracted much attention in the last few years. In fact, the worldwide market for workflow technology was estimated in 1997 to be 1 billion $ [Koz97]. Thanks to a conceptually simple but very powerful abstraction – the notion of process –, workflow tools allow to define, enact, and monitor the execution of complex sequences of application invocations over heterogeneous, distributed environments. The simplicity of the notion, however, carries with it a number of limitations. Because of inherent restrictions in the way processes are currently implemented, the applicability of WFMS is still limited to specific domains and, inside these domains, to specific types of applications. An important research topic is therefore the extension of workflow principles in order to increase their applicability.

One of such inherent restrictions is the encapsulation of the data and behavior of a process within its own scope, information which cannot be readily accessed until the process terminates. This problem should not be confused with that of monitoring the process execution, although they are closely related. Monitoring allows to keep track of the state of a process but this information is not available to running processes. What is sorely missing in current commercial products and many research prototypes is the ability to exchange information between processes while they are executing. This form of cooperation is quite natural and appears in many practical applications. Note that this problem affects both the running instances of different and of the same process classes. Often, within one process, modular programming is achieved defining

[*]Part of this work has been funded by the Swiss National Science Foundation under the project TRAMs (Transactions and Active Database Mechanisms for Workflow Management).

blocks as sub-processes with exactly the same black-box characteristics as a normal process. Modularity is generally desirable to simplify the design but, given the coarse granularity of workflow activities, it is not reasonable to expect the same degree of encapsulation in a workflow tool as in a programming language. Indeed, given the lack of interprocess communication facilities, many workflow designers abandon any notion of modular design in order to accommodate the complex information flow between the different activities of a process. Unfortunately, it is very difficult to implement an elegant and efficient solution for these problems, even if monitoring tools provide the necessary information about the status of running processes. Our experience shows that such functionality, to be feasible and usable in practice, has to be part of the core of the system and has to be integrated within the notion of process. It cannot be added later on, as one can easily see in existing products where the internal architecture does not leave any room for interprocess communication [AM97, AHST97a].

In this paper, we address the problem of incorporating interprocess communication as part of the workflow engine. In doing this, the most critical constraint is not to rely on particular properties of a given system to implement the interprocess communication functionality, since otherwise we will lose the generality, applicability, and portability of the idea. It is important to keep this in mind since our approach is based on event- and rule-based mechanisms [PDW$^+$94, WC96] similar to those used in a number of research projects on event-based workflow architectures [CCPP95, GT96]. Many of these endeavors require the existence of either an active database system or a distributed event engine as the underlying platform. This requirement is often not practical since neither active databases nor distributed event engines are widely available today. Moreover, such architecture introduces a clear dependency on the functionality of the underlying system, thereby limiting the possibilities of generalizing the idea. Because our aim is to develop a generic process support tool that is independent of technical idiosyncrasies (see the OPERA project, briefly described below), we opt instead for an architecture in which interprocess communication is implemented as an additional module of the execution engine. This does not prevent us from using, for instance, an active database when it is available, but it means that we do not have to rely on the availability of such software to provide interprocess communication functionality.

The contribution is twofold. On the one hand, we provide an exhaustive study of the problem of interprocess communication in workflow environments from both a formal and a systems point of view. On the other hand, we describe in detail how to implement such functionality in a clean, elegant, and efficient manner without having to rely on the functionality of third tools. The implementation takes into account the whole system, i.e., it also incorporates necessary features like recovery and exception handling, which are often overlooked when dealing with events. In fact, it is recovery and exception handling that makes the problem of interprocess communication quite complex. In this regard, we believe the ideas presented here are a good starting point for any implementation of interprocess communication in workflow tools.

The rest of the paper is structured as follows. Section 2 presents an example of interprocess communication. Section 3 describes fundamental principles of process support systems. Section 4 describes the event mechanism in which we base our inter-process communication solution. Section 5 discusses the problem of correctness of the event mechanism from the point of view of recovery. Section 6 describes the system architecture. Section 7 presents an overview of related work and section 8 concludes the paper.

2 Motivation and Problem Statement

To better clarify the problem and to have an example to use throughout the paper, this section describes a practical scenario where interprocess communication is needed and discusses the implications of such functionality.

2.1 Example

The example we will use has been taken from a real application scenario [SST98] and represents a typical administrative environment with several heterogeneous systems. In this environment, administrative infor-

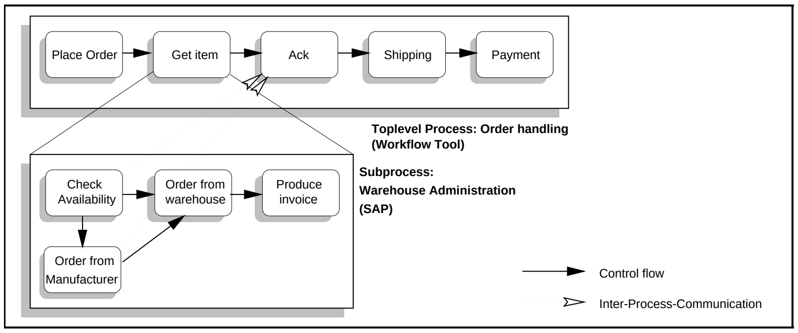


Figure 1: Example process

mation is stored and maintained in SAP R/3. SAP R/3 has its own workflow component [SAP97] allowing to model sequences of operations to be executed in different modules of the system. The workflow functionality is, however, restricted to R/3 activities, and the integration of external programs is not supported [1]. To interact with the rest of the system, a workflow management tool is used, which controls the overall process and invokes R/3 workflows as subprocesses.

Fig. 1 shows a process executed in such a configuration. The process handles the delivery of products by a retail company and captures all steps from the initial placement of the order by a customer to the final payment. The process has been divided into two atomic sub-processes, *warehouse administration* and *order handling*. The company is using SAP R/3 for stock administration, which requires to model the sequence of steps pertaining to warehouse administration as a SAP workflow and leave its execution to R/3. These subprocess, shown in the lower part of the figure, consists of four steps. First, the availability of the requested items is checked. If the item is not in stock, it has to be ordered from the manufacturer. If it is in stock (or when it has been delivered by the manufacturer), it can be ordered from the warehouse, which requires updating the stock databases. Finally, an invoice is produced based on the quantity taken from the warehouse, which completes the process as far as R/3 is concerned. The overall process, shown in the upper part of the figure, contains the initial placement of the order, which includes generating the necessary record of the products ordered. The "Get item" task, executed next, serves as a placeholder for the R/3 process. Upon execution of this task, the workflow tool initiates the execution of the R/3 process. After the products have been extracted from the warehouse, an acknowledgment is sent to the customer announcing the expected delivery date. The final steps of the workflow are concerned with packaging and shipping of the product and with the handling of the payment.

2.2 The need for interprocess communication

The process shown in Fig. 1 can be modeled in several ways but each subprocess necessarily appears as a "black box". As indicated in the introduction, one can always write the entire process in one single system. For instance, one could circumvent the R/3 workflow engine in this case and integrate the SAP related tasks into the top level process, invoking each of them as external activity. This would permit intertask communication, but the solution is both inelegant and impractical since it makes processes unreadable and difficult to maintain. On the other hand, structuring a process like in Fig. 1 greatly decreases the degree of parallelism since progress in the top level process is not possible until the complete subprocess has finished. Moreover, it increases the response time of the system, which may negatively impact several important organizational aspects like service quality and customer satisfaction. In the example, for instance, the acknowledgment to the customer could be sent as soon as it is known that the product is on stock, thereby giving the customer an impression of faster service. Similarly, the shipping could be initiated as soon as it

[1]SAP R/3 is a good example but the same can be said of almost all workflow products in the market. Some of them do not even provide a mechanism to read the data out of the system once a process has terminated

is known from which warehouse the product will be delivered and at which date it will be available, instead of having to wait until the whole warehouse administration process concludes.

Once interprocess communication is in place, a number of other problems become also easily solvable. Extending the process can be done by simply defining a new subprocess which communicates with the existing one. Consider adding a third process which checks the customer credentials before proceeding with the purchase. The integration of a new subprocess into an existing framework is not easy to achieve in current systems, but it would be greatly simplified if interprocess communication were possible: It would suffice to let the "customer credentials" process know that a customer has placed an order in order to trigger the procedure. Once the checking is done, this is communicated to the order handling process in a similar fashion which can then proceed with the order.

These and other possible interprocess communication events (shown in Fig. 1 as dotted lines) can contribute significantly to the design of more efficient processes, as well as allowing more complex applications to be implemented with a workflow tool. Nowadays, there is a quite large number of applications that cannot take advantage of workflow technology because of the limited interaction allowed between processes.

2.3 Event based interprocess communication

One way to implement interprocess communication is to use *events*. Events are typed and parameterized signals that can be raised by an running task to inform other processes of certain situations reached or produced during its execution. In our example, the R/3 subprocess would raise either an event `product_available` or an event `product_not_available` as soon as it is known whether the product is in stock or needs to be ordered from the manufacturer. Both events could as well carry the estimated delivery date as a parameter. Upon receiving the signal, the base process can proceed by sending the corresponding acknowledgment to the customer and preparing the shipment based on the estimated delivery date.

Events have several important advantages over other possible mechanisms (specially over cooperative transaction approaches [Elm92]). Among these:

- Events would allow the integration of heterogeneous workflow tools without having to worry about a common API. A common event notification mechanism suffices.
- Events also allow a process to react to asynchronous occurrences of relevant situations without having to follow the rigid control flow of conventional workflow tools. An example is the integration of groupware tools (e.g. conferencing systems) with process management. A conference has no predefined structure, nevertheless there are certain situations, like the arrival of a given message, which may require the triggering of a process or may influence the flow within a running process [SL95].
- For many employees, the strictness with which many WFMS enforce the process specification limits their acceptance. The higher the skills of a user, the higher the probability that she will refuse to follow rigid prescriptions of a particular order of activities. The event mechanism allows to relax the control flow specification within a process so that a user can decide on the order of activities by herself and simply signal the end of a particular activity when finished.
- Failures that occur inside an invoked program are a special class of events that requires special treatment. The handling of exceptions can be greatly improved when an event-like mechanism is used to report information about the failure which has occurred back to the invoker, thereby giving a chance to fix the failure without having to abort the whole process [HA98].

Note that using events in this fashion is not privative of applications based on sub-processes. A similar approach could be followed in a large class of processes which can be characterized as *phased*, i.e., consisting of a number of phases where the end of a phase might require the start of additional activities in another process.

2.4 Recovery

Processes must at all times guarantee consistency. This is not so simple when the processes interact with each other. In a first approach, what we need is to have *atomic processes*, i.e. processes which are guaranteed to either succeed or to be completely undone. Unfortunately, providing atomicity in workflow contexts is complicated because of the side-effects of the invoked activities in the outside world. A simple rollback, as it is done in database systems, is normally not possible. Instead, *compensation mechanisms* are used to logically undo the effects of activities [EL96, Ley95, HA98]. Allowing inter-process communication adds a new dimension to the atomicity problem.

In the previous example, for instance, consider the case that the warehouse administration is aborted (it might turn out after the product is requested from the manufacturer that it is not produced anymore). We can consider this case as an exception that requires, similar to the occurrence of a failure in database transactions, the abort of the process. The interesting question is now how to treat the events that have already been signaled by the aborted process. Clearly, in the above example the obvious solution would be to inform the customer and then abort the overall process too. There are, however, many cases in which to abort the receiver of the event is not the best solution. In many situations, the invalidation of an event can be solved by the execution of some alternative activities which allow to substitute the failed activity. In other cases, only a partial abort might be required, while some of the effects already executed have to be made persistent.

In what follows, we discuss in detail the event mechanism, its implementation, and the problems associated with recovery when interprocess communication takes place. Then we provide an integrated framework allowing the specification of a family of event recovery schemes for workflow tools.

3 OPERA - a generic process support system kernel

We have implemented the interprocess communication mechanism as part of OPERA, a generic process support system kernel which aims at integrating middleware technology relevant to process support in distributed systems. The OPERA concept is described in detail in [AHST97a, AHST97b]. We will describe here only those aspects that are important for the paper.

3.1 Modeling language

Although OPERA supports a 3 level hierarchy of process representations (application specific, internal format or OCR, and low level representation) [AHST97a], in here, we will use the OPERA graphical workflow language (OGWL) for the purpose of describing language extensions to capture new semantics. OGWL follows similar languages used in in commercial systems [LA94] and is to a large degree conformant with the standard of the workflow management coalition [WMC94]. One of its advantages is its simplicity, which allows to easily apply extensions described in it to other, more complex, languages.

The main components of a process are shown in Fig. 2. It consists of a set of tasks and a set of data objects. Tasks can be activities, blocks, or processes. The data objects store the input and output data of tasks and are used to pass information around. *Activities* are the basic execution steps, meaning that each activity has an *external binding* that specifies a program to be executed, a person responsible for performing this part of the workflow, and/or resources to be allocated for its execution. This information is used by the runtime system to execute external applications or instruct users to perform certain actions. In the simplest form, a process consists only of activities and control and data flow descriptions (Fig. 2). Control flow inside a process is based on *control connectors* which, formally, are a triple (T_S, T_T, C_{Act}), where T_S is the source task, T_T is the target task, and C_{Act} is an activation condition. Each activator defines an execution order between two tasks and can in addition restrict the execution of its target task based on the state of data objects, thereby allowing conditional branching and parallel execution. *Data flow* is possible between tasks and between processes. Each task has an *input data structure* describing its input parameters and an *output data structure* to store any return values. The input parameters of a task can be linked to data items in

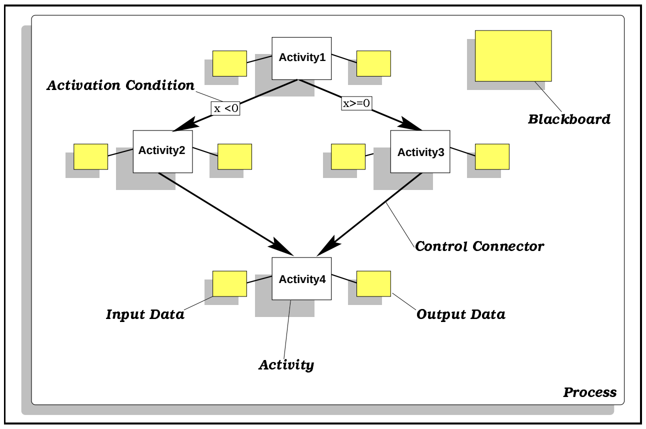


Figure 2: The main constructs of the graphical user-level language (OGWL)

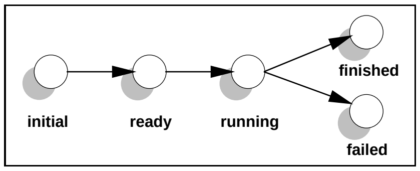


Figure 3: Basic State Diagram for tasks

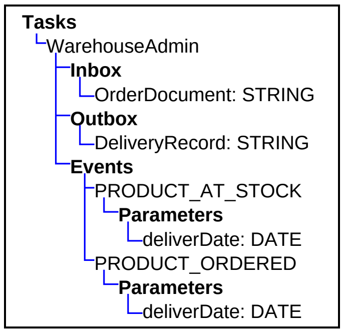
the process' global data area (the blackboard) or in other task's output structures. When a task starts, these bindings are analyzed and the necessary values passed to the task. After the successful execution of a task, a *mapping phase* transfers data from its output structure to the global data area. The input and output data structures of a process are part of the *blackboard* which acts as the global data area for the process.

Larger processes are structured using one of two nesting constructs. *Blocks* are sub-processes defined only within the scope of a given process and are used for modular design and as specialized language constructs (for, do-until, while, fork). Blocks also serve as *spheres of atomicity*, as it will be discussed later. *Subprocesses* are processes used as components of other processes. Subprocesses allow, like blocks, the hierarchical structuring of complex process structures. Late binding (the referenced process is read only when the sub-process is started) allows dynamic modifications of a running process by changing its sub-processes [IBM95]. Blocks and subprocesses allow for the structured modeling of workflows, with leads to a process structure resembling a tree of tasks, where the intermediate nodes are blocks and subprocesses, and the leaf nodes are always activities.

The *state* $S(T)$ of a task T describes whether it is already executing, has finished or failed (Fig. 3). We also distinguish between the states *ready* and *initial*. A task is initial when its process is started. It becomes ready when its activator and condition are true. It becomes running, when the system is sure that it really is executing. This distinction is important because tasks (resp. their external programs) execute on remote hosts and the PSS has no complete control about them. If, for example, a task has to be executed by a human, the system will mark the task as running not before the user acknowledges that he is working on the task. Similar, if the task incorporates the execution of an external program, the task is marked as running not before an acknowledgment has been sent that the task program has been started. This is important for fault-tolerance, since it allows to recover the state of the system as accurately as possible.

3.2 Atomic spheres

OPERA integrates transaction concepts directly into the process support system. These concepts have been described in detail in [AHST97a, HA98, Hag97]. In general, OPERA uses the notion of *spheres* to bracket



```
Tasks
└WarehouseAdmin
  ├Inbox
  │ └OrderDocument: STRING
  ├Outbox
  │ └DeliveryRecord: STRING
  └Events
    ├PRODUCT_AT_STOCK
    │ └Parameters
    │   └deliverDate: DATE
    └PRODUCT_ORDERED
      └Parameters
        └deliverDate: DATE
```

Figure 4: Declaration of the warehouse administration task, including events

operations as units with transactional properties. Spheres are specialized blocks used as: atomic units with the standard all or nothing semantics (sphere of atomicity), isolation units (sphere of isolation), or persistence units (spheres of persistence) [AHST97b]. For the purpose of this paper, the most relevant spheres are spheres of atomicity, hence they are the ones that will be described here in detail.

Atomicity is a property that can be declared for blocks, processes, and activities by including them in an atomic sphere. This is done in the case of activities by the person registering a program or human activity. An activity declared as an atomic sphere has either to be inherently atomic (like a database transaction), or an activity has to be defined that is able to logically undo its effects. In the case of blocks and processes, atomicity is automatically determined based on the properties of the component steps. The atomicity of a process can be guaranteed if it either is declared to be semi-atomic (in this case, the process designer has to provide a *backout method* that performs rollback), or if (a) every component task is atomic or semi-atomic and (b) every component task is compensatable or the process has a *flex-structure*. A flex structured process conforms to rules originally developed as part of the flex transaction model ([ZNBB94]). The information provided by the process designers is used in the following way: If a task has to be aborted, it stops (note that this may require recursive abort of component tasks in the case of a process or block) and is then undone depending on its type. If the task is atomic, then no further actions are performed. If the task is semi-atomic, then *holistic backout* is performed by executing a previously declared rollback method. If the task is non-atomic, then *single-step backout* is performed by executing the compensating tasks of the component activities.

4 Inter-Process Communication

This section describes the interprocess communication mechanism we have implemented based on the notion of events described above.

4.1 Events

An *event type* is a tuple (N, PL), where N is an event name and PL is a parameter list. Event types are registered with the system as part of a task declaration. During execution, tasks signal events (i.e., create *event instances*) by specifying a type name and providing values for all parameters in the parameter list. The parameterization allows to pass arbitrary context data with the event, from simple boolean values to references to external objects, and it is the key to effective interprocess communication. A task is free to signal an event arbitrarily often or even not at all. This gives the event mechanism the necessary flexibility to capture all situations of interest.

The language primitive for signaling an event depends on the flavor of the particular modeling language used. In the case of OGWL and other graphical workflow languages, pseudo-activities could be used. They can be embedded into a process model but have no external binding. Instead, the system generates an event if such a pseudo-activity is executed. The advantage of this approach is its seamless integration with the control and data flow primitives already present in the language.

At run-time, a task can signal only the events listed in its *event-declaration* which is, like the input and output parameters, part of the signature describing the task interface. As an example, see the interface structure of the warehouse administration task from our initial example (Fig. 4). Here, two event types are defined. Each of them has one parameter of type *DATE*, which allows to communicate the expected delivery date of the ordered product. While the task is executing (remember that this implies the execution of the R/3 subworkflow), it may at any time raise one of the two events, thereby signaling either that the product is on stock or that is has been ordered and returning the estimated delivery date. Practically, this will happen when the first activity of the subworkflow has finished. (The SAP workflow module has a callback mechanism that can report state changes in an workflow to the invoker through its API and that can be used to signal the event.)

4.2 Introducing events in the process model

Formally, adding events to our model requires to extend the state model described above. To this end, we define the extended state of a task, $S'(T) := S(T) \cup EQ(T)$, where $EQ(T)$ is the *event queue* of that task. The event queue records all events signaled by the task in the order they occured, and forms the basis for the integration of events into control flow decisions.

Starting a given task can also be made dependent on events that occur in sibling tasks belonging to the same process. This is enabled by a new class of control connectors. An *event based control connector (ECC)* is a quintuple $(T_T, T_S, C_{Ev}, C_{Act}, RM)$, with two new elements, an *event condition* C_{Ev}, which can be an arbitrary predicate over the event queue of the source task, and a *recovery mode* RM. During process execution, the target task is considered for execution as soon as the predicate evaluates to true, which may be any point in time before the source task has finished. An ECC allows, thus, to define the flow of control based on the occurrence of events, while the original, state based control connectors, can only react to the *finished* state of tasks. CCs and ECCs can be arbitrarily combined in the same process, allowing both event driven and flow driven execution. The recovery mode RM determines the reaction of the system if the event should be revoked due to the abort of a task. This issue is discussed in Section 5.

4.3 Scope of events and subscription

The visibility of an event is limited to the block or process a tasks belongs to (similar to local variables in a programming language). To enable inter-process communication, a subscription mechanism is used. A process can register for any visible event, using a *subscription statement* of the form (T, E, N), where T is the task the event is imported from, E is is the event type, and N is a *nickname* under which the event is made accessible to the component tasks of the subscriber. Similar to the event queue of a task, each process has an *imported event queue (IEQ)* which makes all imported events visible to its component tasks. This mechanism supports the propagation of events over several levels of nesting, since a subscription statement issued by a subprocess can refer to an event of its parent's IEQ.

Similarly, each process has an *exported event list* which allows to externalize events signaled by its component tasks. It contains declarations of the form (T, E, N), where T is a component task of the process, E is an event declared by it, and N is the nickname under which it is accessible to other processes. The exported event list forms, together with the event declaration of a process, the complete declaration of all events the process may signal. The export mechanism allows events to arbitrarily "climb up" the nested scopes of a process, while the import mechanism lets them "climb down". Together, they allow for inter-process communication while the ECC concept supports intra-process communication only.

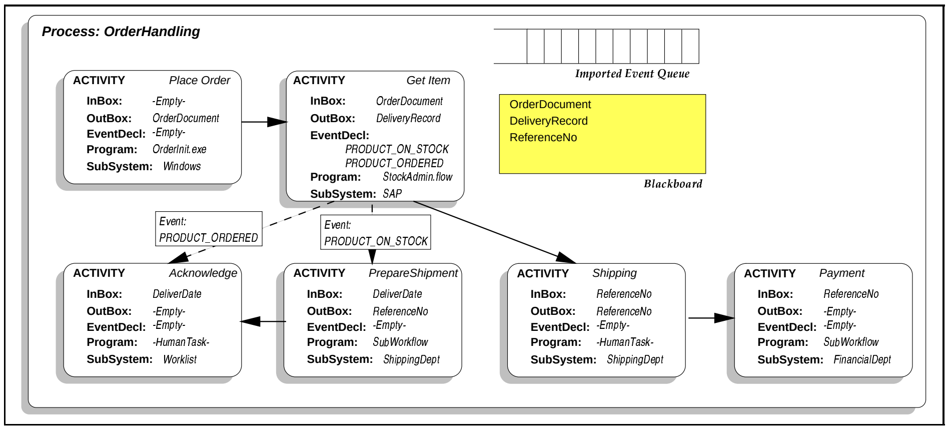


Figure 5: The initial example, now modeled using events for intra-process communication

4.4 Example

Fig. 5 and Fig. 6 show and summarize these ideas using the initial example, now described including the event mechanism. To keep the pictures comprehensible, we have omitted a number of details like the full definition of all tasks and certain aspects of data flow. Fig. 5 shows the intra-process communication mechanism. The GetItem task is now defined according to Fig. 4, declaring two event types. (Remember that this task represents a whole subprocess, executed by the R/3 system.) We have added two event based control connectors (the broken lines) that react to the occurrence of the respective events in the SAP workflow. We have also divided the shipment activity into two steps, where the first one prepares the shipment (by allocating resources, transport facilities, and the like), and the second one triggers the actual shipping. If the requested item is on stock, SAP will signal the PRODUCT_AT_STOCK event, which initiates (through an ECC) the execution of the shipment preparation step. Then the acknowledgment step is started, which informs the customer of the estimated delivery date. If the product has to be ordered, the shipment preparation is omitted. In this case, the Acknowledgment task is directly started, informing the customer of the delay. The actual shipping activity is executed when the subprocess (and with it the GetItem task) has finished, ensuring that the shipment starts properly as soon as the product is available. Note that in this example, the imported event queue of the process is not used since all event exchanges are among tasks at the same level in the process.

Fig. 6 illustrates the inter-process communication mechanism. It describes the same application, but this time modeled using two parallel subprocesses communicating through events. Upon start of the parent process, both subprocesses are started, and the *PlaceOrder* activity is invoked (it is the only activity marked as *StartActivity*, enforcing its immediate begin). When it ends, as well as after the *CheckAvailability* task and at the end of the *StockAdmin subprocess*, events are generated and delivered to the respective other subprocess. Note that each of the two subprocesses subscribes to the events declared by the other, and that activities can access all events through the *imported event queue* using ECCs.

This second example shows the tight relation between the interprocess communication mechanism and event based workflow architectures [GT96], giving the workflow designer the option to choose between two flavors of modeling: Either using process/subprocess hierarchies with RPC-like semantics, or combining independent processes that communicate by exchanging events. When to use each one of them depends on design methodology and the particular characteristics of the application environment.

4.5 Exceptions

Exceptions occurring during process execution can be seen as special kinds of events. The event mechanism described before can be used as the basis for a powerful exception handling facility in process support

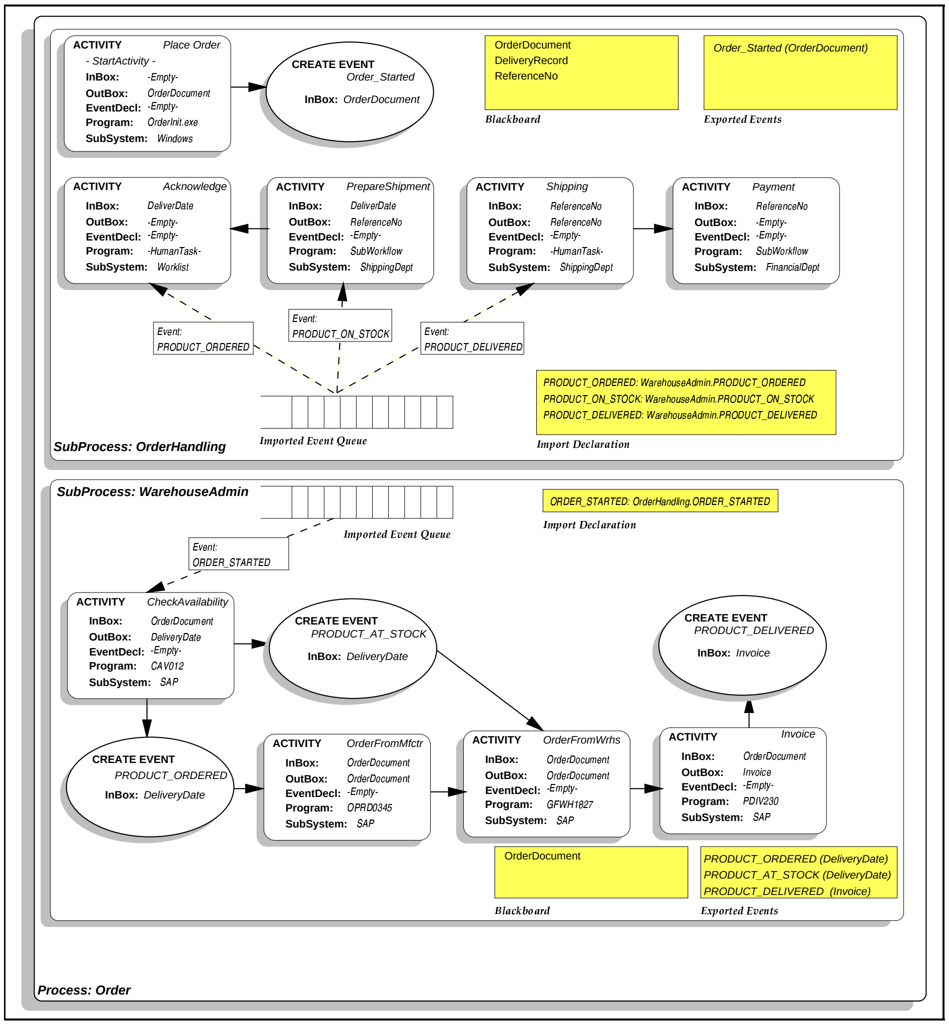


Figure 6: Two subprocesses using inter-process communication

systems [HA98]. This mechanism is based on the principle that upon the detection of a failure, an *exception event* is generated. Like ordinary events, exception events (or, shortly, exceptions) are typed and have a parameter list, so they can carry information about the failure type and its circumstances. Contrary to ordinary events, however, exception events cannot be subscribed to by regular tasks. They are only delivered to dedicated *exception handlers*, which are specialized tasks. If an exception handler receives an event, it first performs all steps necessary to deal with the failure (like executing arbitrary contingency operations, querying a user for modified input data, or informing a system administrator of the failure). After the execution of these procedures, the handler decides how to proceed. There are two possibilities: The failed task is either aborted or resumed. In the case of an abort, the system performs rollback if the task was declared to be semi-atomic (see section 3.2). If the handler decides to resume execution, the failed task is restarted, probably with new input values provided by the user interaction. The resume option is a valuable mechanism in workflow systems with a high degree of human participation since it can be used to implement a “callback mechanism” which allows the user to influence the behavior of the task. Exception handlers can be defined by the workflow designer for each task/exception pair, providing hooks that allow to customize failure handling in arbitrary ways.

To illustrate the concepts, several examples for the flow of control during exception handling are shown in Fig. 7, depending on the decision of the handler. In diagram (a), the exception handler resumes execution of the signaler. In diagram (b), the signaler is aborted and control returns to the process that invoked it.

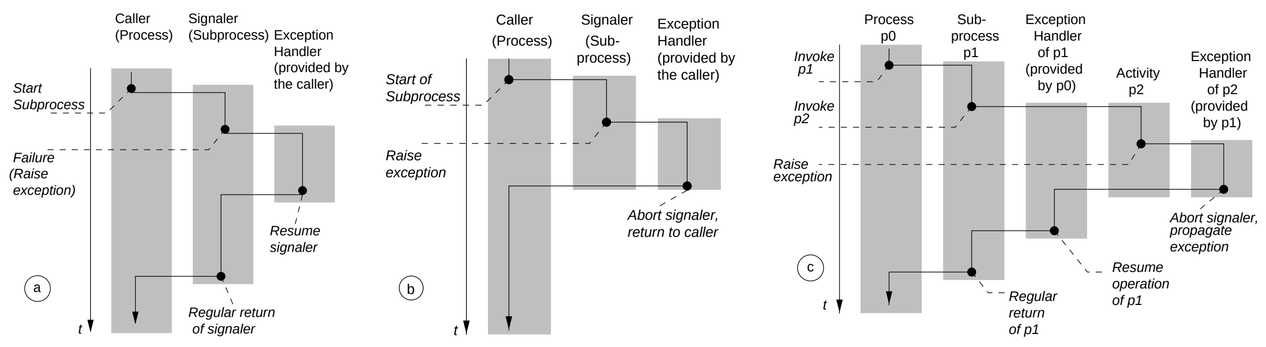

Caller (Process)
Signaler (Subprocess)
Exception Handler (provided by the caller)
Start Subprocess
Failure (Raise exception)
Resume signaler
Regular return of signaler
a
t
Caller (Process)
Signaler (Sub-process)
Exception Handler (provided by the caller)
Start of Subprocess
Raise exception
Abort signaler, return to caller
b
t
Process p0
Sub-process p1
Exception Handler of p1 (provided by p0)
Activity p2
Exception Handler of p2 (provided by p1)
Invoke p1
Invoke p2
Raise exception
Abort signaler, propagate exception
Resume operation of p1
Regular return of p1
c
t

Figure 7: Control flow during exception handling

Diagram (c) shows a two-level nested execution, where the innermost process (p2) raises an exception which is propagated by the exception handler, enforcing the abort of p2 and the invocation of an exception handler associated with p1. This handler resumes the operation of p1.

The event-based exception handling mechanism, described in detail in [HA98], provides further evidence for our initial thesis that weakening the black box approach in a controlled way can greatly improve the functionality of workflow management systems. The exception mechanism is nevertheless relevant in this paper not only because of its tight relation with the event mechanism, but also since it can also be used to handle the event revocation problem sketched in section 2. To this end, in the case of the abort of an atomic sphere, the system generates an exception for each event invalidated as a result of the rollback. These new exceptions are sent to all affected event subscribers and lead, through the exception handling algorithm sketched above, to the invocation of a user-provided operation that can react to the event revocation. We will discuss handling strategies for revoked events in the next section.

5 Recovery algorithms for event revocation

Event revocation denotes the case in which an event that was intercepted by a task or process is undone because of the abort of the event signaler. We have already argued in section 2 that no single strategy will suffice to resolve all possible situations. Our goal is, therefore, to provide multiple handling strategies and to give the process designer the means to select the most appropriate strategy for the application.

5.1 Notation

Let $TSET(t)$ denote all tasks that are *active* at time t. An active task is one whose event queue contains events that may be subject to revocation. Let $ESET$ denote the set of these events, i.e., $ESET := \bigcup\{EQ(T)|T \in TSET\}$. The subscriber relation $SUBSCR(t)$ contains all pairs (T_{PROD}, T_{SUBSC}) of active tasks where T_{SUBSC} has subscribed to an event that is declared by the producer T_{PROD}. The causality relation $DEP_{EV}(t)$ contains all pairs (T_{PROD}, T_{CONS}) of tasks where T_{CONS} is causally dependent on the active task T_{PROD} because the execution of T_{CONS} was triggered by an event in $EQ(T_{PROD})$, or, in other words, a process has an event-based control connector with T_{CONS} as the target and an event of T_{PROD} was referenced in the predicate. Note that T_{PROD} has to be active, but T_{CONS} can be an already finished task. This captures the fact that a consumer may finish before the producer it depends on. In these cases, compensation is necessary to undo the effects of the consumer "post mortem". Let $DEP_{EV}(t, T)$ be a shortcut for $\{T_{CONS}|(T, T_{CONS}) \in DEP_{EV}\}$.

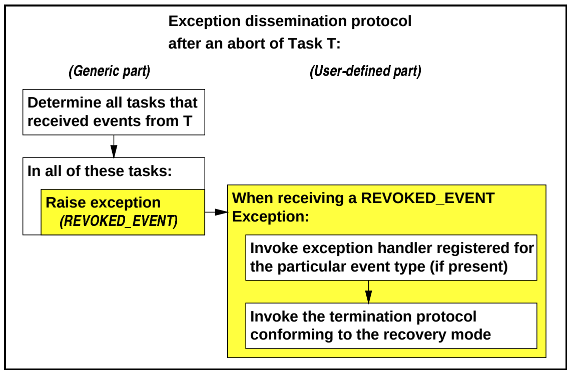

Exception dissemination protocol after an abort of Task T:
(Generic part)
(User-defined part)
Determine all tasks that received events from T
In all of these tasks:
Raise exception (REVOKED_EVENT)
When receiving a REVOKED_EVENT Exception:
Invoke exception handler registered for the particular event type (if present)
Invoke the termination protocol conforming to the recovery mode

Figure 8: Exception dissemination algorithm

5.2 Exception dissemination

The exception mechanism described in the previous section is an ideal "transport medium" to initiate recovery in the case of event revocation. Its principle is simple as shown in Fig. 8. If a task T is aborted, the system determines $DEP_{EV}(t,T)$, the set of all tasks that dependent on T through a consumer relationship. To this end, a registry of all currently existing consumer-producer relationships is maintained by the event manager in a persistent table (see section 6 for an overview of the system architecture). In all consumer tasks determined using the registry, an exception of a predefined type is generated which, through the usual exception handling mechanism, initiates the customized recovery procedure as defined by the workflow designer. This procedure is executed in two phases. First, if defined, an exception handler is executed which performs contingency and/or cleanup operations. Then a termination protocol is invoked which can abort the consumer completely or perform partial rollback. Its behavior is defined by the *recovery mode* specified for each consumed event in the process description.

Note that the order in which the exception dissemination protocol is executed is important. The import and export mechanisms permit a task to have customers on several levels of the same process tree. In these cases, the exceptions are handled in a top-down style: If a parent process and one of its descendants have a causal dependency on the same task, the parent process has to be handled first. This avoids unnecessary work and speeds up processing, since aborting the parent results in an abort of its children as well, making exception handling in the child obsolete.

5.3 Termination protocols

The *recovery mode* specified by the process designer for each event-based control connector determines the fate of a consumer task after the execution of the exception handler. Based on the recovery mode, a termination protocol is executed. The following recovery modes and termination protocols are defined:

IGNORE. No further action is performed, which can be useful if (a) an abort is impossible or too expensive because of the amount of work already performed, or (b) if it is known from the application semantics that no recovery is necessary. Consider our initial example, where the *Acknowledge* task informs the customer. Since this task left unrecoverable side-effects in the outside world, the obvious strategy is not to abort it, but invoke a contingency action (send a new letter) through the exception handler.

ABORT. Abort the parent process of the consumer. This is useful in those cases where after event revocation no forward progress is possible and hence a complete abort is the only feasible strategy.

PARTIAL_ROLLBACK. This is the most interesting recovery mode. It performs a partial rollback, undoing all work causally dependent on the aborted task. After this rollback, the execution continues from the last consistent state before the the original occurrence of the event. While the principle is simple, its implementation turns out to be complicated because of the complex structure of the causality relationship. We investigate this in detail in the next subsection.

5.4 Partial rollback algorithm

Between all tasks in the system, a causal dependency [Lam78] is defined through the ordinary control connectors, data exchanges through the blackboard, and the event mechanism. The goal of the partial rollback algorithm is to set the system back to a consistent state by undoing all effects causally dependent on an aborted task. To this end, the transitive closure of the causal dependency relation is computed and tasks are aborted in the reverse order of their execution. We define the relation $DEP_{ALL}(t)$ over all pairs of tasks in the following way: (T_1, T_2) are in $DEP_{ALL}(t)$ if (1) T_2 was executed after T_1 and both were linked by a control connector, or (2) if T_1 raised an event that caused the start of T_2, or (3) if T_2 read data from the blackboard that was written by T_1. This dependency information can easily retrieved, since usually a workflow system logs all information pertaining to process execution in its execution log for legal reasons and to facilitate process optimization based on statistics [AGL98]. We denote with $DEP^*(t)$ the transitive closure of $DEP_{ALL}(t)$. As a shortcut, we define $DEP^*(T,t)$ as the restriction of $DEP^*(t)$ on those tasks causally dependent on T, i.e., $DEP^*(T,t)$ contains all tasks transitively causally dependent on T.

The goal of the partial rollback algorithm is now to undo all work dependent on an aborted task by aborting all tasks in the dependency set of this task. The algorithm maintains two data structures: S_{DONE}, storing all tasks that have already been handled by the recovery algorithms, and Q_{TODO}, a FIFO-organized list of tasks yet to be considered for recovery. At the beginning of the algorithm, S_{DONE} is initialized with the task that originally failed and caused the recovery procedure and with T, the task that caused the start of the partial rollback algorithm. The Q_{TODO} is initialized with all consumers of events signaled by T. The algorithm proceeds in two phases:

1. First, determine all tasks that are dependent on T because of regular control and data flow. These are tasks belonging to the same process as T. Abort these tasks in reverse order of execution. If an aborted task had consumers, add them to the Q_{TODO}.

2. Perform a breadth-first traversal of the graph containing all causal dependencies on T caused by events. This is done by traversing the Q_{TODO}: An exception is sent to the task at the head of the Q_{TODO}, which is resolved through the exception handling mechanism. If the task has the PARTIAL_ROLLBACK option set, its consumers are appended to the Q_{TODO}, and the task itself is added to the S_{DONE}. Then the algorithm continues by taking the next task from the head of the queue. If a task extracted from the queue is already in the S_{DONE}, it is ignored. This algorithm proceeds until the queue is empty.

After the execution of the algorithm, the system is in a consistent state again. In large process sets with many interprocess dependencies, using only partial rollback strategies can cause significant overhead. The process designer, however, has the option to limit the amount of rollback work to be done by deliberately choosing ABORT or IGNORE recovery methods whenever they are applicable.

Fig. 9 illustrates the recovery algorithms. There, three processes communicate by exchanging events. At the top, we show the current state of the processes at the point in time when a failure occurs in Process1, which leads to its abort. At this point, the first 2 activities of each process have finished, and the last ones are still running. The abort of Process1 leads to the revocation of E1. Subsequently, the exception dissemination algorithm sends exceptions of type REVOKED_EVENT to $T22$, which was the only consumer for this event and has the recovery mode PARTIAL_ROLLBACK. If an exception handler was defined in $T22$, it is executed now. Then the partial rollback algorithm is started, with S_{DONE} containing $T11$, $T12$, $T13$, and $T22$. Q_{TODO} contains $T32$, since it consumed the event $E2$ signaled by $T22$. The first, local phase of the algorithm leads to the abort of $T23$, which has generated no events. Then, the second phase is started with an exception being generated in $T32$ which is at the head of the queue now. This task had generated event E3, which was consumed by $T13$, so $T13$ is appended to the Q_{TODO}. Exception handling for $T32$ leads to an abort of $T33$, and then $T13$ is extracted from the queue. Since this task is already in the S_{DONE}, it is ignored. Now the queue is empty, and the system is in a consistent state again.

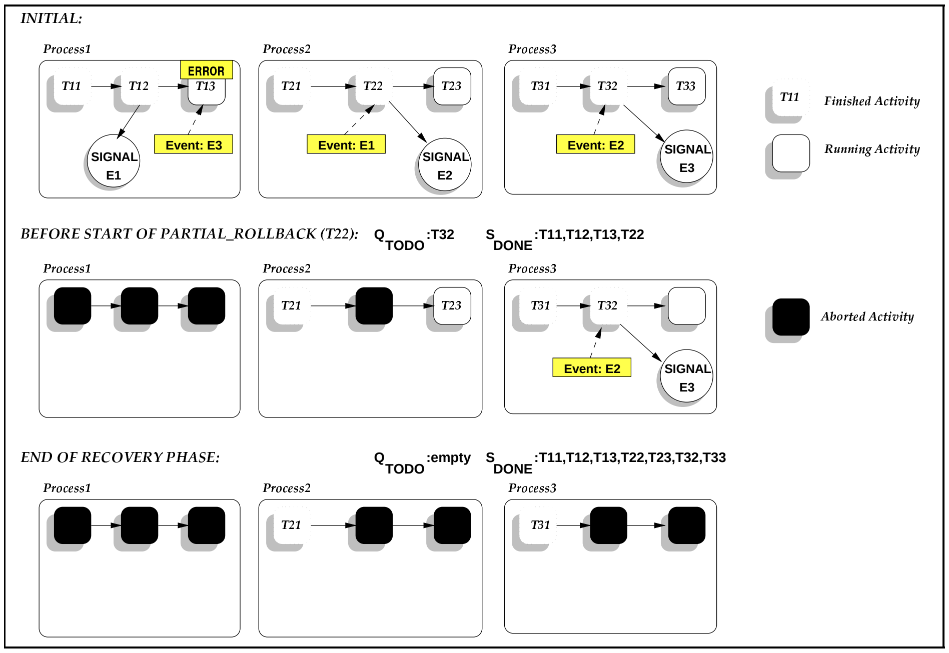


Figure 9: Example for the event revocation algorithm

5.5 Advantages of the approach

The mechanisms described above give the process designer the freedom to choose from a family of recovery strategies in the case of a failure. By combining the exception handler option and the recovery method, it is possible to implement the following algorithms:

- Ignore the failure by providing no exception handler and choosing the strategy IGNORE. This is useful in all cases where a task had no side effects and thus no recovery is necessary.
- Perform additional operations, but no abort (strategy IGNORE, provide an exception handler that invokes alternate activities). This will be used if contingency operations need to be invoked, but an abort is not necessary or possible.
- Abort the process containing the failed task, which is the strategy to choose when no graceful degradation is possible and a complete abort is the only solution.
- Invoke contingency operations that *replace* the invalidated task (strategy ABORT, provide an exception handler implementing the contingency logic). This is the option to be used if semi-atomic tasks have to be aborted and some cleanup work has to be done to really undo their effects.
- Rollback to the state before the event occured, and continue work from there (use the PARTIAL_ROLLBACK method).

6 System Architecture

The event services can be encapsulated in an *event manager* which is part of the PSS server and interacts with the other modules of the system. Fig. 10 shows this interaction using the OPERA system architecture [AHST97b] as an example. In this architecture, the workflow control functionality is divided between two components. The *navigator* interprets the process models, enforcing control and data flow and auditing all relevant transitions. The *dispatcher* is responsible for invoking and monitoring external applications,

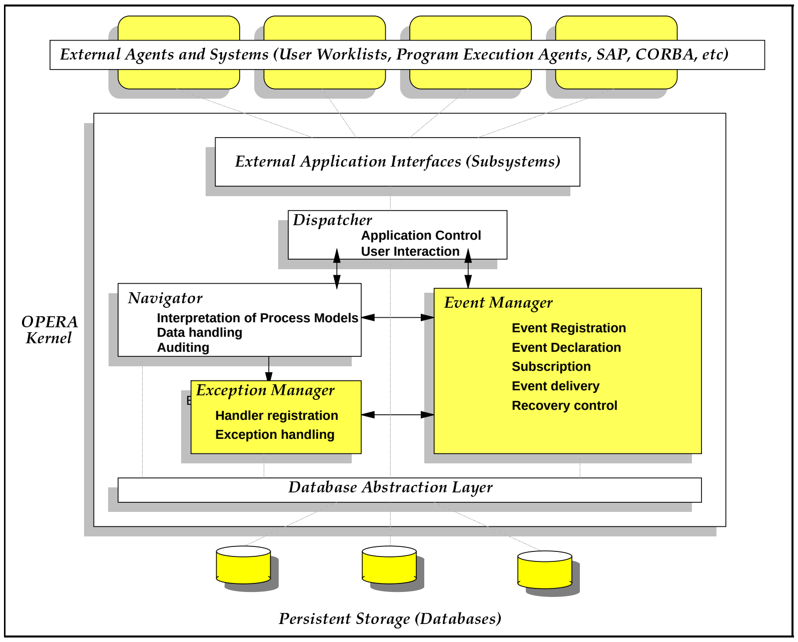


Figure 10: Integration of the event mechanism into PSS system architecture

sending requests to users, and controlling the externally accessible agents through application interfaces. Both dispatcher and navigator store administrative data in persistent storage accessible through an *database abstraction layer* shielding the system from the particularities of a specific database architecture. The event manager uses the persistent storage to keep runtime information, namely a registry of all existing event subscriptions and export declarations and a table with all currently existing consumer/producer relationships. Note that it would in principle be possible to derive subscription information directly from the process models – storing this data separately does, however, speed up event delivery tremendously. The consumer relationship information is needed for event recovery only. It is used to compute the dependency sets if events have been revoked.

Of the main system components, only the navigator interacts directly with the event manager, sending it subscription and export information whenever a new process is instantiated, and reporting the occurrence of events in external applications or processes. In turn, whenever an event occurs, the event manager computes the set of subscribers and passes it back to the navigator, which performs intra-process communication, checking control connectors and starting affected tasks and activities. Events that occur in external programs are reported to the event manager indirectly: The external applications inform the dispatcher through the interface layer, which in turn informs the navigator. This indirection allows the navigator to update the persistent state of the processes (which contains the state of the event queues) before the event manager is informed. The exception manager is another system extension that handles all tasks related to the exception mechanism. It keeps a registry of registered exception handlers and starts the exception handling procedures whenever it is informed of an exception event by the event manager. Due to space limitations, we cannot present the OPERA architecture concept in detail here, but an important aspect to note is that, because of the modular design of the event and exception manager, they can easily be integrated into other process support systems.

7 Related work

[CCPP96] provides a classification of possible workflow interactions if the processes access common data. A *workflow integration* approach is proposed that aims at identifying and removing interferences between concurrent workflows. The paper does not discuss event-based interaction of processes, and aspects of exception handling and recovery are not evaluated.

Contrary to what might be expected, the work done in the context of active database management systems (ADBMS) [PDW$^+$94, WC96] does not apply directly to what we discuss in this paper, although the ideas proposed could be implemented on top of an ADBMS. ADBMS allow to define events as part of the event-condition-action (ECA) rule paradigm: Upon the occurrence of an event, which can (among others) be an operation executed as part of a transaction, the associated condition (a predicate defined on data objects) is evaluated and, if successful, the *action*, a subtransaction containing arbitrary operations, is executed. While this principle appears to be related to our approach, a closer look reveals a fundamental difference: ECA rules cannot be used to link existing transactions. Instead, the action is executed either as part of the transaction raising the event (in form of a subtransaction), or a new transaction is started. Both cases simplify recovery tremendously, because the action can simply be aborted in the case of an event revocation. This may, in the worst case, require cascading aborts, but the isolation provided by the transaction ensures that no interferences with other transactions have to be taken into account. Since we distinguish spheres of atomicity and spheres of isolation as separate concepts (meaning that a set of interacting processes may well consist solely of atomic spheres without any isolation enforced by the system), event revocation is much more complicated. The difference can be illustrated by looking at the the *dependency graph* defined by the producer-consumer relationship of events. While this is a tree in active databases, it is a mesh in our case.

The same is true for other database-oriented areas of research, for instance the large body of work on controlled interaction of transactions [Elm92]. All of these models rely on common data as the medium of interaction between transactions (or at least on some notion of shared information) and, because of this, cannot be used in event-based workflow environments. Finally, event-like mechanisms have been proposed in the context of groupware frameworks in [SL95, SS96]. While the general idea, exchanging events between isolated applications, is similar to our approach, the mechanisms we have developed go much further because of their tight integration with exception handling mechanisms and the possibility for recovery.

8 Conclusions

In this paper, we have investigated the problem of inter-process communication in workflow management systems. An event-based mechanism was proposed that has the advantage of integrating event mechanisms into the workflow engine instead of relying on underlying event-based technology. Our language extensions allow to enhance traditional, flow-oriented workflow languages with event mechanisms, giving the workflow designer the option to arbitrarily combine both paradigms. We have especially focussed on the recovery problems existing if interprocess communication is used in transactional workflow environments.

The algorithms we have presented are likely to widen the scope of applicability of workflow techniques by allowing to implement new modes of interaction between otherwise isolated processes. We believe that "cutting windows into the black box" by using the event and exception mechanisms proposed in this paper will lead to better process models since designers do not have to use workarounds like using flat, unstructured process models if interaction between applications is essential. Our approach, as shown in the examples, also provides the means for the integration of heterogeneous workflow tools.

The combination of interprocess communication, event-based exception handling, and atomic spheres delivers a very powerful programming model facilitating the specification and enactment of robust and flexible processes suitable for mission-critical applications. Our algorithms for recovery in the presence of event revocation provide the process designer with the option to choose from a family of recovery strategies, allowing them to gradually adapt the system behavior to the application needs.

Although we have described the mechanisms in the context of OPERA, adapting these ideas to other process support systems should be straightforward since we don't require special properties from the workflow language (the language extensions were applied to the most simple language possible), and the event manager should be embeddable into any system providing sufficient interfaces to access the navigation services.

References

[AGL98] R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. In *EDBT 98*, 1998.

[AHST97a] G. Alonso, C. Hagen, H.-J. Schek, and M. Tresch. Distributed processing over stand-alone systems and applications. In *23rd International Conference on Very Large Databases (VLDB '97)*, Athens, Greece, 1997.

[AHST97b] G. Alonso, C. Hagen, H.-J. Schek, and M. Tresch. Towards a platform for distributed application development. In A. Dogac, L. Kalinichenko, T. Ozsu, and A. Sheth, editors, *1997 NATO Advance Studies Institute (ASI)*, Istanbul, Turkey., August 1997.

[AM97] G. Alonso and C. Mohan. Workflow management: the next generation of distributed processing tools. In Sushil Jajodia and Larry Kerschberg, editors, *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997.

[CCPP95] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual modeling of workflows. In Springer-Verlag, editor, *Proc. of the 14th Object-Oriented and Entity-Relationship Approach Intl. Conf.*, pages 341–354, Goldcoast, Australia, 1995. LNCS.

[CCPP96] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Semantic workflow interoperability. In *Proc. of the 5th Conf. on Extending Database Technology (EDBT)*, pages 443–462, 1996.

[EL96] J. Eder and W. Liebhart. Workflow recovery. In *First IFCIS Intl. Conf. on Cooperative Information Systems (Coop-IS'96)*. IEEE Computer Society Press, June 1996.

[Elm92] A. Elmagarmid. *Transaction Models for Advanced Database Applications*. Morgan-Kaufmann, 1992.

[GT96] A. Geppert and D. Tombros. Event-based distributed workflow execution with EVE. Technical Report Technical Report 96.5, University of Zurich, 1996.

[HA98] C. Hagen and G. Alonso. Flexible exception handling in the OPERA process support system. In *Proc. 18th Intl. Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998.

[Hag97] C. Hagen. Atomarität in Workflow- und Prozessunterstützungssystemen. In *9. Workshop "Grundlagen von Datenbanken"*, Friedrichsbrunn, Germany, May 1997.

[IBM95] IBM. *Managing your workflow*. IBM, March 1995. Document No. SH19-8243-00.

[Koz97] M. A. Kozlowski. *Delphi Research Shows Workflow Market Continues Modest Growth As Technology Proliferates*. The Delphi Group, Boston,MA,USA, 1997. http://www.delphigroup.com.

[LA94] F. Leymann and W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 33(2):326–348, 1994.

[Lam78] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Ley95] F. Leymann. Supporting Business Transactions via Partial Backward Recovery in Workflow Management Systems. In *Datenbanksysteme in Büro, Technik und Wissenschaft*, pages 51–70, 1995.

[PDW$^+$94] N.W. Paton, O. Diaz, M.H. Williams, J. Campin, A. Dinn, and A. Jaime. Dimensions of active behaviour. In N.W.Paton and M.H.Williams, editors, *Proc. 1st Int. Wshp. on Rules In Database Systems*, pages 40–57. Springer-Verlag, 1994.

[SAP97] SAP AG. SAP business workflow. Available through http://www.sap.com, 1997.

[SL95] K. Schwab and H. Ludwig. Ein ereignisbasierter Ansatz zur Integration von Workflow-Management-Systemen mit Groupware-Werkzeugen. In W. Augsburger, H. Ludwig, and K. Schwab, editors, *Koordinationsmethoden und -werkzeuge bei der computergestützten kooperativen Arbeit*. Universität Bamberg, 1995. (In German).

[SS96] K. Schwab and K. Schaller. Synchronisation der Ablaufsteuerung von Workflows und CSCW-Applikationen im PlanKo-Kooperationsmanagementsystem. In S. Jablonski, H. Groiss, R. Kaschek, and W. Liebhart, editors, *Geschäftsprozessmodellierung und Workflowsysteme*, Klagenfurt, Austria, 1996. (In German).

[SST98] H. Schuldt, H.-J. Schek, and M. Tresch. Coordination in CIM: Bringing database functionality to application systems. In *Proc. of the 5th European Concurrent Engineering Conference (ECEC'98)*, Erlangen, Germany, April 1998.

[WC96] J. Widom and S. Ceri, editors. *Active Database Systems*. Morgan Kaufmann Publishers, 1996.

[WMC94] THe Workflow Management Coalition. The workflow reference model. Technical report, Workflow Management Coalition, http://www.aiai.ed.ac.uk/WfMC, 1994.

[ZNBB94] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring relaxed atomicity for flexible transactions in multi-database systems. In *Proc. ACM SIGMOD*, pages 67–78, 1994.