Flexible Exception Handling in Process Support Systems *

Technical Report No. 290
ETH Zürich, Department of Computer Science

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

February 5, 1998

Exceptions are one of the most pervasive problems in process support systems. In installations expected to handle a large number of processes, having exceptions is bound to be a normal occurrence. Any programming tool intended for large, complex applications has to face this problem. However, current process support systems, despite their orientation towards complex, distributed, and heterogeneous applications, provide almost no support for exception handling. This paper shows how flexible mechanisms for failure handling are incorporated into the OPERA process support system using a combination of programming language concepts and transaction processing techniques. The resulting mechanisms allow the construction of fault-tolerant workflow processes in a transparent and flexible way while ensuring reusability of workflow components.

1 Introduction

A *process* can be defined as a sequence of program invocations and data exchanges between distributed and heterogeneous stand-alone systems. Workflow management systems (WFMS) provide support for business processes [AS96, GHS95, Hsu95, Hsu93, JB96, She96, WfM96], while a process support system (PSS) generalizes this idea to arbitrary types of processes [AM97, AHST97a, AHST97b]. In this sense, a process support system appears as a tool for "programming in the large" over heterogeneous and distributed environments [AM97].

Programming tools intended for large, complex applications incorporate *exception handling* mechanisms to separate the failure semantics from the "normal" program logic and thus facilitate the design of readable, comprehensible programs [Par72, Goo75, YB85, vZBC96]. Exception handling mechanisms are also incorporated into frameworks for distributed applications like CORBA and into operating systems like Windows NT. Despite the similarities between process support systems and programming environments [AHST97a, AHST97b] there is little support for exception handling in

*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).

current process systems. Any possible exception must be encoded in the process. All exceptions that are not hard-wired into the process result in either aborting the process or require human intervention. Since processes tend to be long (days, even months), involve a considerable number of resources and personnel, and since there might be a very large number of processes, neither aborting nor human intervention are satisfactory solutions [AS96, KR96, GHS95].

Current efforts towards standardization and modularization of workflow management systems are a further motivation for this work. The Workflow Management Coalition is in the process of standardizing interfaces for application and subprocess invocation across different workflow systems. Furthermore, there are ongoing efforts towards the standardization of business objects, predefined subworkflows that can be purchased and integrated into larger business processes. We believe that the incorporation of exception declarations into these standard interfaces could be an important contribution since it would allow to develop fault tolerant workflows out of pre-existing building blocks.

To address these issues, the paper describes the exception handling mechanisms implemented in the Opera process support system [AHST97a, AHST97b]. Opera aims at generalizing process support concepts to application domains other than business processes. It is an adaptable kernel for process support which can be tailored to different environments and purposes, ranging from job control in distributed, heterogeneous systems to experiment management in scientific applications [AH97]. The Opera approach to exception handling borrows its main ideas from a combination of programming language concepts and transaction processing techniques, adapting them to the special characteristics of process support systems. To our knowledge Opera is the first system to integrate language primitives for exception handling into workflow management systems. Previous approaches were limited to achieving atomicity but did not take into account the need to react differently to different types of failures and the need of a tight integration of failure handling and modeling language. A further contribution of the paper is the integration of programming language concepts and transactional ideas. We show how the semantics defined through the language constructs are enforced through the usage of an execution model based on advanced transaction models.

The paper is organized as follows. The next section provides an example and motivates the ideas proposed in the paper. Section 3 introduces the exception handling concepts in Opera. Section 4 discusses the relation between workflow recovery and transaction models. Section 5 describes the integration of exceptions signaled by external applications. Section 6 proposes several language primitives to integrate exception handling and graphical languages. Section 7 contains an example and discusses the described approach. Section 8 briefly discusses related work and Section 9 concludes the paper.

2 Motivation and Example

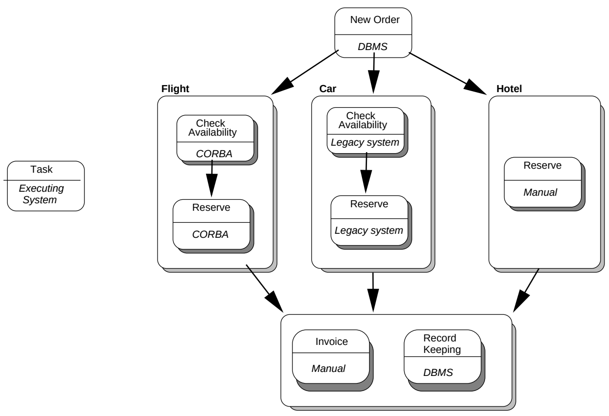

Figure 1: A workflow process

As a running example for the rest of the paper, consider a process incorporating the reservation of various flights, rental cars and accommodations as well as the final sending of the documents and invoice to the customer and the storage of the result in the travel agency's internal database (Figure 1). The programs and services incorporated in the process are executed by different autonomous systems: Flight reservation is done through a CORBA gateway to a booking system. Sending the documents and invoice as well as reserving a hotel are manual task to be handled by the travel agency's personnel. Record keeping in the local database takes place via a TP monitor, and the reservation of a rental car is done through a legacy system. The possible failures can be classified into several categories: Program failures, design and communication errors, and semantic failures.

Program failures lead to the unsuccessful abort of an external application. For instance, the flight reservation step can fail because no seats (or no seats in the requested category) are available any more; manual tasks like the *invoice* task usually have an associated time limit which may be violated; the *record keeping* task may fail because of a transaction abort. A fault tolerant process has to provide handling strategies for all of these failures. In the case of unavailable seats the solution may be to ask the customer to select a different flight, to accept a different category, or to cancel the whole reservation process. In the case of unmet temporal constraints the task can either automatically be dispatched to a different employee or a supervisor can be notified. The transaction failure may be solvable by simply retrying execution if the reason for the abort was a deadlock. Note that often the external application provides some level of fault-tolerance. A database application, or the database system itself, may automatically reschedule an aborted transaction. While this simplifies the work to be done by the PSS, it does not solve the problem of more complex failure handling strategies. For instance, using a train reservation as a contingency plan for the flight/car reservations needs a combination of backward and forward recovery (we assume here that no renatl car is needed if a train is booked since the railway station is central enough) : If a car has already been reserved, but no flight is available, recovery requires that the reservation be canceled before the train is booked.

Design and system failures, like incorrect parameters for program invocation or impossible program executions because of configuration changes, must also be considered. If the process programmer has erroneously provided an incorrect program name, object id, or host address, or if an application has been moved from one host to another, the invocation will fail. Such failures must be captured by the process model in order to allow the specification of flexible reactions. The same is true for *communication failures* that may prevent the invocation of a program. The difference between the two is that design errors usually lead to aborting the process, while communication failures may be resolved by re-trying execution at a later point in time or by choosing a different host. *Semantic failures* are the most interesting kind of exceptions. The execution of a program without errors does not always mean that it was successful, since there may be additional constraints. In the example given, the *checkAvailability* service returns successfully even if no available seat has been found. This case is, however, an exception since without available seats no reservation can be made. It is up to the process code to detect this and invoke appropriate measures. Thus the process system needs mechanisms to define what is regarded as semantic exception in order to incorporate them into the general exception handling scheme. We will discuss these issues in section 5.

As a further motivation of the need for structured exception handling, Figure 2 shows a slightly simplified implementation of the example using FlowMark [LA94], including recovery mechanisms like compensation (the Cancel_Flight activities) and contingency tasks (*Hotel2* is a replacement for *Hotel1*, *ReserveTrain* replaces *BookFlight* and *RentCar*) as well as partial rollback (compensation of *BookFlight* if no car is available and subsequent booking of train). Since FlowMark, like most WFMS, provides no mechanisms for structured exception handling, the interleaving of the original tasks with the recovery steps makes the fault tolerant version very complex and the original process logic hardly recognizable. The complexity explosion in this very small example points out the drawbacks of including exception handling as part of the normal process specification. Mixing business logic and exception handling logic makes it difficult to keep track of both, complicating the verification of processes as well as later modifications. Moreover, such an approach makes it almost impossible to reuse components since they will lack meaning once out of the context for which they were originally designed. One of the key features of process systems is the ability to reuse subprocesses, very much in the same way that libraries are used in programming languages. This can only be accomplished if there is some form of system support for exception handling that allows to separate it from the normal code. The main objective of the OPERA approach to exception handling is thus the augmentation of its process modeling languages in an appropriate way.

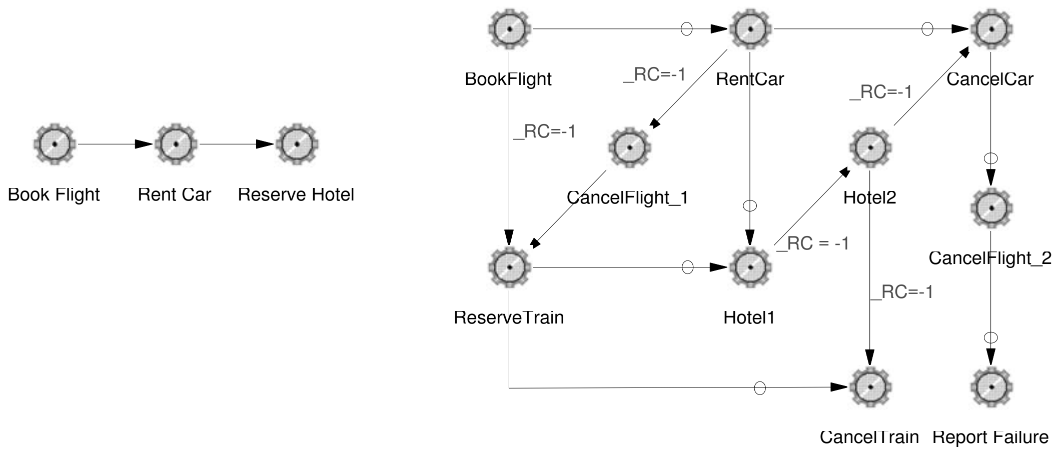

Figure 2: Handling exceptions by programming them in the control flow (FlowMark notation)

In addition to the problem of modeling exception handling strategies in a comprehensible way, PSS have to capture failure signals returned by external applications. There are several ways in which these exceptions are signaled. The first one uses explicit exceptions, as illustrated in Figure 3, which shows an interface description in CORBA's interface definition language (IDL). CORBA services communicate failures to their callers through *exceptions* that are returned instead of the defined return values. In our example, two potential exceptions are declared: *invalidData* and *noSeats*. The second approach for communicating failures is the more traditional one of using return codes, usually some integer representing the failure. In the example, the record keeping activity (implemented in Encina, for example) signals failures this way. A third important signaling method is the use of explicit failure channels like they are used in persistent queuing systems or queue-oriented TP monitors like Tuxedo. The process management system must translate these heterogeneous failure signals into an internal format that allows to treat them uniformly.

3 Exception handling in OPERA

3.1 Basic mechanisms

The exception mechanism used in OPERA is based on programming language concepts proposed by Goodenough [Goo75] and later adopted in many programming languages, including CommonLisp [Ste90], Standard ML [Pau91], C++ [Str91], and Java [Fla96]. These constructs are also an integral part of communication standards for distributed systems like CORBA [Obj92], and exception handling has even been integrated into Windows NT [Cus93], where it is used to handle system and user-defined exceptions in an uniform way.

An exception is an unusual event, erroneous or not, that is detectable either by hardware or software and that may require special processing [Seb96]. The overall goal of exception handling is to give the programmer means to adapt the behavior of operations, allowing them to flexibly react to exceptions

```
interface flightSystem {
  exception invalidData {string reason;};
  exception noSeats {string reason;};

  int checkAvailability (in date depDate, in time depTime,
                         in string depCity, in string destCity,
                         out string category, out string flightNumber,
                         out int noOfSeats)
                         raises (invalidData);

  int bookFlight (in string flightNumber, in string category,
                         in string number, out string code)
                         raises (invalidData,noSeats);
};
```

Figure 3: Sample IDL definition including exception declarations for flight reservation tasks in the example process

of various kinds, while preserving information hiding and autonomy. This is achieved by separating exception *detection* from exception *handling* in nested process structures.

As in other process support systems [CD96, CD97, LA94], a workflow process in OPERA has a nested structure that can be represented by a tree with different tasks (processes, blocks, or activities) as its nodes. The set of child nodes of a task T_i is defined by the subtasks that are invoked inside T_i. Each task has a clearly defined signature that specifies its call parameters and return values. Information hiding demands that only the signature has to be known in order to invoke a task. OPERA's exception handling mechanism is based on the principle that, in case of failures, a child task T_{ik} stops execution and returns an *exception* instead of proper return values. Exceptions are typed data structures that can contain information about the failure context. A task returning exceptions is thus polymorphic: If it is executed successfully, it returns data conforming to its signature, but if it encounters an unusual event, it returns an exception with a different structure. If the parent has defined an *exception handler* (an arbitrary subprocess) for the exception returned by the child (the *signaler*), then when the exception is signaled, control is passed to the handler which contains the necessary steps for failure handling. If no handler is defined by the programmer, then a *default handler* is provided by the system that aborts the parent.

The approach allows modular design, since the programmer of a procedure must only be concerned with exception detection (performed by the invoked operation), while exception handling, which may be context-dependent, is left to the invoker of the procedure. Flexibility is further improved by giving the exception handler control over whether the signaler can continue: The handler has the possibility to either *abort* the signaler or to *resume* its execution after it has dealt with the exception. Resuming execution is used in those cases in which the exception was raised because of invalid parameters and in which exception handling incorporates querying a user for different data. Furthermore, if a handler cannot deal with a given exception, it *propagates* the exception up one level in the call hierarchy where it will be processed by a handler associated with the corresponding invoker.

Figure 4 shows several examples for the flow of control in OPERA during exception handling, 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. 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.

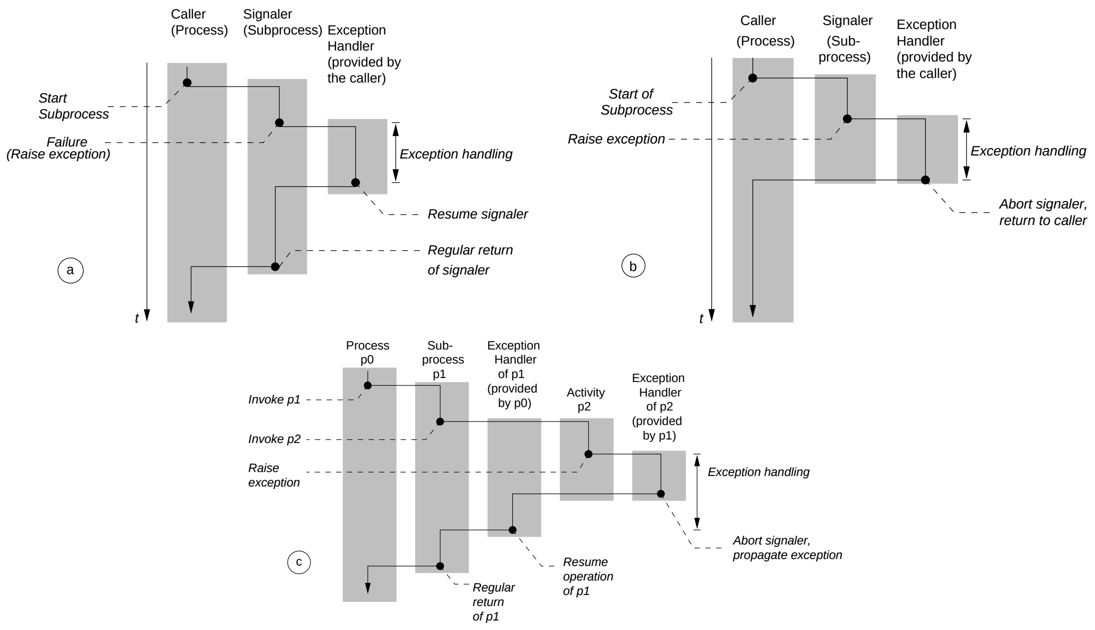

Figure 4: Control flow during exception handling

3.2 Semantics

The semantics of the Opera exception handling mechanisms are based on the *replacement model* [YB85]. Logically, the exception handler *replaces* either the signaler (if the latter is aborted) or the statement in the signaler that raised the exception (if the execution of the signaler is resumed). This poses additional requirements on the process execution model: If the exception process aborts and replaces the signaler, the system has to "undo" possible side effects caused by the signaler. This is achieved in programming languages by cleaning the stack and performing some additional cleanup work, like calling destructors of purged objects [Str91]. In Opera, each step corresponds to one or more external activities that may cause arbitrary side effects in the outside world (sending a message, deleting a file, changing a record); the engine has no knowledge of these effects. In order to be able to undo any side effects, Opera relies on semantic information provided with the failed task [AKA+96]. This semantic information is provided in the form of compensating tasks and managed using the transactional mechanisms discussed in section 4.

4 Transactional recovery and exceptions

The constructs provided by OPERA allow process designers to describe the failure semantics of a process in a convenient way but, in order to *enforce* these failure semantics, additional mechanisms are necessary. For instance, "aborting a task" needs to be properly defined. In practice, aborting a task and executing an exception handler is equivalent to what is known as *partial backward recovery* in advanced transaction models [Elm92, Mos85, GH94, GHM96, GS87, ELLR90]. (In fact, we integrated exceptions into our system primarily because we were looking for a way to specify *spheres of atomicity* at the modeling language level).

Opera integrates transaction concepts directly into the system. While the approach is similar to advanced transaction models in that it provides flexible recovery, it is more general since it can distinguish between different failure states of a task. Most transaction models can only distinguish "committed" from "aborted" transactions and hence allow to specify only one recovery strategy for each task. (To our knowledge, only [BDS+93] has discussed the implications of multiple possible failures on recovery mechanisms.) Furthermore, transaction models usually assume that tasks are atomic. This is, however, not true for workflow environments where activities can be arbitrary program executions or human activities and may thus not be atomic at all. They may leave "the outside world" in an inconsistent state if they fail. The recovery mechanism in Opera also copes with such non-atomic tasks.

The transactional aspects of OPERA are embedded in the notion of *spheres* which are used as a way to bracket operations as units with transactional properties. Spheres are basically specialized blocks and, in the current prototype, there are three possibilities which can be combined: blocks as atomic units with the standard all or nothing semantics (sphere of atomicity), blocks as isolation units (sphere of isolation), or blocks as persistence units (spheres of persistence)[AHST97b]. For the purposes of exception handling, 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 as well as for processes and activities. The information on whether an activity is atomic or not has to be provided by the person registering a program or human activity. This is done as part of the process of selecting a task interface. Currently, the following tasks interfaces are provided in Opera:

- *Basic (non-atomic):* This is the basic task interface that serves as the root of the interface hierarchy. Activities that conform to this interface are assumed to be non-atomic. The system cannot guarantee atomicity.

- *Semi-atomic:* Semi-atomic tasks do not perform automatic rollback in case of a runtime failure. They do, however, keep enough information to allow an undo after the failure has happened. Examples are certain CAD systems that perform logging during operation. After a failure, work can be undone by issuing a special command. Part of the semi-atomic task interface is thus a *rollback* method that describes how to undo a partially executed task.

- *Atomic:* These tasks have no side effects if they fail. Transactions issued to a database or to a distributed environment through a TP monitor fall into this category.

- *Restartable:* If this sort of task fails, it can be executed again and will eventually succeed. Examples are many programs (like word processors) that may fail due to failures in the program or operating system [ELLR90].

- *Compensatable:* This task interface applies to applications that can be rolled back *after* they have finished. Compensation is used in the same way it is applied in many transaction models. The *Compensatable* task interface contains a method that allows to invoke an activity or task that semantically undoes the original task's effects [ELLR90].

Based on the atomicity declarations for the basic steps, Opera enforces atomicity of composite tasks, i.e., processes and blocks. A process is guaranteed to be atomic 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 the rules given for the flex transaction model in [ZNBB94]. It can consist of compensatable, restartable, and so-called *pivot* (i.e. neither compensatable nor restartable) tasks and is guaranteed to be atomic if certain rules on its structure hold. The basic principle of recoverability in flexible transactions is that only one pivot task is allowed and that all compensatable task have to be executed before the pivot, while all restartable tasks have to be executed after the pivot. Rules become more complicated if parallel tasks and contingency executions are taken into account. The problems of correctness rules for flexible transactions are covered in depth in [ELLR90, ZNBB94]. Therefore we do not discuss them further here. Opera uses these rules to determine the recoverability of processes at compile-time.

OPERA uses the information provided by the process designers in the following way: If a task raises an exception and is aborted by its exception handler, 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 was declared atomic, then no further actions are performed. If the task was declared semi-atomic, then *holistic backout* is performed by executing its rollback method. If the task was declared non-atomic, then *single-step backout* is performed by executing the compensating tasks of the component activities. Note that for flex structured processes, an atomic abort is only possible while only compensatable activities have been executed. Once a pivot or repeatable activity has succeeded, these processes become semi-atomic in the sense that they can only be aborted through a backout method. The process designer has the possibility to determine the behavior of flex structured processes in specifying whether holistic or single-step backout is preferred when there are only completed compensatable tasks.

5 Exception detection

Internally, an exception is represented as a triple (N, O, I, R), where N is a name, O denotes the allowed control flow options (abort or resume), I is the input data structure that is used to pass information about the context where the exception occured, and R is the return data structure used when data has to be sent back to the source of the exception. The programs that are part of a workflow do usually not use this format but rely on proprietary ways of exception signaling. (We have given examples for this in section 2). OPERA translates these external exceptions into its internal format at run-time based on mapping information provided when the programs are registered

Program registration ensures that the engine has the necessary information to invoke a program or notify a user, like IP addresses and command lines in the case of programs or role specifications and desriptive text for human activities. In OPERA, exceptions have also to be declared. This includes the declaration of an *exception translation function* defining which external signals result in which internal exceptions. The format in which this function is given is dependent on the type of the external application:

Workflow-aware applications [SJHB96] are applications that have been specifically designed to be used with OPERA by incorporating calls to the OPERA application programming interface (API). The OPERA API is a library of procedure calls like the ones provided by most WFMS. The API provides calls to directly signal OPERA exceptions, hence there is no need for the translation of exceptions in workflow-aware applications.

Legacy applications are programs that are not aware of the workflow system. Being stand-alone applications invoked through the operating system, they signal failures through special return codes that are converted to OPERA exceptions by the runtime system. The conversion is based on a *translation table* registered with the program that maps specific return values to appropriate exception types.

Standard environments. Distributed environments like CORBA [Obj92] provide their own exception mechanisms. Exceptions returned by their applications are directly converted into OPERA exceptions. The exception declarations for a service are parsed from its IDL file that has to be provided when the service is registered. Note that these application do usually not allow signaler resumption, i.e. programs are always aborted when they send an exception.

Manual activities. Humans communicate with the WFMS through worklists, which are graphical user interfaces. The signaling of exceptions by human agents is supported through the worklist by a suitable GUI.

5.1 Internal exception detection

In OPERA, the exception mechanism is also used to detect and signal semantic failures. OPERA provides two options: synchronous exception raising, based on special *signal proxies* embedded into the control flow description, and asynchronous exception raising, which is based on predicates over process-internal data.

Signal proxies can occur anywhere inside a process. A signal proxy is associated with an exception name, data containers, and an exception category. If the flow of control in a process reaches a signal proxy, control is passed to the appropriate exception handler. Figure 5 gives an example of explicit signalisation. If after the execution of Activity1, the value of the parameter R is negative, an exception is raised. Since the exception type (cf. Section 6) is *Escape*, this leads to the termination of the process.

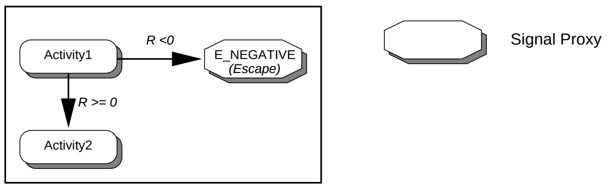

Figure 5: Explicit signaling of an exception

For implicit signaling, the workflow designer has to provide a set of predicates that define under which circumstances a given exception must be raised. The variables for these predicates are those in the containers or in the “blackboard”associated with each process [AHST97b]. The designer can for instance define *startup predicates* and *termination predicates*. The former evaluate the parameters passed to the process upon startup. They allow to check if these values allow the process to execute correctly. If incorrect values are encountered, a user can be informed through a *Notify* exception. This user can then correct the parameters and resume the process execution. Termination predicates check the return values of the process and raise exceptions if incorrect values are detected.

6 Integration of exception handling into modeling languages

Process support systems have a strong focus on process modeling. They usually provide a graphical modeling language that allows to specify processes in an intuitive way. It is thus important that an exception handling mechanism integrates well with graphical process languages. In this section we describe the modeling languages used in OPERA and how the exception handling primitives are incorporated in them.

OPERA uses a model *hierarchy* rather than one single language to describe processes. At the top of the hierarchy there are domain-specific representations of processes such as *OGWL (Opera Graphical Workflow Language)*, a modeling language for business processes. OGWL is a simple graphical process description language, based on IBM’s FDL [LA94], which closely follows the model proposed by the Workflow Management Coalition [Hol96]. OGWL specifications are only used for interacting with the user. Internally, OGWL specifications are compiled into *OCR (Opera Canonical Representation)*, which is later translated into the data models of the underlying databases used as OPERA repositories.

6.1 Graphical representation in OGWL

In OGWL, a *process* is a directed acyclic graph, whose nodes represent tasks, and whose arcs are *control connectors* and *data connectors*. Note that in spite of the graph being acyclic, loops are still possible using block constructs. Control connectors define the flow of control in a process. A control connector is a directed edge that links two activities, regardless of whether they are simple activities, blocks, or subprocesses.

Each control connector can be marked with a *transition condition*, a boolean expression over elements of the source activity’s output data structure. The condition determines a connectors *state*, and is used to model conditional branching: if a process is created, all connectors are in the *unevaluated* state. Upon completion of a task, the states of its outgoing control connectors are computed by evaluating the transition conditions, which leads to the connectors becoming either *true* or *false*. The execution of a task is then dependent on the states of its incoming control connectors; a special *start condition* (a boolean predicate over the states of the incoming connectors) determines which connectors have to be *true* in order to schedule the task's execution. Figure 6 contains an example process. Note that the two arcs starting at *Activity1* are equivalent to an *if-then-else*-construct (since their conditions are disjoint). Parallel execution can be specified by control connectors with predicates that can be true at the same time.

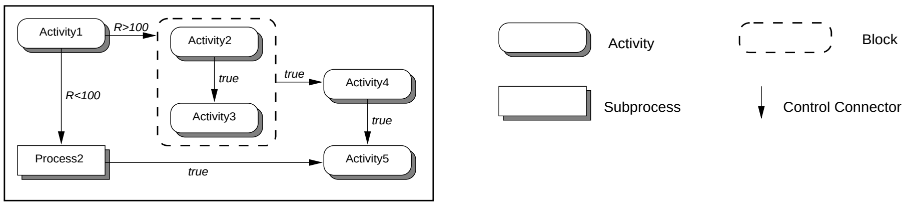

Figure 6: Control flow description

Data containers are used to model the flow of data into and out of activities, blocks, and processes. They are the equivalent to OCR's in- and out-boxes. Data flow is specified through special *data connectors*. A data connector is a directed edge that links two activities and indicates a mapping between the source's output container and the target's input container. Upon completion of a task, the system copies the data of the task's output container into those input containers that are linked by data connectors.

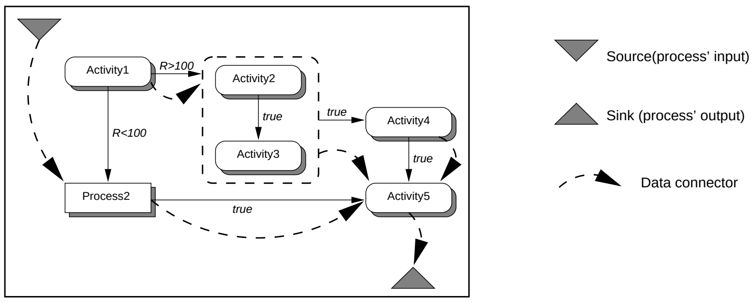

Figure 7: Data flow description

6.2 OCR process representation

OGWL is suitable for user interaction but not for internal use within Opera. For this purpose, OCR is used. Thus, when a new process is registered, the OGWL compiler translates it into OCR objects. Control connectors and their conditions are transformed into *guards*. OGWL data connectors are directly translated by storing with each input parameter the task and output parameter it reads data from.

In OCR, a process is represented as a set of tasks (activities, blocks, or subprocesses) and the so-called *blackboard*, a set of data items used for the exchange of data between tasks. *Activities* are basic processing steps. An activity is associated with a number of attributes, like the program to be executed and people responsible for execution and supervision. *Subprocesses* and *blocks* allow nesting of processes, with the difference that blocks are defined only within their parent process (similar to blocks in programming languages). In addition, blocks can be assigned special properties; for instance, *iteration* blocks and *while* blocks allow to implement loops; or *atomic spheres*, which are used to define units of rollback. Each task is represented as an object with a well-defined interface that allows to access its state and other information necessary for navigation and execution. The structure of the basic task interface is given in table 1. It represents the root of an *interface hierarchy* that allows to add new functionality to the system by extending the basic interface.

Name	**Type**	**Description**
Guard	Guard	Control flow description
InBox	Parameterlist	Call parameters
OutBox	Parameterlist	Allows to access return values
State	ExecutionState	Allows to access the current state of task execution.
ExceptionDecl	List of Exception	declares the exceptions that can possibly be thrown by the task
EventQueue	List of Event	Allows to query the list of events signalled by the task.
Type	TaskType	Distinguishes between subprocesses, blocks, and activities.
Reference	Object	A reference determining how the task is to be executed

Table 1: Basic task interface

Control flow inside a process is determined by the *guards* associated with its tasks. The paradigm used is similar to *ECA rules* in active databases [WC96]. The guard consists of an *activator* that specifies when the task has to be considered for execution, and a *start condition.* The activator is a predicate on the execution states of other tasks. The start condition can access the blackboard as well as the output data of sibling tasks. Note that only tasks within the same process are visible to the guard. This guarantees efficient evaluation of guards and is an important difference with active database systems, where all events are visible to all rules. The *InBox* and *OutBox* components of a task interface represent its input and output data structures. Data flow between tasks is possible by copying data between out- and in-boxes and the blackboard. While the incorporation of exception handling into a text-based language like OCR is straightforward, the integration into a graphical language like OGWL is not. We will focus on graphical representations in the remainder of this section.

6.3 Exceptions

Synchronous exceptions are represented like ordinary activities. They have input and output containers and a unique name which allows to distinguish different exceptions. Each exception has an associated *exception category*, which can be used to restrict the behavior of the exception handler. Three exception categories are defined in OPERA:

- *Signal:* Allows the handler to either abort or resume the signaler after processing the exception. The decision will depend on the handler's ability to deal with the exception.
- *Escape:* Requires to abort the signaler. This will be used for exceptions that do not allow resuming execution.
- *Notify:* Disallows an abort. This forces the handler to return control to the signaler, an option especially useful when humans are involved in the process.

The same data flow mechanism used for normal activities is used to handle the data flow during exception handling. Since an exception has data containers, when the exception occurs, its input container is used to pass information to the handler. Similarly, the handler has the possibility to return data to the signaler using the output container of the exception.

Asynchronous exceptions are similar to synchronous exceptions except for the fact that they do not take part on the normal control flow. They are only invoked when exceptions occur. Conceptually, they could be seen as activities to which all other activities are connected through a control connector that gets activated when the exception occurs. The advantage is that now this control flow towards the exception is implicit and the workflow designer does not need to construct it explicitly. Note that in current systems the only way to achieve similar functionality is to actually treat the exception as an activity and add control connectors between all activities that could possibly raise the exception and the exception activity. The same would need to be done with the data connectors. Many actual implementations actually resort to this very inelegant, very inefficient solution to be able to provide a minimal failure handling capability.

6.4 Exception handlers

Exception handlers can contain arbitrary activities, blocks, or subprocesses. In addition, the language provides special constructs that are useful for effective reaction to failures.

Each possible exception a task may signal has a corresponding *default handler*, which is either system-provided or defined when the task was registered. The *system default handler* matches every exception without specified handler, aborts the signaler and then propagates an exception to the caller. A process designer can, however, provide user-level default handlers where this is appropriate. For each task integrated into a process, the designer can provide *override handlers* for those exceptions where the default behavior needs to be modified. The advantage of this approach is that it facilitates modular design: Reusing components becomes easier since they will either cope with any possible exception themselves or will pass the exception up to the caller. This is a significant advantage over existing systems in which exception handling is entirely hard-wired and needs to be modified every time a process is used in a different context. By combining default and override handlers, the designer can let the system take care of exceptions and specialize the behavior when necessary.

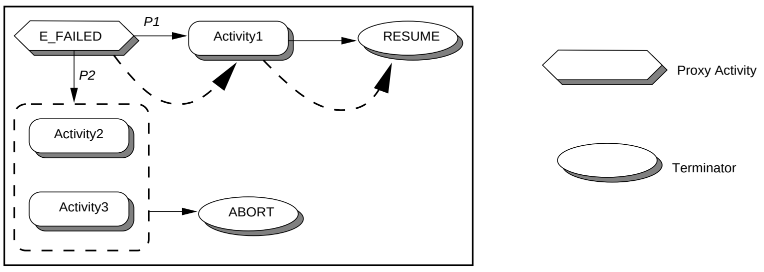

Figure 8: An exception handler

An example for the OGWL representation of an exception handler is given in figure 8. The entry point to a handler process is always a so-called *proxy activity* that can be seen as a placeholder for the exception that occurred. The output container of the proxy contains the data that have been passed by the signaler together with the exception. This makes the case-dependent data accessible inside the handler. In our example, two predicates, $P1$ and $P2$, are defined on these data. This allows to take different execution paths depending on the information provided by the signaler. *Terminator proxies* define the endpoints of a handler. They determine how the control flow has to proceed after termination of the handler. Different types of terminators can be used depending on whether the signaler has to be aborted or resumed and whether an exception is to be propagated.

In addition to the functionality provided in ordinary processes, Opera provides special constructs that can only be used in exception handlers. They are syntactical shortcuts that facilitate the convenient specification of recovery-related tasks:

- *Retry:* If recovery requires to retry the execution of the task that caused an exception, *retry proxies* are introduced. They refer to the task that raised the exception currently handled and can be marked with a time interval to specify a delay before the re-execution is to be scheduled. Since this mechanism could lead to an indefinite number of recursive invocations, the following semantics are defined for the retry mechanism: If during the retrial of T the same exception is raised again, the system does *not* call the exception handler again. Instead, the control flow returns to the first invocation of the exception handler. Note that repeated invocations of T are still possible with this rule, but need to be explicitly specified in the exception handler.

- *Human interaction:* Due to the complexity of workflow processes and the rules that determine their behavior, it may not be possible to determine the strategy to follow with exceptions at the time the process is defined. In these cases, human intervention is necessary. We provide a special *notificator proxy* that allows to transfer control to a user responsible for dealing with the exception. Most workflow management systems provide a staff modeling component that allows the flexible assignment of users to activities, usually through a role concept [Bus94]. The same mechanism is used to assign humans to exception handling tasks. This means that each notificator has an associated role that determines who has to be notified. Notificators have input and output containers that allow to pass information concerning the failure to the users and are used to send information back to the handler once the failure has been resolved.

7 Discussion

The language extensions in Opera allow the elegant specification of fault-tolerant workflow processes. These extensions are not present in other workflow systems but their functionality, if not their syntax, will have to be included in future systems if they are to provide enough functionality to allow the development of distributed applications. The language extensions, along with the corresponding system support, will result in cleaner process specifications and less overhead when designing fault-tolerant workflow processes. Regarding the system support, much of the implementation details can be directly derived from the descriptions above as the new primitives are to be treated as standard constructs and will not require significant changes to the engine (for instance, an exception can be treated as an activity which is triggered when another activity raises an exception. This involves minimal changes to the control flow logic).

To illustrate the advantages of the approach, a specification of the introductory example (Figure 2) using the new primitives is given in Figures 9 and 10. The left hand side of Figure 9 shows the graphical representation in OGWL, the right hand side displays the control flow for the case that the car rental activity fails, and figure 10 shows the textual representation in OCR. (Remember that the OCR representation is compiled from the OGWL specification, hence a process designer does not write OCR code but uses a graphical design tool that hides much of the OCR syntax.)

The process description has been decomposed into a process (*Travel*), shown in the center of the graphical process representation, and three exception handlers. Note that the process itself contains only the business logic, plus a sphere (*Transport*) that indicates that flight booking and car renting are regarded as atomic with respect to failures. The graphical representation shows the elegance of the proposed approach, especially if compared with the process in Figure 2, which is the only possible way to cope with exceptions in most current systems. Furthermore, modifying the process becomes straightforward. Consider the addition of another task in the booking process (e.g., reserving theater tickets). While in the conventional design this would require embedding several new nodes and arcs to the graph to avoid violating the failure semantics, in Opera only one new task has to be added to the process description, since all recovery-related steps are taken care of by the system. Thus the Opera approach to exception handling guarantees reusability of process descriptions, since existing specifications can easily be used as a basis for new processes. Moreover, reusability of tasks is improved, since the exception mechanism can be seen as a form of parameterization of activities and processes allowing to use a once-declared task in a large number of contexts. Consider the BookFlight program as it is shown in Figure 10. It has the required seat category as a parameter and raises the exception NO_SEAT_IN_CATEGORY if the flight has free seats, but none in the requested category. Assume that the default exception handler for this exception aborts the program and propagates an exception.

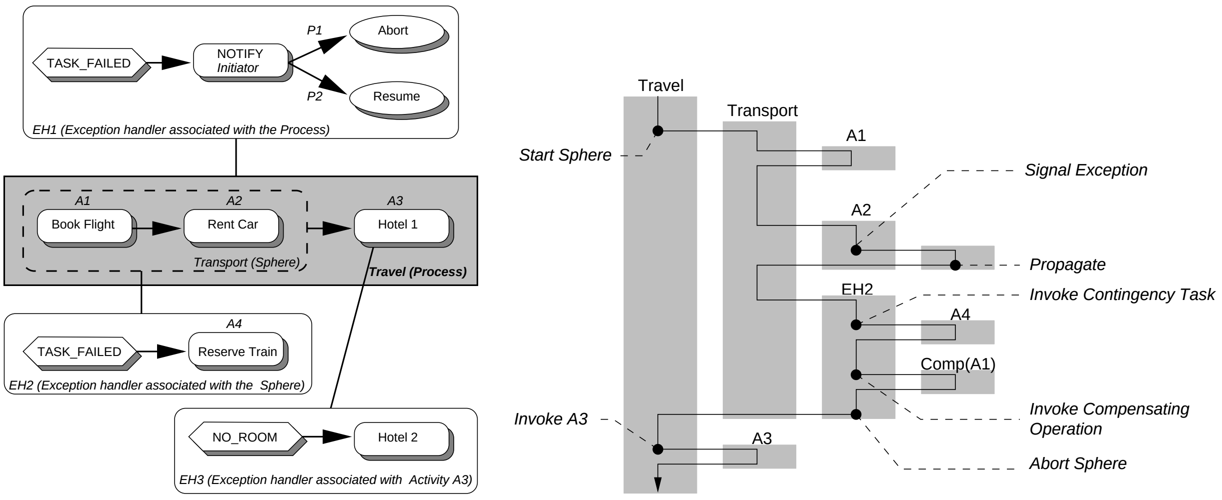

Figure 9: The travel example modeled with the new primitives and the control flow when handling a failure of activity A2

While this is the appropriate behavior in many cases, it is possible to change it by providing an override exception handler that resumes the execution of the program, allowing it to reserve a seat in another category than originally requested. Hence the criteria of reusability and flexibility are met by the approach.

The NO_ROOM exception is registered with the system, which ensures that its parameters are known to all processes and exception handlers. Note that the exception TASK_FAILED does not have to be declared, since it is a system default exception. All activities have associated default exception handlers that propagate this exception. The sphere has an associated override handler (*EH2*) that catches the propagated exception. If either A1 or A2 fail, this handler gains control and calls the train reservation task (*A3*) while the sphere is aborted, and the backout mechanism cancels reservations already made (the sphere's backout mode is declared as *Single Step*). Should the train reservation fail as well, its exception is propagated automatically to the process itself (remember that we do not allow recursive handler calls). This activates the process's handler, which notifies the process's invocator. This person has the possibility to either abort the whole process or to perform appropriate actions (for example, organize a travel by private car) and resume execution. If the execution is resumed, the process continues with the hotel reservation. This activity has an associated exception handler (*EH3*) that invokes the reservation of another hotel if no rooms are available. If this activity fails, too, an exception is propagated to the process and its handler gets control again, informing the invoker of the process.

As an illustration of the forward and backward navigation performed by the system if a failure occurs, the right hand side of Figure 9 shows the control flow if activity A2 (RentCar) fails. First, the default handler for A3 is invoked, which propagates the standard exception TASK_FAILED to the next higher level, which in this case is the sphere S. This leads to the invocation of EH2, an exception handler associated with the *Transport* sphere , which calls activity A4 (ReserveTrain) in order to handle the exception. After the completion of A4 the sphere is aborted (because of the single step backout method the system automatically calls the compensating operation for A1, canceling the flight), and operation continues with P's next regular operation A2. This example shows how, based on the failure semantics specified through spheres, exceptions and exception handlers, flexible recovery is enforced.

In the OCR representation in Figure 10, all exception handling related parts have been marked through boldface typing. The program declarations in the top region contain the declaration of their transactional properties, a reference to a compensating process (if it exists), and a declaration of the

Program registrations (User provided):

```
PROGRAM BookFlight
   INPUT: DepTime, DepCity, ArrCity, Category : String;
   OUTPUT: BookingCode: String;
   PROPERTIES: COMPENSATABLE;
   COMPENSATING: CancelFlight (BookingCode);
   EXCEPTIONS: NO_SEAT, NO_SEAT_IN_CATEGORY;
   SUBSYSTEM: ...
   INVOCATION: ...
```

```
PROGRAM ReserveTrain
   INPUT: DepTime, DepCity, ArrCity: String;
   OUTPUT: BookingCode: String;
   PROPERTIES: COMPENSATABLE;
   COMPENSATING: CancelTrain (BookingCode);
   EXCEPTIONS: NO_SEAT;
   SUBSYSTEM: ...
   INVOCATION: ...
```

```
PROGRAM BookHotel
   INPUT: City, Hotel, Begin, End: String
   OUTPUT: BookingCode: String;
   PROPERTIES: COMPENSATABLE;
   COMPENSATING: CancelHotel (BookingCode);
   EXCEPTIONS: NO_SEAT
   SUBSYSTEM: ...
   INVOCATION: ...
```

```
PROGRAM RentCar
   INPUT: City,Begin,End: String
   OUTPUT: BookingCode: String;
   PROPERTIES: COMPENSATABLE;
   COMPENSATING: CancelCar (BookingCode);
   EXCEPTIONS: NO_CAR;
   SUBSYSTEM: ...
   INVOCATION: ...
```

Registration of Exceptions and Exception handlers (Compiled from graphical representation):

```
EXCEPTION NO_ROOM
   INPUT: City, Begin, End: String;
   OUTPUT: Void
```

```
EXCEPTION HANDLER EH1
   INPUT:Start, End, City: String;
   COMPONENTS:
      TF: PROXY TASK_FAILED;
      N: NOTIFY (INITIATOR),
         ACTIVATOR: TF;
         OUTPUT: Continue: Boolean;
      T1: TERMINATOR ABORT,
            ACTIVATOR: finished(N),
            CONDITION: (TF.Continue = TRUE);
      T2: TERMINATOR RESUME,
             ACTIVATOR: finished(N);
            CONDITION (TF.Continue = TRUE);
```

```
EXCEPTION HANDLER EH2
   INPUT: Begin,End,Destination: String;
   COMPONENTS:
      TF: PROXY TASK_FAILED;
      A4: ACTIVITY ReserveTrain (Destination, Time),
            ACTIVATOR: TF;
```

```
EXCEPTION HANDLER EH3
   COMPONENTS:
      NR: PROXY NO_ROOM;
      A5: ACTIVITY BookHotel (NR.City,'Central',NR.Begin,
                                          NR.End),
            ACTIVATOR: NR;
```

Process representation (Compiled from graphical specification [Fig. 10]):

```
PROCESS Travel
   INPUT: Begin, End, City: String;
   OUTPUT: Success: Boolean;
   BACKOUT: SINGLE;
   COMPONENTS:
      Transport: BLOCK;
      A3: ACTIVITY BookHotel (City, 'Hilton', Begin, End),
           ACTIVATOR: finished(Transport),
           HANDLERS: EH3 HANDLES NO_ROOM;
    HANDLERS: EH1(Begin, End, City)
                     HANDLES TASK_FAILED;
```

```
BLOCK Transport
   PROPERTIES: ATOMIC;
   BACKOUT: SINGLE;
   COMPONENTS:
      A1: ACTIVITY BookFlight (Begin, 'Zurich', City, 'Any');
      A2: ACTIVITY RentCar (City, Begin, End);
            ACTIVATOR: finished (A1);
   HANDLERS:
      EH2(Begin,End,City) HANDLES TASK_FAILED;
```

Figure 10: OCR representation of example workflow

exceptions they may raise. Exceptions themselves and their exception handlers are declared in the middle region. Note that the structure of an exception handler is similar to the structure of an ordinary process (e.g. the *Travel* process in the bottom region), with the difference that the handler contains a proxy activity that represents the exception to be handled. Due to space limitations, the descriptions of the compensating tasks (they are programs in this example, but they could be processes of arbitrary complexity) as well as the specifications of subsystems and external references are not included in the figure.

To summarize, the extensions presented meet important criteria for a flexible process exception mechanism:

- *Support for flexible recovery strategies:* Processes and spheres provide natural boundaries for partial backward recovery. Semantic recovery mechanisms like compensation and holistic backout ensure the necessary flexibility of backward navigation, while exception handlers guarantee forward progress.

- *Reusability:* It has been shown in the example that the proposed primitives improve the reusability of process descriptions (because of the separation of business logic and failure handling semantics) as well as the reusability of tasks in different contexts (because of the parameterization realized through override handlers). This is a significant advantage over what is possible in current systems. Furthermore, failure handling strategies can be re-used since exception handlers are registered with the system and can thus be applied in various processes. As a result, it is possible to reuse defined processes and activities without further modification and without having to redo the exception handling procedures entirely.

- *Openness:* The mapping mechanisms we have described in Section 5 allow the incorporation of arbitrary applications, ranging from legacy applications over CORBA-like services to workflow-aware programs that can utilize the Opera API. Hence it is possible to achieve a seamless integration of external applications and environments with the fault tolerance mechanisms present in the system.

Finally, our model requires only minimal modifications to the representation of the business logic. The above shows that it is only necessary to add spheres in order to specify atomicity. Since all other recovery related information is described separately in the exception handlers, the process description remains comprehensible.

8 Related work

From a research point of view, the problem of workflow recovery has recently been considered in a number of projects. [Ley95] introduces *spheres of joint compensation* in FlowMark, which are sets activities that are to be jointly aborted if one component fails. Rollback is performed either by compensating each step that succeeded so far or by executing a special higher-level compensating activity. As initially proposed, spheres of joint compensation lead to very complex semantics. The most interesting ideas from spheres of compensation are incorporated in the proposed concept in the from of *backout modes* (section 4). [EL96] give a general classification of exceptions and describe recovery facilities in WAMO, a research prototype. There, processes are treated as *workflow transactions*, their components as subtransactions that can either fail or succeed. If a *vital* subtask fails, its supertask is aborted and compensated. This similar to the flexible transaction model [ELLR90]. An advanced approach for exception detection is used in the WIDE prototype[CGP$^+$96]. Here ECA rules are used for the specification of exception conditions and their handling. This approach is similar to the asynchronous exceptions provided in Opera. While it is very powerful, it does not take into account nested process hierarchies and exceptions that are signaled by external applications.

Several research projects have focussed on the integration of databases and non-databases into distributed computing environments, including work on extended transaction models (ETM) in distributed object management (DOM) [BOH+92, GH94, GHM96], where a framework for flexible transaction structures in workflows were developed, and the ConTracts project [WR92], which focussed on long-running applications and provided relaxed atomicity based on compensation and forward recovery. Recently, the work on recovery in transactional workflows has been extended in [CD96] and [CD97], where recovery mechanisms have been developed for transactional workflows that consist of transaction hierarchies with arbitrary deep nesting.

In addition, a considerable amount of work towards flexible recovery has been done in the context of advanced transaction models [BDS+93, ELLR90, Elm92, GHM96, GS87, WS91, WR92]. In particular, [AKA+96] describe mappings of advanced transaction models to workflow description languages and show how some of the concepts used in transaction management can be used in workflow environments. Our approach differs from this work mainly because of its strong focus on modeling language aspects and because we do not assume a transactional environment.

Finally, the work on exception handling in programming languages [Par72, Goo75, YB85, vZBC96, Seb96] has provided the basics for our approach. While it provides constructs for the seamless integration of exception handling into workflow descriptions, it lacks the consideration of practical aspects that become important in workflow systems such as the participation of autonomous, heterogeneous legacy systems and the strong impact of human intervention.

9 Conclusions

We have presented an extension for workflow specification languages that allows the flexible handling of exceptional situations that occur during the execution of processes. While the concepts were presented in the context of a specific system, they are generic enough to be applied to arbitrary process models.

The solution is based on exception handling concepts developed for programming languages coupled with some ideas from the advanced transaction model domain such as atomicity and partial rollback. The paper shows how these language extensions can be used to better describe workflow processes and exception handling logic as well as the support they provide in terms of forward and backward navigation, transparency, reusability, flexibility and openness.

Process support systems are certainly gaining importance for mission-critical applications. In these environments, flexible exception handling is a key aspect unfortunately not well supported by current systems. The proposed primitives allow the transparent modeling of fault-tolerant processes and could be a fundamental building block in future systems.

References

[AH97] G. Alonso and C. Hagen. Geo-opera: Workflow concepts for spatial processes. In *SSD 97*, 1997.

[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.

[AKA+96] G. Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *Proc. Intl. Conf. on Data Engineering*, New Orleans, February 1996.

[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.

[AS96] G. Alonso and H.-J. Schek. Research issues in large workflow management systems. In Sheth [She96], pages 126–132.

[BDS+93] Y. Breitbart, A. Deacon, H.-J. Schek, A. Sheth, and G. Weikum. Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *ACM SIGMOD Record*, 22(3), September 1993.

[BOH+92] A. Buchmann, M. Oszu, M. Hornick, D. Georgakopoulos, and F.A. Manola. A transaction model for active distributed object systems. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 123–158. Morgan-Kaufmann, 1992.

[Bus94] C. Bussler. Policy resolution in Workflow Management Systems. *Digital Technical Journal*, 6(4):26–49, 1994.

[CD96] Q. Chen and U. Dayal. A transactional nested process management system. In *Proc. of the 12th International Conference on Data Engineering (ICDE '96)*, 1996.

[CD97] Q. Chen and U. Dayal. Failure handling for transaction hierarchies. In *Proc. of 13th International Conference on Data Engineering (ICDE '97)*, 1997.

[CGP+96] F. Casati, P. Grefen, B. Pernici, G. Pozzi, and G. Sanchez. WIDE workflow model and architecture. Technical report, University of Twente, 1996.

[Cus93] H. Custer. *Inside Windows NT.* Microsoft Press, 1993.

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

[ELLR90] A.K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for interbase. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 507–518, Brisbane, Australia, 1990.

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

[Fla96] D. Flanagan. *Java in a Nutshell.* O'Reilly & Associates, 1996.

[GH94] D. Georgakopoulos and M.F. Hornick. A framework for enforcable specification of extended transaction models and transactional workflows. *Intl. Journal of Intelligent and Cooperative Information Systems*, September 1994.

[GHM96] D. Georgakopoulos, M.F. Hornick, and F. Manola. Customizing transaction models and mechanisms in a programmable environment supporting reliable workflow automation. *IEEE Transactions on Knowledge and Data Engineering*, 1996.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[Goo75] J.B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–695, December 1975.

[GS87] H. Garcia-Molina and K. Salem. Sagas. In *Proc. ACM SIGMOD*, 1987.

[Hol96] D. Hollinsworth. The workflow reference model. Technical Report TC00-1003, Workflow Management Coalition, December 1996. Accessible via: http://www.aiai.ed.ac.uk/WfMC/.

[Hsu93] M. Hsu, editor. *Bulletin of the IEEE Technical Committee on Data Engineering. Special Issue on Workflow and Extended Transaction Systems.* IEEE Computing Society, June 1993.

[Hsu95] M. Hsu, editor. *Bulletin of the IEEE Technical Comittee on Data Engineering. Special Issue on Workflow Systems.* IEEE Computer Society, March 1995.

[JB96] S. Jablonski and C. Bussler. *Workflow Management.* International Thomson Computer Press, 1996.

[KR96] M. Kamath and K. Ramamritham. Bridging the gap between transaction management and workflow management. In Sheth [She96].

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

[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.

[Mos85] J.B.E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, Mass., 1985.

[Obj92] Object Management Group. *The Common Object Request Broker: Architecture and Specification (CORBA).* John Wiley and Sons, 1992.

[Par72] D.L. Parnas. Response to detected errors in well-structured programs. Technical report, Computer Science Dept, Carnegie-Mellon Univ., 1972.

[Pau91] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Seb96] R.W. Sebesta. *Concepts of Programming Languages.* Addison-Wesley, 3rd edition, 1996.

[She96] A. Sheth, editor. *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems*, Athens, Georgia, USA, May 1996.

[SJHB96] H. Schuster, S. Jablonski, P. Heinl, and C. Bussler. A general framework for the execution of heterogenous programs in workflow management systems. In *CoopIS Conference*, Brussels, Belgium, 1996.

[Ste90] G.L. Steele. *Common Lisp: The Language.* Digital Press, 2 edition, 1990.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, 2 edition, 1991.

[vZBC96] P. van Zee, M. Burnett, and M. Chesire. Retire superman: Handling exceptions seamlessly in a declarative visual programming language. In *Proceedings of the IEEE Symposium on Visual Languages*, Boulder, Colorado, USA, September 1996.

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

[WfM96] Workflow Management Coalition – Terminology and Glossary, Version 2.0. Available at http://www.aiai.ed.ac.uk/WfMC, June 1996.

[WR92] H. Waechter and A. Reuter. The ConTract model. In A. Elmagarmid, editor, *Transaction Models for Advanced Database Applications*, chapter 7, pages 219–263. Morgan-Kaufmann Publ., 1992.

[WS91] G. Weikum and H.J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 13. Morgan Kauffman, San Mateo, CA, 1991.

[YB85] S. Yemini and D.M. Berry. A modular verifiable exception-handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2):214–243, April 1985.

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