A Discussion on Approaches to Handling Exceptions in Workflows

Fabio Casati
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Via Ponzio 34/5, I-20133 Milano, Italy

e-mail: casati@elet.polimi.it

Abstract
Workflow technology can be fully exploited when predictable and repetitive processes are executed. Unfortunately, many processes are faced with the need of managing exceptional situations that may arise during their execution, and possibly even more exceptions can occur when the process execution is supported by a WFMS. In this paper, we analyze the characteristics of the different types of exceptions that may affect the execution of a workflow, and present some approaches to their handling. Finally, we introduce the exception handling techniques that we have developed within the WIDE workflow project.

1. Introduction
Workflow Management Systems (WFMSs) are software systems supporting the modeling, analysis, and enactment of business processes. A workflow (WF) is basically defined by a set of tasks or subprocesses, and by a flow graph that defines the order of task execution. The WF engine then executes a process by scheduling tasks when they are ready for execution and by assigning them to the proper executing agents.

Workflow technology can be fully exploited when predictable and repetitive processes are executed. Unfortunately, many processes are faced with the need of managing exceptional situations that may arise during their execution, and possibly even more exceptions can occur when the process execution is supported by a WFMS. Exceptional situations may be caused by system (hardware or software) failures, or may by related to the semantics of the business process, such as when a deadline for a proposal presentation expires or when a customer cancels a travel reservation request.

Although research in workflow management has been very active for several years, and the need for modeling exceptions in information systems has been widely recognized (see, e.g., [Borgida85, Saastamoinen95]), only recently the workflow community has tackled the problem of exception handling. One of the first contributions came from Eder and Liebhart [Eder95], who analyzed the different types of exceptions that may occur during workflow execution, and provided a classification of such exceptions. They divided exceptional situations into basic failures, corresponding to failures at the system level (e.g., DBMS, operating system, or network failure), application failures, corresponding to failures of the applications invoked by the WFMS in order to execute a given task, expected exceptions, corresponding to predictable deviations from the normal behavior of a process, and unexpected exceptions corresponding to inconsistencies between the business process in the real world and its corresponding WF description.

Currently, both WF managers and research prototypes provide little support for exception management. In particular, while some support for handling basic and application failures is provided, mainly by relying on the recovery capability of the (database) environment on the top of which the WFMS is executed, systems are typically unable to effectively manage expected and unexpected exceptions, that are peculiar of workflow management, and require the development of ad-hoc methods and tools in order to be properly managed.
In this paper, we analyze the four classes of exceptions identified by Eder and Liebhart; for each class, we discuss the characteristics of the exceptions and the techniques that can be adopted for their management. Finally, we present the WIDE approach to exception management. WIDE is an ESPRIT project, started in 1995, with the aim of developing a commercial strength workflow management system with advanced features for managing exceptions and transactions. Due to space limitations, we will only sketch the solutions adopted in WIDE, and we will refer the reader to the appropriate WIDE technical report or publication for details.

In the paper, we will focus our attention mainly on expected and unexpected exceptions, both because these are peculiar of workflow management and because they have received little attention up to now.

2. Failures

Basic and application failures are concerned with failures of the WFMS, of the WFMS invoked application, or of the environment in which these applications are executed. Basic failures correspond to failures of the WFMS or of its underlying platform, such as hardware failures, network failures, or failures of the DBMS supporting the WFMS; application failures are instead malfunctions of the WFMS invoked applications: the external application may keep running without returning any values to the WF engine, or it may return an error code, such as out of memory or system unreachable. These kind of exceptional situations are not specific of business processes and workflow management; approaches to failure handling have in fact been developed in several different contexts, particularly in the area of transaction processing. Therefore, WFMSs may (and in facto do) handle failures by relying on existing concepts and technology. For instance, basic failures are handled at the system level, by relying on the capability of the underlying DBMS to maintain a persistent and consistent state, thus supporting forward recovery.

However, only a fraction of the operations performed by the WF engine is enclosed within transaction boundaries, and often logically related actions are not executed atomically. In particular, the communication between the WFMS and WF client applications (e.g., the scheduling of a new tasks and its assignment to one or more agents) and the registration of this operations in the database are not atomic operations. Analogously, notifications of completion of tasks executions coming from the clients and the corresponding database transaction recording the event at the server site are not executed as atomic operations. Furthermore, tasks already dispatched by the WF engine could be started and executed while the engine is down. Thus, at the reboot of the system, the engine's view of the state of a WF execution might not be accurate, and therefore the engine has to reconstruct the WF state by communicating with the WF clients. An effective management of basic failures can be achieved by integrating the WFMS with transactional queues; queues handle the communication between server and clients, and takes care of delivering messages when needed. Furthermore, they are transactional resources, so that operations on the queue are made permanent only if the transaction that performs them commits, otherwise they are rolled back [Bernstein97].

Application failures are instead relevant to WF management since they results in task failures: since the application could not be completed, the task in the context of which the application has been invoked cannot be completed successfully, i.e., it cannot meet its business objective. Although managing the modifications required by the applications in order to execute correctly is outside the workflow domain, managing the business failures that results from the application failures is indeed a business problem.

A generic approach to handling task failures involves the integration of workflow models with advanced transaction models [Worah97]. In fact, if the WF model provides "traditional" transaction capabilities such as the partial and global rollback of a process, task failures can be handled by rolling back the process execution until a decision (split) point in process is reached from which
forward execution can be resumed along a different path. This model is supported by several systems; for instance, ConTracts provide an execution and failure model for workflow applications [Reuter97]; a ConTract is a transaction composed of steps. Isolation between steps is relaxed, so that the result of completed steps are visible to other steps; in order to guarantee semantic atomicity, each step is associated with a compensating step, that (semantically) undoes its effect. When a step is unable to fulfill its goal, backward recovery is performed by compensating completed steps, typically in the reverse order of their forward execution, up to a point from which forward execution can be resumed along a different path. WAMO [Eder95, Eder98] and Crew [Kamath98] extends this approach by providing more flexible and expressive models, more suitable for workflow applications. Commercial systems do not typically provide this kind of functionality. However, the Exotica project [Alonso94,Alonso96] has developed a tool that provides the WF designers of IBM FlowMark with an extended WF model, allowing the implementation of advanced transaction models such as sagas and flexible transactions. Specifications in the extended model are then translated into FDL (FlowMark Definition Language) by properly inserting additional "compensating" paths after each task, to be conditionally executed upon a task failure (captured by means of the task return code).

The WF model of WIDE includes a two-layer transaction model that provides the required flexibility in order to handle task failures. At the global (workflow) level, a WF is modeled as set of business transactions (subprocesses), that are atomic units from a transactional perspective, and that are executed according to a nested transaction model. Isolation is not enforced between business transactions. A WF execution can be (partially or globally) rolled back by aborting running business transactions and by compensating committed ones, up to a savepoint from which forward execution can be resumed. The reader is referred to [Grefen97] for details.

Finally, it should be noted that transactional approaches offer in general extreme and expensive solutions in terms of lost work, and therefore some task failures may deserve an ad-hoc handling; in this case, task failures are in fact handled as expected exceptions, and can be modeled by applying the constructs and techniques described in the next section.

### 3. Expected Exceptions

Expected exceptions are predictable deviations from the normal behavior of the process. Examples of expected exceptions are:

- in a travel reservation process, the customer cancels the travel reservation request;
- in a proposal presentation process, the deadline for the presentation has expired;
- in a car rental process, an accident occurs to a rented car, which is not available for subsequent rentals.

Unlike basic and application failures, expected exceptions are strictly related to the WF domain: they are part of the semantics of the process, and it should be possible to model them within the process, although they are not part of its "normal" behavior.

By working on expected exceptions, also with the support of the users in the WIDE project, we have analyzed the characteristics of such exceptions, in order to devise the most appropriate modeling paradigm for their management. We have observed that expected exceptions can be broadly divided into four classes, according to the event that causes them:

*Workflow exceptions* are raised in correspondence of the start or completion of tasks and WF instances, and are therefore synchronous with the progression of the workflow.

*Data exceptions* are raised by modifications to WF relevant data.
Temporal exceptions are raised at the occurrence of a given timestamp (e.g., the deadline for the proposal presentation), periodically (e.g., every night at 7pm), or as a defined interval has elapsed since a reference event (e.g., 20 minutes after the activation of the task).

Finally, External exceptions are activated by external events, explicitly notified to the WF engine by agents or external applications (e.g., the customer cancels his/her travel reservation).

The above examples and observations show that although exceptions may be synchronous with the progression of the workflow, most often they are asynchronous. In fact, the only synchronous exceptions are workflow exceptions, and we experienced that they are only a fraction of the expected exceptions that need to be modeled. External and temporal exceptions are in general asynchronous, while data exceptions occur synchronously with task completions (if the modifications to WF relevant data are made visible only at the end of the tasks that modify them), but are not associated to a particular task, since different tasks in the same WF may modify the same data.

We also observed that although managing an exception may indeed require the activation of a task, quite often the appropriate reaction involves sending notification messages to agents (such as notifying to the task executor or to the workflow responsible that a deadline is close to its expiration), reassigning a task to a different agent, or rollbacking the execution of a process.

The typical WF process model, offered by the large majority of the WFMSs, allows to specify a process by means of a graph of tasks. This model is indeed very good for specifying the normal behavior of a process, described by the sequencing of tasks; however, since exceptions may not be raised by task completions, and often are not managed by activating tasks, they cannot be suitably represented within the graph of tasks. In general, only workflow exceptions can be modeled within the graph, although good design criteria recommend to separate the description of the normal and exceptional behavior. A detailed discussion on the drawbacks of modeling expected exceptions with a basic WF model and with the WF model provided by IBM FlowMark is provided in [WIDE3027].

A modeling paradigm that seems instead promising for representing expected exceptions, due to the above mentioned characteristics, is a rule-based approach. An exception can be modeled by an Event-Condition-Action (ECA) rule, where the event describes the occurrence of a potentially exceptional situation, the condition verifies that the occurred event actually corresponds to an exceptional situation that must be managed, while the action reacts the exception. Thus, the event part in the rule language should enable the modeling of the four different kinds of events that may trigger an exception; the condition part should enable the evaluation of the state of tasks and processes as well as of possible event parameters, while the action part should allow the specification of reactions such as notifications to agents, reassignment of tasks, rollback of process executions, activation of tasks or processes, and so on.

Following these guidelines, at Politecnico di Milano we have defined a rule language for defining expected exceptions, called Chimera-Exc, and we have developed a system capable of executing Chimera-Exc specifications asynchronously with respect to the WIDE WF engine, that support instead the normal execution of processes. Chimera-Exc is derived from the object-oriented database language Chimera; in Chimera-Exc rules, triggering events belong to one of the four classes mentioned above (data, temporal, external, or workflow); the condition is a predicate over system and WF relevant data whose evaluation determines whether the action part should be executed or not, and may in addition return some bindings to be passed to the action part for targeting the reaction over specific objects; finally, the actions allow to send notifications to selected agents, to start, suspend or terminate the execution of tasks and processes, to reassign tasks to different agents, or to rollback process executions. Figure 1 shows a simple (and pleasant) example of a Chimera-Exc rule that at the end of the working day (7pm in the example) notifies the executors of running tasks in the travel reservation process that they should suspend their work. Both the language and the system supporting the execution of Chimera-Exc rules are described in the technical report.

http://ccs.mit.edu/klein/cscw98/paper26/
Presently, commercial WFMSs provide little support for expected exceptions; in general, only synchronous exceptions can be captured by the model, and can be handled by defining "exceptional" execution paths in the control flow, often resulting in specifications that are complex and difficult to understand. For instance, widespread products such as IBM FlowMark [Flowmark] or Hewlett Packard AdminFlow [Adminflow] can only manage deadlines for tasks and processes, by sending messages to agents or by executing an "exceptional" part of the flow as a deadline expires. A few systems, such as COSA [Cosa], StaffWare [Staffware], and InConcert [MCS93], offer some modeling features for capturing and reacting to events, although with less functionality than that provided by Chimera-Exc. They basically allow the definition of event-action rules (or of event steps in StaffWare) that can be triggered by external events, WF events, or by deadline expirations, and can react to the triggering event by activating tasks or (sub)processes, or by sending messages to agents.

An interesting approach to exception handling is provided by the OPERA research prototype [Hagen98]. OPERA allows the definition of exceptions triggered by data, workflow, and external events, and then models reaction to exceptions with the same formalisms defined for the specifying normal processes, i.e., tasks and control flows. Furthermore, OPERA allows to handle exceptions in a modular way: as an exception is detected, the process is suspended and the control is transferred to the exception handler of the subprocess in which the exception occurred. The exception handler can then propagate the exception to higher level handlers (e.g., to the handler of the parent process). OPERA and WIDE have similar exception handling capabilities, although WIDE supports a larger set of events and actions that simplify exception management, and allows the specification of conditions that determine whether the triggering event actually corresponds to an exception that must be managed. WIDE lacks instead the capability of handling exceptions in a modular way, and the inclusion of this capability in our rule-based model is part of our future research.

4. Unexpected Exceptions

Unexpected exceptions corresponds to inconsistencies between a business process in the real world and its workflow representation. For instance, assume that a new agreement between Italy and the US requires Italian tourists traveling to the US to previously request and obtain a visa from the US consulate. If the travel reservation WF (Figure 2-a) has not yet been modified according to the new law, then its execution does not lead to the successful completion of the business process.
Unexpected exceptions occur quite frequently, both because the WF modeling phase is complex and often not accurate enough, and because the characteristics and requirements of the business processes may change over time, due for instance to new laws, new business objectives, or technological innovations.

Since these exceptions have not been foreseen and modeled, both their detection and handling requires human intervention [Heinl98]. As WF agents detect an unexpected exception, they are faced with the following alternatives:

Figure 2 - The Travel Reservation Workflow. (a): initial version; (b): modified version; (c): ad-hoc version for managing instances that cannot be migrated to the correct version.
1. "Fool" the system: the WF agent performs actions outside the control of the WFMS, in a way that by proceeding with the execution of the WF, the business process can be actually completed as required. For instance, in the travel reservation process, the clerk asks for the visa and waits until the visa has been granted before proceeding with the workflow; the system is unaware of this.

2. Modify the running workflow instance: the WF instance is modified, so that its execution leads to the successful completion of the business process. For instance, in the travel reservation process, the WF agent modifies the running WF in order to include a task for requesting the visa. Note that, with this alternative, only one instance is affected by the modifications: future instantiations of the travel reservation workflow will be executed according to the old workflow definition.

3. Modify the workflow definition: the modifications are applied not only to the running instance for which the exception has been detected, but to the workflow definition itself, so that both running instances and future instantiations of the same workflow will be executed according to the modified definition. This approach leads to a number of interesting issues, briefly described in the following, concerning the management of running instances when the corresponding WF definition is modified.

The first alternative represents the fastest and simpler way to handle the exception. The drawbacks are that it does not prevent further occurrences of the same exception in other instances of the same workflow, and that actions executed outside the control of the WFMS cannot be supported and monitored by the system. Thus, this alternative can be adopted if we do not expect the same exception to occur again for other WF instances and if we do not require the support of the WFMS for the execution of the corrective tasks or flows.

The second alternative allows the entire business process to be supported and monitored by the WFMS. However, this solution implies the overhead of appropriately modifying the specifications followed by the WF instance (also involving access rights and security issues) and, as with the previous approach, does not prevent further occurrences of the same exceptions. This alternative can be adopted if the system support is needed for executing and monitoring the corrective activities, and if the same exception is not expected to occur again for other instances of the same workflow.

Finally, this solution is applicable only if the system allows dynamic modifications to running WF instances.

The third alternative is the most appealing, since it prevents further occurrences of the same exception; in fact, all instances will be executed according to the modified definition. For instance, in the corrected travel reservation workflow, all travel reservations for Italian citizens traveling to the US will include a task for requesting the visa to the US consulate (Figure 2-b). An interesting problem with this alternative is how to manage running instances (including the one for which the exception has been raised) when the corresponding WF definition is modified. This issue was first discussed in a paper by Ellis et al. [Ellis95], and then addressed in several other papers (e.g., [Casati98a, Dadam97, Liu98]). In the context of the WIDE project [Casati98a], we have defined a set of operations for dynamically and incrementally modifying a WF definition, and we have defined a taxonomy of approaches for managing running instances upon a modification of the corresponding WF definition. In particular, we have defined formal criteria, based on the modifications applied to the WF definition and on the state of the WF instance, that define whether a running instance can migrate to the modified WF definition or it should instead migrate to a temporary, ad-hoc defined WF. Intuitively, in the travel reservation process, all instances that are not concerned with Italian citizens going to the US or that are in their early stages (i.e., task record customer data has not been completed yet) can migrate to follow the modified WF definition shown in Figure 2-b, while the other instances will have to be handled ad-hoc, for instance by migrating to the WF shown in Figure 2-c, that still allows to achieve the goals of the process, although with a sub-optimal process. Due to space limitations, we do not discuss further this interesting topic. The interested reader is referred to the cited papers [Casati98a, Dadam97, Ellis95, Liu98].
5. Concluding Remarks

In this paper we have presented some approaches in order to handle the different types of exceptions that may occur in a workflow execution. Starting from classification of exceptions proposed by Eder and Liebhart, we have discussed the characteristics of exceptions in each class, and we have presented possible approaches to their management. We have also briefly described current approaches to exception handling in commercial systems and research prototypes, and we have presented how exceptions are managed in the WIDE workflow project.

The future work of the WIDE team at Politecnico di Milano will focus on expected exceptions. We plan to extend our system in order to manage exceptions in a modular way, by first activating a local, process specific exception handler, and by then allowing the propagation of the exception to the parent process. We will also address the issue of reusing the expected exception handler developed within the WIDE project for other commercial workflow systems, by interacting with the API offered by the WF engines. We believe in fact that some features of our component, and in particular the capability of capturing and handling temporal and external events, could improve the functionality offered by current commercial WFMSs, that often do not effectively manage these types of events.

Note: Research presented in this paper has been sponsored by the WIDE ESPRIT project n.20280.

References


