

Flexible Exception Handling in the OPERA Process Support System*

Claus Hagen Gustavo Alonso

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

Abstract

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, while a process support system (PSS) generalizes this idea to arbitrary types of processes, acting thereby as a tool for “programming in the large” over heterogeneous and distributed environments [4, 2].

Programming tools intended for large, complex applications incorporate *exception handling* mechanisms to separate the failure semantics from the program logic and thus facilitate the design of readable, comprehensible code [14, 8, 18, 17]. Despite the similarities between process support systems and programming environments [2] there is little support for exception handling in current process systems. Any possible

exception must be encoded in the process. Exceptions 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 [4, 16, 11, 7].

To address this issue, the paper describes the exception handling mechanisms implemented in the OPERA process support system [2, 1]. 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, without taking into account the need to react differently to different types of failures and the need for 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. Section 2 provides a motivating example. Section 3 introduces OPERA’s exception handling concepts. Section 4 discusses the relation between workflow recovery and transaction models. Section 5 describes the integration of external applications. Section 6 proposes language primitives for exception handling. Section 7 discusses the described approach. Section 8 concludes the paper. An enhanced, more detailed version of this paper is available as technical report [9].

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

2 Motivation and Example

As a running example for the rest of the paper, consider a process reserving flights, rental cars and accommodations, sending travel documents and invoices to the customer, and updating 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. Of these, *semantic failures* are the most interesting kind of exceptions since the execution of a program without errors does not always mean that it was successful. 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.

Further motivation for structured exception handling is provided by the complexity resulting from interleaving of the original tasks with the recovery steps, which makes the fault tolerant version of a process very complex and the original process logic hardly recognizable. 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.

3 Exception handling in OPERA

The exception mechanism used in OPERA is based on programming language concepts proposed by Goodenough [8] and later adopted in many programming languages as well as systems like CORBA and Windows NT. In all these cases, a key aspect of exception handling is separating exception *detection* from exception *handling* in nested process structures.

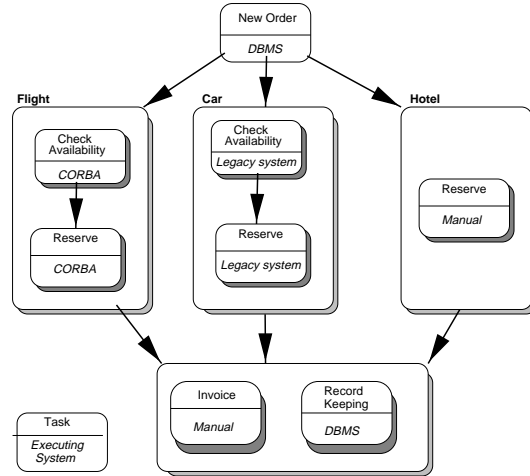


Figure 1: A workflow process

3.1 Exception Handling

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: It normally returns data conforming to its signature, but upon an error, 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

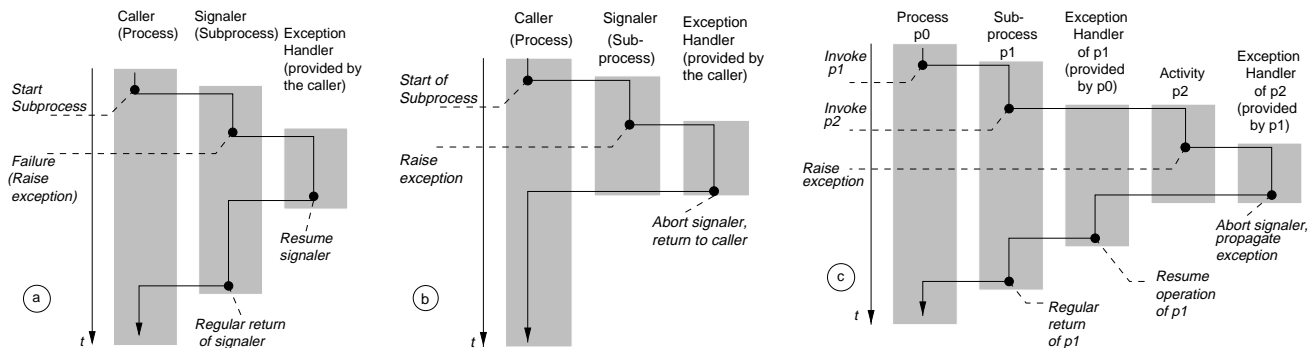


Figure 2: Control flow during exception handling

abort the signaler or to *resume* its execution after it has dealt with the exception. 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.

3.2 Exception propagation

Figure 2 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.

3.3 Semantics

The semantics of the OPERA exception handling mechanisms are based on the *replacement model* [18]. 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. 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 system 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 [3]. 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. To *enforce* these failure semantics, OPERA uses transaction concepts. We generalize advanced transaction models in that we distinguish between different failure states of a task (see, for instance, [3]). 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 [5] has discussed the implications of multiple possible failures on recovery mechanisms.) Furthermore, most transaction models 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.

The transactional aspects of OPERA are embedded in the notion of *spheres of atomicity*, which are used as a way to bracket operations as units with transactional properties[2]. Spheres are basically specialized blocks. 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 when 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.

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. 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 [19].

Compensatable: This task interface applies to applications that can be rolled back *after* they have finished using compensation. The *Compensatable* task interface contains a method that allows to invoke an activity or task that semantically undoes the original task's effects [19].

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 of the flex transaction model [19]. 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 back-

out 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. These *backout modes* generalize ideas initially proposed in [12].

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 in which the exception took place, and R is the return data structure used when data has to be sent back to the source of the exception.

5.1 External Exception Detection

For external programs, OPERA translates their exceptions into its own internal format at run-time based on mapping information provided when the programs are registered. Program registration provides the information necessary to invoke a program or notify a user. In OPERA, exceptions have also to be declared, which 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 [15] are applications using the OPERA API, a library of procedure calls similar to that provided by most WFMS. The API allows to directly signal OPERA exceptions, thereby skipping the translation step.

Legacy applications are programs unaware of the workflow system. They signal failures through special return codes, which are converted into 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 like CORBA provide their own exception mechanisms, which 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 after throwing 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.

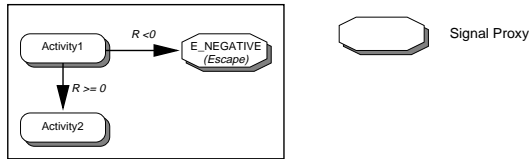


Figure 3: Explicit signaling of an exception

5.2 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 3 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.

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 designer can for instance define *startup predicates* and *termination predicates*, which trigger a *Notify* exception if incorrect values are encountered upon starting or terminating an activity or process.

6 Integration of exception handling into modeling languages

Process support systems usually provide a graphical modeling language to specify processes and, therefore, the exception handling mechanism must be integrated into the graphical language. In the case of OPERA, processes are described using a model *hierarchy* rather than one single language. At the top of the hierarchy there are domain-specific process representations such as *OGWL (Opera Graphical Workflow Language)*, a modeling language for business processes based on IBM's FDL [13] which also follows the Workflow Management Coalition Model [10]. Internally, OGWL specifications are compiled into *OCR (Opera Canonical Representation)*, a rule-based language which is later translated into the data models of the underlying databases used as OPERA repositories. While the incorporation of exception handling into a

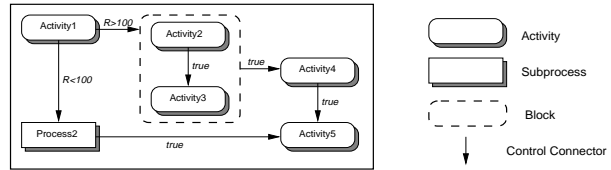


Figure 4: Control flow description

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. An example of a process representation in OCR including exception handling specifications can be found in [9].

6.1 Graphical specification language

In OGWL, a *process* is a directed acyclic graph (loops are possible using block constructs), whose nodes represent tasks, and whose arcs are *control connectors* and *data connectors*. Control connectors define the flow of control by linking pairs of activities, regardless of whether they are simple activities, blocks, or subprocesses.

Each control connector has a state determined by its *transition condition*, a boolean expression over elements of the source activity's output data structure. The connector's state is used to model conditional branching. Initially, all connectors' states are set to *unevaluated*. Upon completion of a task, the states of its outgoing connectors are computed by evaluating the transition conditions, which leads to the connectors becoming either *true* or *false*. When a task is executed is determined by its *start condition* (a boolean predicate over the states of the incoming connectors).

Figure 4 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.

6.2 Exceptions

Synchronous exceptions are represented like ordinary activities. 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 in the normal control flow. Conceptually, they could be seen as activities to which all other activities are connected through a control connector that gets activated when an exception occurs. The advantage being that this control flow towards the exception is implicit. 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. Many actual implementations actually resort to this very inelegant, very inefficient solution to be able to provide a minimal failure handling capability. As another alternative, in the WIDE project [6], ECA rules are used for the specification of exception conditions and their handling. This approach is similar to the asynchronous exceptions provided in OPERA, although without taking into account nested process hierarchies and exceptions signaled by external applications.

6.3 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 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

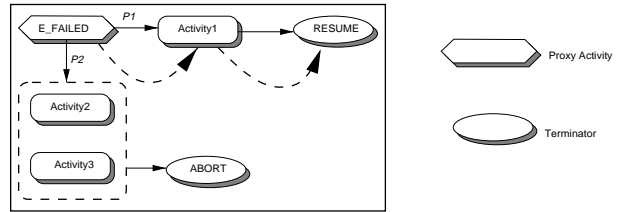


Figure 5: An exception handler

in which exception handling is entirely hard-wired. In Opera, by combining default and override handlers, the designer can let the system take care of exceptions and specialize the behavior when necessary.

An example for the OGWL representation of an exception handler is given in figure 5. 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: is performed through *retry proxies*. 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. To avoid an indefinite number of recursive invocations, if during the retrieval of T the same exception is raised again, instead of calling the exception handler again, the system returns control to the first invocation of the exception handler. Repeated invocations of T are still possible but need to be explicitly specified in the exception handler.

Human interaction: In a workflow process it is not always possible to determine in advance what to do with exceptions. These cases are handled through human intervention via a special *notificator proxy* that allows to transfer control to the user responsible for

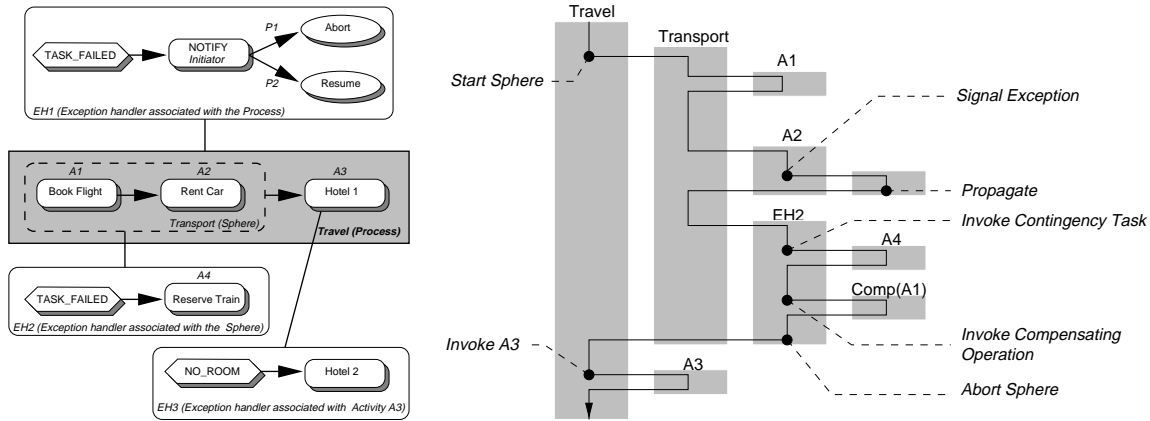


Figure 6: Travel example, using the new primitives and control flow when handling a failure of activity A2

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. The same mechanism is used in Opera to assign humans to exception handling tasks.

7 Discussion

We see the language extensions in OPERA as an elegant solution to the problem of fault-tolerance in workflow processes. The proposed extensions, along with the corresponding system support, result in cleaner process specifications and less overhead. They can be easily added to existing systems since they require only minor changes. For instance, an exception handler 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 using the new primitives is given in Figure 6. The left hand side shows the graphical representation in OGWL, the right hand side displays the control flow for the case that the car rental activity fails. 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. A clear advantage of this approach is that modifying the process becomes straightforward, thereby increasing reusability. Consider the addition of another task in the booking process (e.g., reserving theater tickets): only one new task has to be added to the process description,

since all recovery-related steps are taken care of by the system. In this regard, 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.

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 takes 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 (recursive handler calls are not allowed). This activates the process's handler, which notifies the process's invoker, who has then the possibility to either abort the whole process or to perform appropriate actions before resuming execution. In the latter case, 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 6 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 (*Reserve-*

Train) 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.

8 Conclusions

We have presented an extension for workflow specification languages that allows the flexible handling of exceptions. Given the increasing importance of process support systems in mission-critical applications, the proposed primitives could be a fundamental building block in future systems. The primitives are based on exception handling concepts developed for programming languages coupled with ideas from advanced transaction models. They provide flexible recovery strategies, increase reusability, and are open in the sense of being capable of supporting arbitrary external systems. In terms of recovery, 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. In terms of reusability, the examples above show the improvements in reusability of process descriptions and of tasks in different contexts. Furthermore, failure handling strategies can be re-used since exception handlers are registered with the system and can thus be applied in various processes. In terms of openness, the API mechanism provided allow to achieve a seamless integration of external applications within the fault tolerance mechanisms of Opera. 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.

References

- [1] G. Alonso and C. Hagen. Geo-Opera: Workflow concepts for spatial processes. In *Proc. 5th Intl. Symposium on Spatial Databases (SSD '97)*, Berlin, Germany, 1997.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] F. Casati, P. Grefen, B. Pernici, G. Pozzi, and G. Sanchez. WIDE workflow model and architecture. Technical report, University of Twente, 1996.
- [7] 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.
- [8] J.B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–695, December 1975.
- [9] C. Hagen and G. Alonso. Flexible exception handling in process support systems. Technical Report 290, ETH Zurich, Department of Computer Science, February 1998. <http://www-dbs.ethz.ch/papers>.
- [10] D. Hollinsworth. The workflow reference model. Technical Report TC00-1003, Workflow Management Coalition, December 1996. Accessible via: <http://www.aii.ed.ac.uk/WfMC/>.
- [11] M. Kamath and K. Ramamritham. Bridging the gap between transaction management and workflow management. In Sheth [16].
- [12] F. Leymann. Supporting Business Transactions via Partial Backward Recovery in Workflow Management Systems. In *Datenbanksysteme in Büro, Technik und Wissenschaft*, pages 51–70, 1995.
- [13] F. Leymann and W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 33(2):326–348, 1994.
- [14] D.L. Parnas. Response to detected errors in well-structured programs. Technical report, Computer Science Dept, Carnegie-Mellon Univ., 1972.
- [15] H. Schuster, S. Jablonski, P. Heinl, and C. Bussler. A general framework for the execution of heterogenous programs in workflow management systems. In *First IFCS Intl. Conf. on Cooperative Information Systems (CoopIS'96)*, Brussels, Belgium, 1996.
- [16] A. Sheth, editor. *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems*, Athens, Georgia, USA, May 1996.
- [17] 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.
- [18] 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.
- [19] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring relaxed atomicity for flexible transactions in multi-database systems. In *Proc. ACM SIGMOD*, pages 67–78, 1994.