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

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

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 (PSS) allow to define, enact, and monitor the execution of complex sequences of application invocations over heterogeneous, distributed environments. These sequences of invocations are represented in the form of a *process* in which the data and control flow between the different application is specified. Traditionally, a process appears as a black box to other processes, that is, during execution, the state of the process is not visible to the outside world except when using the monitoring capabilities of the PSS, which are designed to follow up execution but are not suitable for programming purposes. As a result, in current commercial products and many research prototypes it is not possible to exchange information between processes while they are executing. In many application areas, this is a severe limitation that interferes with good programming and restricts the usability of the system.

In this paper, we address the problem of incorporating interprocess communication into the PSS engine. Our approach is based on event- and rule-based mechanisms [13, 16] similar to those used in event-based workflow architectures [4, 8]. However, we do not necessarily rely on active database technology to implement our solution. The contribution is twofold. On the one hand, we provide a 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 integrate such functionality in a clean, elegant, and efficient manner. Especially, 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.

The rest of the paper is structured as follows. Section 2 contains a motivation. 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 presents an overview of related work and section 7 concludes the paper.

2 Motivation and Problem Statement

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

2.1 Example

The example we use, Fig. 1, represents a typical administrative environment [14] to handle the delivery of products by a retail company. The process shown captures all steps of the procedure, from the initial placement of the order by a customer to the final payment. The process has been divided into two sub-processes, *warehouse administration* and *order handling*. The warehouse subprocess, shown in the lower part of the figure, takes care of stock control: if the item is in stock, it is ordered from the warehouse. Otherwise, it is ordered from the manufacturer. Once the item is ready, an invoice is produced for the amount purchased. The order handling process, shown in the upper part of the figure, deals with the purchase order itself. It includes activities such as receiving the order, triggering the warehouse subprocess, confirmation of the order to the customer, as

* 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).

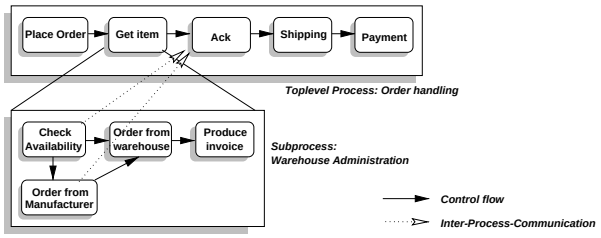

Figure 1: Example process

well as the shipping and payment procedures.

2.2 The need for interprocess communication

This example can be modeled in several ways. However, in existing systems and once the designer decides to have two subprocesses, each subprocess necessarily appears as a "black box". That is, the order handling subprocess has to wait until the warehouse subprocess finishes before it can resume execution. This greatly decreases the degree of parallelism: for instance, the acknowledgment to the customer could be sent as soon as it is known that the product is in stock, thereby giving the customer an impression of faster service. Similarly, the shipping could be initiated as soon as it 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.

In the example, we work with subprocesses but the same can be said of any other example in which behavior is encapsulated within a process. One way to bypass this limitation is, of course, to write the entire process as one single flat process. This would permit intertask communication but the solution is both inelegant and impractical since it makes processes unreadable and difficult to maintain. We see processes as a very valuable resource: they need to be defined, designed, deployed, and maintained, much like any other software application. Structuring the process properly will certainly help with these tasks. On the other hand, this structuring should not result in a loss of expressiveness. Hence, the question is how to allow processes to intercommunicate while still preserving the necessary modularity.

2.3 Event based interprocess communication

To address this issue, we implement interprocess communication using *events*. Events are typed and parameterized signals raised by a running task to inform other processes of certain situations reached or produced during its execution. In our example, the warehouse 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 mechanisms. 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. In addition, 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. Finally, 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 [10].

Allowing processes to communicate with each other has obvious advantages. It also has a clear cost. In particular, processes must at all times guarantee consistency. This is not so simple when the processes interact with each other.

When processes are isolated, consistency can be guaranteed using the notion of *atomic processes.* Unfortunately, providing atomicity when processes communicate with each other is a more complex matter. A simple rollback, as it is done in database systems, is normally not possible. Instead, *compensation mechanisms* are needed to logically undo the effects of activities [6, 12, 10].

3 OPERA - a generic process support kernel

The interprocess communication mechanism described in this paper has been implemented as part of OPERA, a generic process support system kernel [3] which provides the necessary infrastructure for enactment and persistence.

3.1 Modeling language

OPERA supports a 3 level hierarchy of process representations: application specific, internal, and low level representation [3]. In here, we will use an application-specific model, the OPERA graphical workflow language (OGWL), to describe language extensions and capture new semantics. OGWL follows similar languages used in commercial systems [12] and is to a large degree conformant with the standard of the workflow management coalition [15].

In OGWL, a process 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. 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 connector 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. In a similar fashion, *data flow* is specified through *data connectors* linking activities and indicating the flow of information between them. Each task has an *input data structure* describing its input parameters and an *output data structure* to make return values accessible.

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 [12]. 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.

3.2 Atomic spheres

OPERA integrates transaction concepts directly into the process support system. These concepts have been described in detail in [3, 10, 9]. In general, OPERA uses the notion of *spheres* to bracket 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) [2]. For the purpose of this paper, the most relevant spheres are spheres of atomicity.

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 ([17]). 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.

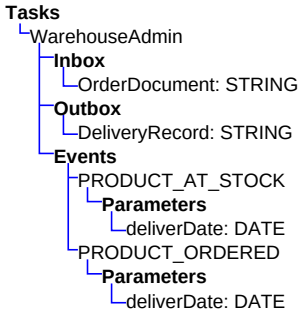

Figure 2: Declaration of the warehouse administration task, including events

4 Inter-Process Communication

This section describes the implementation of the interprocess communication mechanism.

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. 2). 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 warehouse 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.

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 events is limited to the block or process enclosing a task (like 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.

4.4 Example

Fig. 3 and Fig. 4 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 (full definition of all tasks, certain aspects of data flow). Fig. 3 shows the intra-process communication mechanism. The GetItem task is now defined according to Fig. 2, declaring two event types. We have added two event based control connectors (the broken lines) that react to the occurrence of the respective events in the warehouse subprocess. 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, the warehouse subprocess 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. 4 illustrates the inter-process communication mechanism. It shows the same application, this time modeled using two parallel subprocesses communicating through events. Upon start of the parent, both subprocesses are

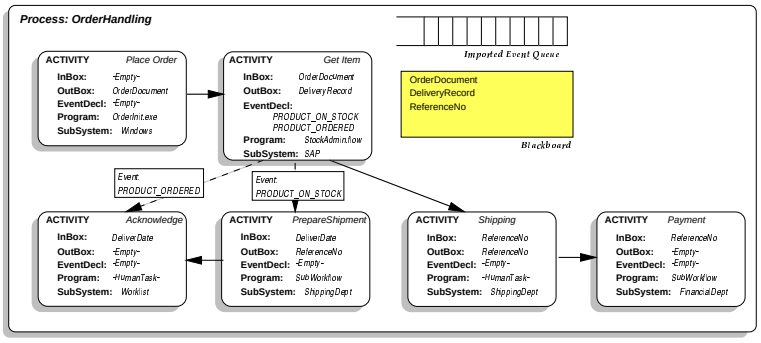

Figure 3: The initial example, now modeled using events for intra-process communication

started, and the *PlaceOrder* activity is invoked (it is the only activity marked as *StartActivity*, enforcing its immediate begin). At its end, 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 subprocess subscribes to the events declared by the other, and that activities can access all events through the *imported event queue* using ECCs.

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 systems [10]. 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 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 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 where an event that was intercepted by a task or process is undone due to the abort of its 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. 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, requiring compensation to "post mortem" undo the effects of the consumer. Let $DEP_{EV}(t, T)$ be a shortcut for $\{T_{CONS} | (T, T_{CONS}) \in DEP_{EV}\}$.

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. 5. If a task T is aborted, the system determines $DEP_{EV}(t, T)$. To this end, a registry of all currently exist-

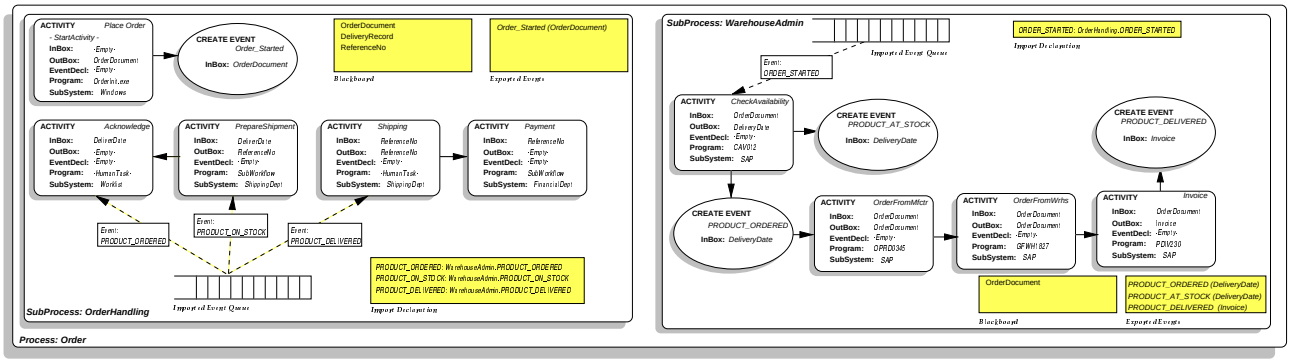

Figure 4: Two subprocesses using inter-process communication

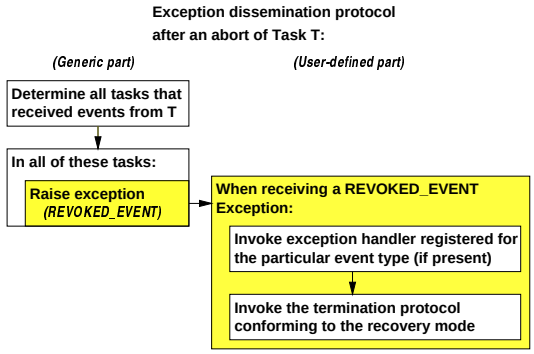

Figure 5: Exception dissemination algorithm

ing consumer-producer relationships is maintained by the system in a persistent table. 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 the importance of the order in which the protocol is executed. 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 below.

5.4 Partial rollback algorithm

Between all tasks in the system, a causal dependency [11] 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 [1]. 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. 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 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 Q_{TODO}: An exception is sent to the task at the head of Q_{TODO}, which is resolved through the exception handling mechanism. If the task has the PARTIAL_ROLLBACK option set, its consumers are appended to 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. 6 illustrates the recovery algorithms, with three processes communicating through events. At the top, we show the current state of the processes at the time a failure occurs in Process1, leading to its abort. At this point, the first two 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 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.

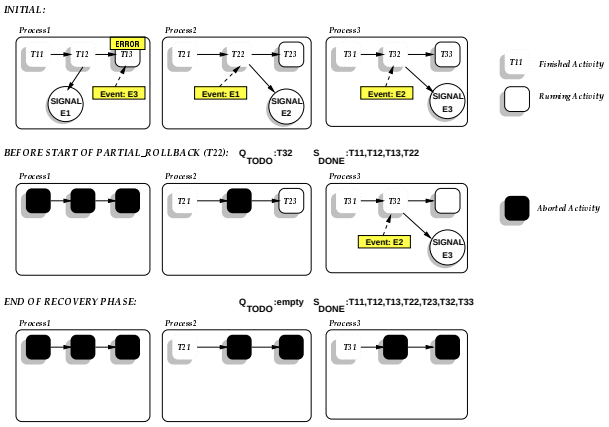

Figure 6: Example for the event revocation algorithm

5.5 Recovery strategies

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 several recovery strategies like IGNORE (ignore the failure), NO_ABORT (provide an exception handler invoking alternative activities), ABORT (abort the affected process), CONTINGENCY (invoke contingency operations *replacing* the invalidated task), or ROLLBACK (rollback to the state before the event, and continue work from there).

6 Related work

[5] provides a classification of possible interactions if processes access common data. A *workflow integration* ap-

proach is proposed that aims at identifying and removing interferences between concurrent workflows, but without discussing event-based process interaction, exception handling, or recovery.

Contrary to what might be expected, the work done in the context of active database management systems (ADBMS) [13, 16] 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 [7]. 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.

7 Conclusions

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. 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.

References

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

[2] G. Alonso. Processes + transactions = distributed applications. In *High Performance Transaction Processing (HPTS) Workshop*, Asilomar, California, Sept. 1997.

[3] 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.

[4] 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.

[5] 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.

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

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

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

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

[10] C. Hagen and G. Alonso. Flexible exception handling in the OPERA process support system. In *Proc. ICDCS 1998*, Amsterdam, The Netherlands, May 1998.

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

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

[13] N. Paton, O. Diaz, M. Williams, J. Campin, A. Dinn, and A. Jaime. Dimensions of active behaviour. In N.W.Paton and M.H.Williams, editors, *1st Int. Wkshp. on Rules In Database Systems*. Springer-Verlag, 1994.

[14] 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.

[15] The Workflow Management Coalition. Interface 1: Process Definition Interchange Process Model. Document No. WfMC TC-1016-P, Nov. 1998. Available on the WWW: http://www.wfmc.org.

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

[17] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring relaxed atomicity for flexible transactions in multidatabase systems. In *Proc. ACM SIGMOD*, 1994.