Exception Handling in Workflow Management Systems

Claus Hagen, *Member*, *IEEE Computer Society*, and Gustavo Alonso

Abstract—Fault tolerance is a key requirement in Process Support Systems (PSS), a class of distributed computing middleware encompassing applications such as workflow management systems and process centered software engineering environments. A PSS controls the flow of work between programs and users in networked environments based on a "metaprogram" (the process). The resulting applications are characterized by a high degree of distribution and a high degree of heterogeneity (properties that make fault tolerance both highly desirable and difficult to achieve.) In this paper, we present a solution for implementing more reliable processes by using exception handling, as it is used in programming languages, and atomicity, as it is known from the transaction concept in database management systems. The paper describes the mechanism incorporating both transactions and exceptions and presents a validation technique allowing to assess the correctness of process specifications.

Index Terms—Dependability, exception handling, workflow management, process support systems.

1 Introduction

For the purposes of this paper, *distributed processes* can be characterized as sequences of program invocations and data exchanges between distributed and heterogeneous stand-alone systems. Business processes are perhaps the best known example of such processes. The basic tool for developing and executing business processes is a Workflow Management System (WFMS) [7], [22], [31], [32], [33], [44], [50]. A *process support system (PSS)* generalizes this idea to any type of process [3], [4]. A PSS generally consists of *buildtime* and *runtime* environment, where the buildtime environment provides a modeling language and appropriate design tools allowing us to specify processes. The runtime environment offers the necessary services for process automation and monitoring. In this sense, a process support system can be seen as a tool for "programming in the large" over heterogeneous and distributed environments [6].

Due to the characteristics of the environment where they execute and their long duration (days, maybe weeks), distributed processes are susceptible to a wide variety of failures. For instance, communication problems, computer outages, or program failures are some of the many *technical* sources for errors during process execution. Erroneous process specifications, unexpected changes in the system configuration, or absent employees are examples of the many possible *user-originated*. Thus, in order to build realistic systems, it is crucial to deploy mechanisms that allow the system to continue processing even if failures occur.

In their general classification of *system dependability* aspects, Laprie et al. [36] distinguish two ways of coping with failures in dependable systems, *fault prevention* and *fault tolerance*. While the former is concerned with "how to prevent fault occurrence or introduction," the latter deals with "how to provide a service complying with the specification in spite of faults" [36]. Fault prevention is, to a large degree, a design issue—it requires the existence of suitable design methodologies and construction rules which help to avoid introducing failures in a system. In process support systems, validation facilities such as simulation tools are used to support fault prevention [1], [39]. A complete avoidance of failures, however, is not possible. Hence, there is a clear need for inherently fault-tolerant processes.

Supporting the design of fault-tolerant processes requires:

1. Enhancing the modeling language by adding constructs for error detection and error handling and
2. Modifying the runtime system in order to implement the new semantics.

In this sense, a process support system is not different from a programming environment and, consequently, concepts for application fault tolerance, as they have been developed in many fields, could be applied. Two techniques are especially relevant for process support environments: atomicity and exception handling.

The concept of atomicity, as it is used in databases, provides a well-known abstraction for failure handling. It is based on *backward recovery*: In the case of a failure, an application or parts of it are "rolled back" to a previous consistent state. From this state, the computation can continue by retrying the previously failed instructions or by following alternative execution paths. The advantage of the atomicity abstraction is that the programmer (or process designer) does not need to specify all necessary steps for undoing work. Instead, this is left to the runtime system, which performs recovery based on logged information.

- *C. Hagen is with Credit Suisse, CIXT, CH-8070 Zürich, Switzerland. E-mail: claus.hagen@credit-suisse.ch.*
- *G. Alonso is with the Department of Computer Science, Swiss Federal Institute of Technology (ETHZ) ETH Zentrum, CH-8092 Zürich, Switzerland. E-mail: alonso@inf.ethz.ch.*

Manuscript received 2 Mar. 1999; revised 10 Dec. 1999; accepted 28 Jan. 2000.
Recommended for acceptance by A. Romanovsky.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 111487.

0098-5589/00/$10.00 © 2000 IEEE

Paradoxically, adding fault tolerance mechanisms to a system can reduce its fault tolerance due to the resulting system complexity [8]. To avoid this, programming language designers have developed language elements that allow us to separate the failure handling aspects from the "normal" flow of control [25], [40]. The same ideas could be used with processes since, when properly used, they can provide forward recovery (execution of alternative and/or compensating activities) and, thus, complement the backward recovery provided by the atomicity concept. The problem is, however, how to combine both concepts and adapt them to the context of distributed processes.

This paper describes how this problem has been addressed in the OPERA process support system [3]. In particular, we deal with exception handling of the type encountered in normal programming languages like C, C++, or Java and how exception handling can be combined with transactional atomicity. In this regard, the contribution is twofold. In the first place, we put forward the idea that process support systems are programming environments and should be treated as such. This view is not common among the process support community and, in fact, many process modeling languages lack even essential abstractions (such as modularization and nesting) necessary for comprehensively modeling large processes. In the second place, we show how to incorporate exception handling and transactional atomicity into process support systems. The result is a flexible and powerful solution that preserves consistency even if processes access multiple databases. Furthermore, to help process designers to identify potential consistency problems, an integral part of our solution is a validation service which checks the well-formedness of process specifications and gives hints to designers on how to improve their design.

The paper is organized as follows: In Section 2, we present an example and discuss the problem of integrating fault tolerance into process support systems. Section 3 gives a short overview of OPERA, the system in which these ideas have been implemented. Section 4 presents the concept of *spheres of atomicity* and Section 5 describes our approach for incorporating exception handling functionality. Section 6 illustrates the solutions using an example. Section 8 discusses the problem of well-formed processes, develops a correctness criterion, and presents an algorithm for process validation. Finally, Section 9 concludes the paper.

2 Motivation and Example

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 documents and invoices to the customer and storage of the result in the travel agency's internal database (Fig. 1). The programs and services incorporated in the process are executed by different autonomous systems. In here, we will assume the flight reservation is done through a CORBA gateway to a booking system. We will also assume that sending the documents and invoice, as well as reserving a hotel, are manual tasks to be handled by the travel agency's personnel. The record keeping in the local database will take place via a transaction processing monitor [10] and the reservation of a rental car will be done through a legacy system. During execution, this process can encounter a wide range of problems: Applications might be down, machines may have crashed, programs may return the wrong results, the process logic might be incorrect, messages may disappear, the server where the process runs may fail, and so forth. One way to deal with some of those failures is to replicate the process using back-up techniques so as to be able to resume execution even if the server where the process runs fails [30]. In some other cases, researchers have proposed dynamic modification of the process structure so as to be able to circumvent failures [35]. In this paper, we are interested in those failures that are normally considered *exceptions* in traditional programming languages like C or C++.

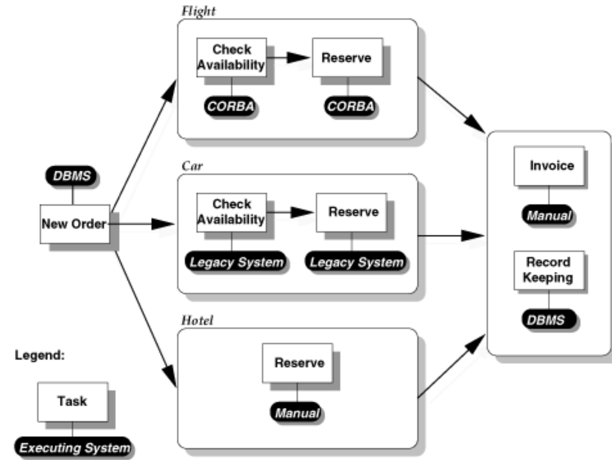

Fig. 1. A workflow process.

Exception handling, however is not enough in the context of distributed processes. Processes interact with external applications and have side effects. Dealing with failures during process execution implies being able to account for those side effects. More concretely, aborting a process in the case of a failure leaves it in an undefined state with only parts of its goals met and external resources (such as accessed databases) possibly inconsistent and with large amounts of work already performed. This is particularly undesirable for processes which are very long, consist of expensive tasks, or access multiple external data repositories. To allow the continuation of a process in spite of a failure, it is necessary to have an extended process specification which includes failure handling directives. Such directives can specify, for instance, the compensation of already executed activities in order to restore the consistency of external resources and the execution of contingency plans for failed activities. This is where transactions come into the picture. Transactions provide the necessary semantics to determine the state of the process and implement a clean recovery strategy. The question is how to combine exception and transactions in an efficient way.

As an example of the problems to avoid, it has been proposed to integrate the necessary steps for handling failures directly into process descriptions [5], [9]. This, however, is not advisable. Fig. 2 shows a slightly simplified

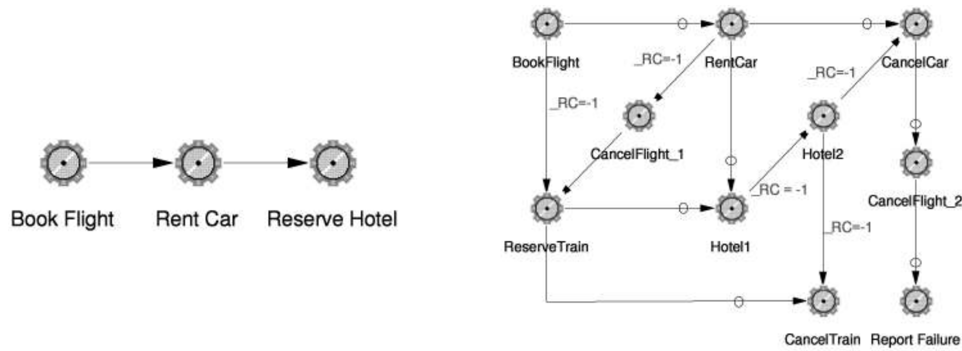

Fig. 2. Handling exceptions by programming them in control flow.

representation of our initial travel planning process using the modeling language of FlowMark [39], a workflow management system. (In the FlowMark representation of processes, the wheels denote activities, the arcs represent execution order, and conditions attached to the arcs restrict execution.) At the left side, we display the pure "application logic" without failure handling, while, at the right side, an attempt has been made to include all necessary steps for handling the failures mentioned above. The model includes 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).

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 us to separate it from the normal code.

3 OPERA—A GENERIC PROCESS SUPPORT KERNEL

The work discussed in this paper has been done in the context of OPERA, a generic process support system kernel [3]. In here, we will restrict the description of OPERA to those aspects of modeling support that are relevant to the paper. More information on OPERA can be found in references [2], [3], [27], [29].

OPERA allows the use of application-specific modeling languages for the description of processes, which are then translated into an internal format (*OCR, Opera Canonical Representation*) prior to execution [3]. To make the presentations in the paper as comprehensible as possible, we will use one of the application-specific languages, OPERA graphical workflow language (OGWL), to describe language extensions and new semantics. OGWL is an adaptation of the modeling language of a commercial workflow tool, IBM's FlowMark [38]. Like most process modeling languages, it is relatively simple and, hence, well-suited to demonstrating the new modeling features. Although we describe our solutions in the context of OGWL, they are easily applicable to other, more complex languages. Our experiences with integrating them into OPERA's internal language, OCR, have shown that very few extensions to a language are necessary to support exception handling and atomicity. Due to space limitations, however, we will not discuss these issues in detail. The interested reader will find examples of OCR in [28].

An overview of the main OGWL components is given in Fig. 3. A process consists of a set of *tasks* and a set of *data objects*. Tasks can be activities, blocks, or processes. The data objects store the input and output data of tasks and are used to pass information around. The different task types have the following semantics: *Activities* are the basic execution steps, meaning that each activity has an *external binding* that specifies a program to be executed by a person responsible for performing this part of the workflow and/or resources to be allocated for its execution. This information is used by the runtime system to execute external applications or instruct users to perform certain actions. *Blocks* are—in analogy to the block concept of programming languages—-subunits defined within the scope of a given process which are used for modular design and as specialized language constructs (for, do-until, while, fork). Blocks also serve as *spheres of atomicity*, as will be discussed later. *Subprocesses* are processes used as components of other processes. Subprocesses allow, like blocks, the hierarchical structuring of complex process structures. Late binding (the referenced process is read only when the subprocess is started) allows dynamic modifications of a running process by changing its subprocesses [38]. Blocks and subprocesses allow for the structured modeling of workflows, which leads to a process structure resembling a tree of tasks, where the intermediate nodes are blocks and subprocesses and the leaf nodes are always activities.

Control flow inside a block or process is based on *control connectors* which, formally, are triples (T_S, T_T, C_{Act}), where T_S is the source task, T_T is the target task, and C_{Act} is an activation condition. Each connector defines an execution

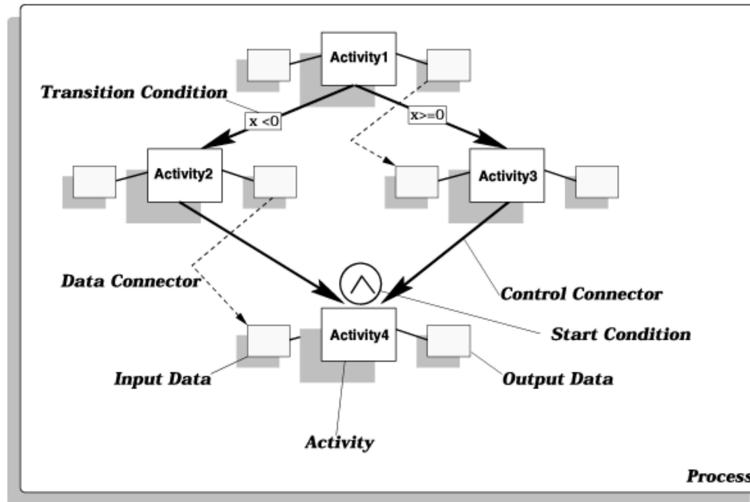

Fig. 3. Main components of OGWL.

order between two tasks and can, in addition, restrict the execution of its target task based on the state of data objects, thereby allowing conditional branching and parallel execution. Note that the concepts of control connectors and specialized blocks, when taken together, allow us to model arbitrary patterns of control flow. Using multiple control connectors departing from the same task, but with different activation conditions, it is possible to model conditional branching and trigger parallel execution.

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

4 Spheres of Atomicity

Transactional semantics are added to OPERA's process modeling languages using *spheres of atomicity*. The notion of a sphere follows that suggested by Davies [16]. The sphere concept can be seen as a way to assign to a process only those transactional properties that are actually needed. To this end, it is possible to distinguish between *spheres of atomicity*, *spheres of isolation*, and *spheres of persistence* [3], depending on the property being enforced. Here, we will focus on spheres of atomicity.

4.1 Execution Characteristics and Task Properties

The *execution characteristics* of programs, blocks, and processes are based on the notions of compensatable, retriable, and pivot tasks [19], [37]. Informally, a compensatable task is one that can be undone one way or another in case the process either fails or it is cancelled. In most real world processes, compensation is possible by executing a number of actions that cancel the effects of the initial tasks. These actions may be directly related to the task (e.g., a transactional undo) or be a semantic compensation (e.g., a letter is sent notifying the user of a given mistake). By labeling a step as compensatable we are acknowledging this fact. On the other hand, a task is a pivot when the overhead or cost of compensating it is not acceptable (note that the definitions are not mutually exclusive, it depends on the concrete application). Committing a pivot task means we are committed to complete the process because, otherwise, things will get expensive or difficult.

Finally, there are tasks that can be accomplished in many different ways. When automating a process, one can only automate one or two of these ways, but if they fail, one can still resort to the many others options. Such tasks are identified as retriable. Note that pivot tasks are assumed not to be retriable.

By introducing these notions we allow the designer to express an important aspect of the execution of a process, i.e., how severe failures can be and how to deal with them. In turn, using these notions, the system can act appropriately when failures occur. It can also check whether the process specification makes sense by analyzing it before hand and making sure that no irrecoverable situation can arise.

In OPERA, we have extended this basic classification to be able to distinguish among several types of tasks depending on how they can be treated in the event of failures.

Thus, in regard to atomicity, we distinguish between *atomic*, *quasi-atomic*, and *nonatomic* tasks. Atomic tasks are those with transactional semantics, i.e., they have no effect at all if they fail. This category also includes tasks that do not cause any changes, such as read-only operations, even if they are not transactional in nature. Quasi-atomicity represents tasks that are not atomic in the transactional sense (i.e., their effects do not automatically disappear when the transaction aborts or fails), but that can be made to appear atomic by tinkering with the system (i.e., to eliminate the effects of the task, a rollback method has to be explicitly invoked). Nonatomic tasks are those whose effects cannot be eliminated once they commit. Nonatomic tasks correspond to the notion of pivot outlined above.

In OPERA, when an activity is declared to be quasi-atomic, a *rollback-method* has to be registered which can be invoked by the system to invoke the undo. Quasi-atomicity can also be declared for blocks or processes by specifying a

rollback-method which is executed, instead of compensating each single task.

Once transactions commit, their effects can only be undone through compensation (note the difference between a *rollback method,* which is applied before a tasks commits, and a *compensation method,* which is applied after the tasks commits). A task is considered to be compensatable if the designer has specified a compensation method for it. Moreover, if a block or process is declared to be compensatable, it has to be specified whether it is to be compensated by a compensating-method or through the compensation of its component tasks. (This distinction between *integral* and *discrete* compensation was proposed in [38].)

Many tasks cannot be easily compensated once they commit. Nevertheless, they can be treated as compensatable if they are based on systems that support two-phase commit (2PC), i.e., can be rolled back after they have finished, but before they receive a final commit signal. The most common example for this class is database transactions which are controlled through an XA interface [26]. Upon declaring this characteristic, OPERA requires three methods to be registered. The *prepare-method* and *commit-method* are used for the respective messages of the 2PC protocol. The *abort-method* initiates the undo of the activity. Note that, when a task is compensatable, it does not make sense to defer the commit. The commit is deferred only for those tasks that, once they commit, they cannot be compensated. By deferring the commit, we prevent getting into a situation in which we cannot go backward and eliminate side effects. Hence, although it is always technically possible to defer the commit of any task, this will only be done for tasks that are not compensatable but need to be treated as such.

Finally, tasks can also be categorized by their ability to commit. In here we can distinguish between *guaranteed success*, *retriable*, and *nonretriable*. Guaranteed success applies to blocks or processes. For instance, a process that consists only of retriable tasks is guaranteed to succeed. Retriable tasks, as pointed out above, are those that when they fail, can be executed again or in another form and will eventually succeed [19]. Nonretriable tasks are those that if they fail once, will never succeed afterwards (or the user is not interesting in trying).

4.2 Well-Formed Spheres

Based on the task properties outlined above, OPERA enforces the atomicity of the spheres defined within a process. Moreover, it can also tell whether a process will end up in an impossible situation where recovery is not feasible. Thus, a process as a whole is atomic if:

1. There is a rollback-method defined for the whole process or
2. Every task is atomic or quasi-atomic and the process is well-formed.

This simple rule is a direct consequence of the task properties we have discussed. First, if the process has a rollback-method, it can be aborted at any time and it will not leave any side effects behind (more formally, the process is quasi-atomic). Second, the notion of well-formed is used to imply that the process is organized in such a way that there is always a way either forward (toward termination) or backward (compensate everything). Of course, it is still necessary to make sure that if a task fails in the middle of the execution, its effects can be undone, hence, the requirement that all tasks should be either atomic or quasi-atomic. The reason why processes need to be well-formed is that a task can be atomic but if it commits, we may not be able to compensate it. That is, once that task commits we cannot go back. If the process does not have the right structure (i.e., it is well-formed), this can quickly lead us into trouble if, at a later time, the process cannot progress forward any longer.

To define well-formed processes more formally, we first need to establish some notation.

Definition 1 (Basic notation).

- *A sphere S is a tuple* $(T, \rightarrow)$. $T = t_1, t_2, .., t_n$ *is a set of tasks.* $\rightarrow \subset (T \times T)$ *is the precedence relation defined on the tasks.*
- *Let* $PROP$ *be the set of task execution characteristics, i.e.,*

$$PROP := \{atomic,\ quasiatomic,\ retriable,\ guaranteed_success,\ compensatable\}.$$

- *The function* $prop : T \rightarrow 2^{PROP}$ *assigns to each task a set of execution characteristics. As a shortcut, we introduce test predicates allowing us to determine certain properties of tasks directly. For instance, the predicate* $atomic : T \rightarrow P$ *is true if and only if the task to which it is applied is atomic. Respective test predicates exist for all task execution characteristics.*
- *To simplify matters, the test predicate* $comp(t)$ *returns* true *if* t *is compensatable. Likewise, the Boolean predicate* $retr(t)$ *returns* true *if* t *is either retriable or guaranteed to succeed.* $pivot(t)$ *returns* true *if* t *is a pivot (i.e., if it is noncompensatable and nonretriable).*

Intuitively, a pivot task is neither retriable nor compensatable. Hence, the outcome of a pivot decides the outcome of the complete sphere. If the pivot fails, the sphere has failed (because the pivot is not retriable) and has to be rolled back (by calling its rollback-method or by compensating all tasks executed so far). If the pivot succeeds, undoing is possible only through the rollback-method, but not through compensation of tasks (because the pivot has no compensation). Hence, for a sphere without rollback-method, after the success of the pivot it has to be guaranteed that no rollback will become necessary. This, obviously, is the case only if there is path consisting solely of retriable or "guaranteed success" tasks from the pivot to the end of the process. From here, a well-formed sphere is defined as follows (we use the term *well-formedness* as in [19].

Definition 2 (Well-formed atomic spheres).

1. *A sphere S is allowed to have at most one pivot task.* $\forall t_1, t_2 \in T : pivot(t_1) \wedge pivot(t_2) \Rightarrow t_1 = t_2$.
2. *Every possible execution path to this pivot* t_p *must consist solely of compensatable tasks.* $\forall t \in T : (t \rightarrow t_p) \Rightarrow comp(t)$.
3. *Every possible execution path from a pivot or retriable task must consist solely of tasks which*

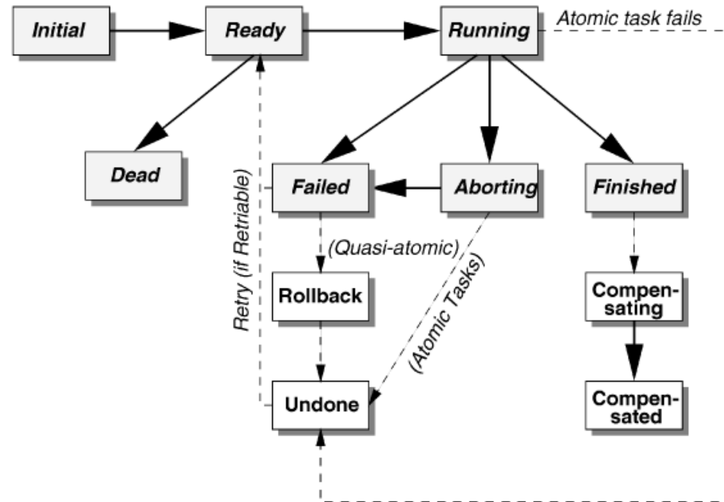

Fig. 4. Task state diagram for atomic tasks.

are retriable or guaranteed to commit. $\forall t_1, t_2 \in T : (t_1 \rightarrow t_2 \wedge (pivot(t_1) \vee retr(t_1))) \Rightarrow retr(t_2)$.

4. *Parallel tasks must have the same type.* $\forall t_1, t_2 \in T : (\neg(t_1 \rightarrow t_2 \wedge t_2 \rightarrow t_1)) \Rightarrow (comp(t_1) \wedge comp(t_2)) \vee (retr(t_1) \wedge retr(t_2))$.

This definition ensures that, within a sphere, it is always possible to either go back to the beginning by compensating if the pivot has not committed or, if the pivot has committed, successfully commit by following a path of retriable tasks.

4.3 The Rollback Algorithm

Aborting a sphere has to take into consideration the task execution characteristics of the sphere, as well as its component tasks. Given that the sphere is well-formed, the following algorithm can be used:

1. **Stop execution:** Abort all tasks that are currently running. Depending on the execution characteristics of these tasks, it is necessary to invoke their rollback-method (for quasi-atomic tasks).
2. **Rollback:** We distinguish between the atomic and the quasi-atomic case. If the sphere is atomic, perform single-step back out: Compensate each task, thereby using the reverse order of the original execution.[1] If the sphere is quasi-atomic, invoke its *rollback-method*. The component tasks are not compensated. This strategy can speed up the recovery significantly and is possible in many situations where no work actually has to be undone, but some notification of the abort has to be written to a log or sent to an administrator. We note again that the holistic back-out should not be confused with holistic *rollback*, which is performed for compensatable composite tasks for which a compensating method was defined.

1. Note that it is possible to compensate all steps in a sphere in parallel if it is known that there are no conflicts between the compensating activities. However, we have no knowledge of possible conflicts. The separate rollback of steps in reverse order is always safe.

Note that, for well-formed spheres, an atomic abort is only possible when only compensatable activities have been executed. Once a pivot or repeatable activity has succeeded, these spheres become quasi-atomic in the sense that they can only be aborted through a rollback-method. The process designer has the capability of determining the behavior of spheres by specifying whether holistic or single-step back-out is preferred when there are only completed compensatable tasks.

4.4 Task State Model

The task execution characteristics described above lead to the state transition diagram shown in Fig. 4. The gray states are the standard ones existing for a traditional process model. The new states, which have white background, reflect the new execution semantics. The state transitions, which are drawn as dotted lines, apply only to specific types of tasks. The most relevant changes over the traditional model are as follows: After the failure of a quasi-atomic activity, its rollback is initiated, which leads to the *Undone* state. If an atomic activity fails, it changes into the *Undone* state directly. Retriable activities reenter the *ready* state, either from the *Failed* state (if they are nonatomic) or from the *Undone* state (if they are atomic or quasi-atomic). A compensating task changes into the *Compensating* state while its compensation method is executing. After the successful undo, it changes into the *compensated* state.

5 Exception Handling

5.1 Basic Mechanisms

The exception mechanism used in OPERA is based on programming language concepts proposed by Goodenough [25] and later adopted in many programming languages, including CommonLisp [45], Standard ML [41], C++ [46], and Java [20]. The main difference of our approach is the tight integration of the exception handling functionality with transactional semantics realized through the spheres of atomicity. Together, they provide strong semantics which are not provided in general.

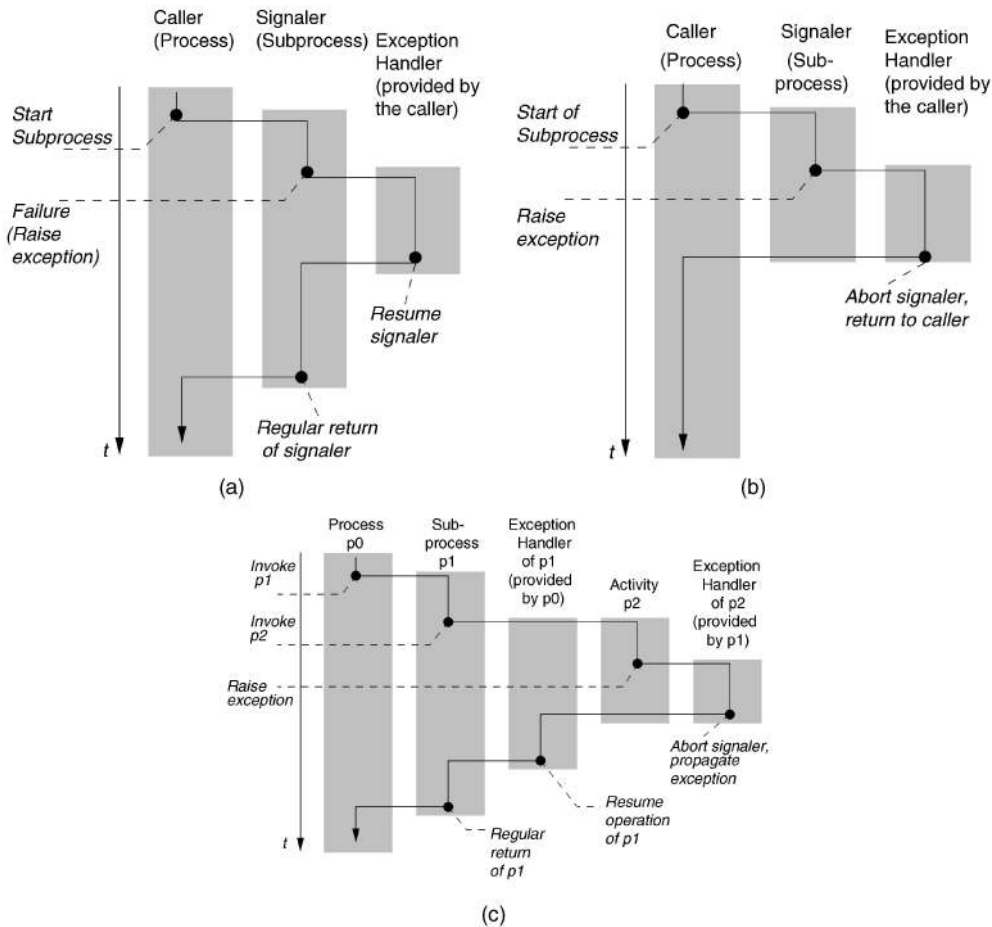

Fig. 5. Control flow during exception handling. (a) Resume. (b) Abort. (c) Propagation.

An exception is an unusual event, erroneous or not, that is detectable either by hardware or software and that may require special processing [43]. 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 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 [14], [15], [39], 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 to a level in the call hierarchy where it will be processed by a handler associated with the corresponding invoker.

Fig. 5 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

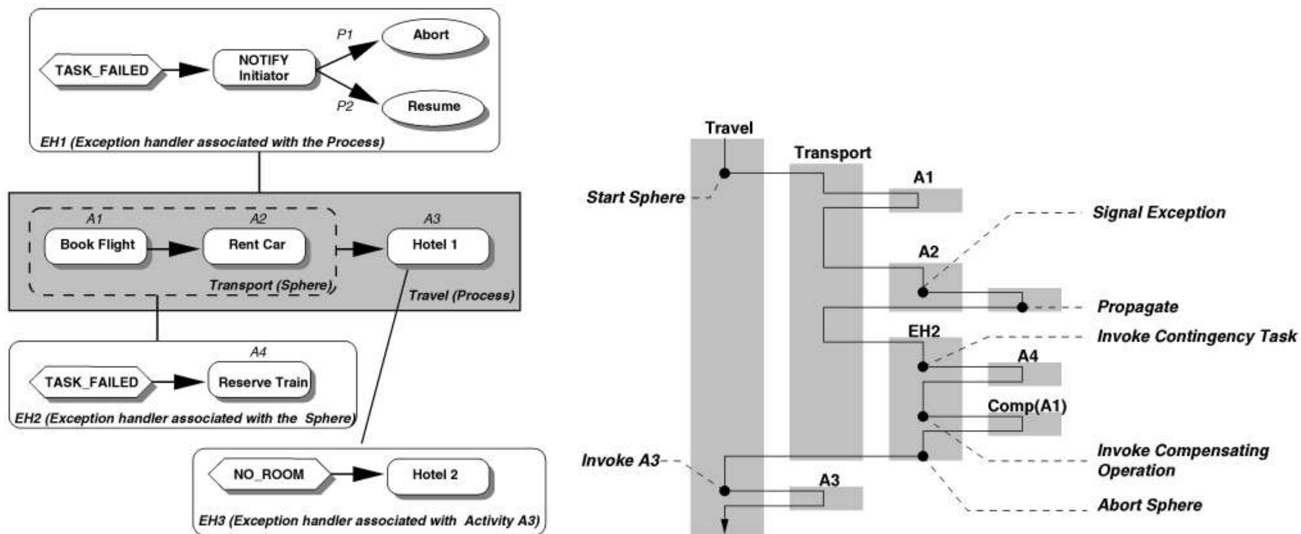

Fig. 6. The travel example modeled with the new primitives and the control flow when handling a failure of activity A2.

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.

5.2 Semantics

The semantics of the OPERA exception handling mechanisms are based on the *replacement model* [52]. 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 [46]. 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, etc.); 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. This semantic information is provided in the form of compensating tasks and managed using the transactional mechanisms discussed in Section 4.

6 EXAMPLE

To illustrate the approach, a specification of the introductory example (Fig. 2) using the new primitives is given in Fig. 6. The lefthand side shows the graphical representation in OGWL and the righthand side displays the control flow for the case that the car rental activity fails.

The process description has been decomposed into a process (*Travel*) 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 Fig. 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 use of a once-declared task in a large number of contexts.

As an illustration of the forward and backward navigation performed by the system if a failure occurs, the righthand side of Fig. 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 (ReserveTrain) in order to handle the exception. After the completion of A4, the sphere is aborted (because of the single step back-out 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.

7 CORRECTNESS OF FAULT-TOLERANT PROCESS SPECIFICATIONS

Atomicity and exception handling mechanisms give process designers the ability to specify fault-tolerant processes by

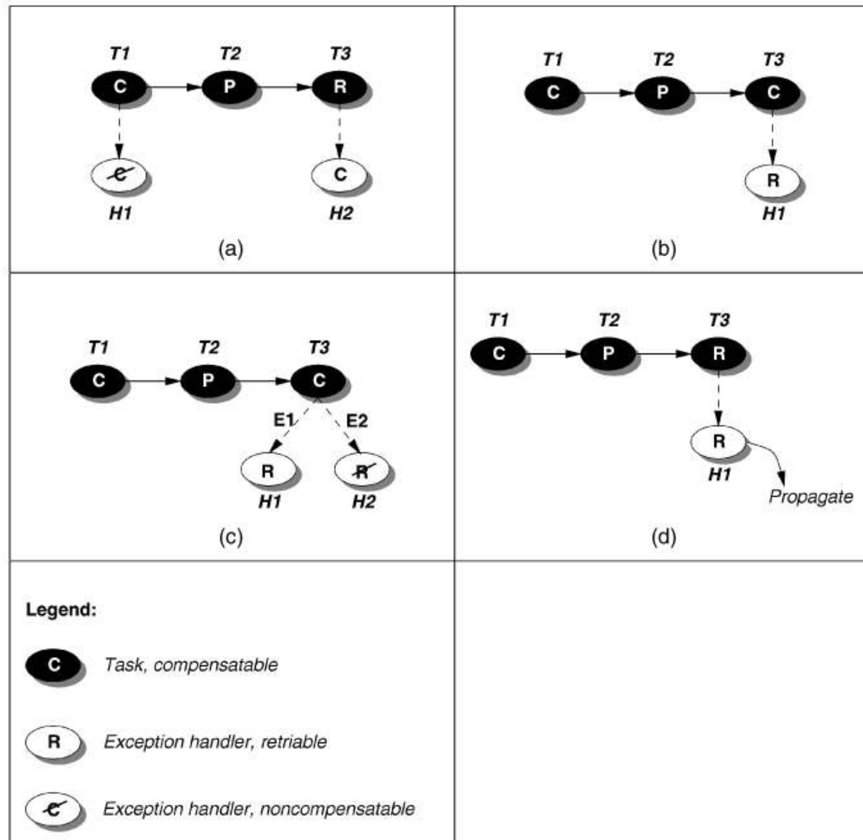

Fig. 7. Examples for changed execution semantics due to exception handlers.

deliberately creating spheres as units of joint compensation and defining appropriate failure recovery strategies through exception handlers. The combination of the mechanisms, however, complicates the detection of failures in process specifications. The rules for well-formed processes, given in Section 4.2, are not sufficient since new violations of atomicity can be introduced through inappropriate use of exception handlers. Hence, what is needed is a new correctness criterion and a validation mechanism that aids process designers by checking process specifications, detecting failures, and proposing changes that lead to correct processes. The aim of this section is to define the notion of a well-formed process specification and to describe an algorithm for correctness checking. In doing so, we use the well-formedness criterion for atomic spheres (Section 4.2) as a starting point. It will, however, be extended in order to take the exception handling functionality into account.

7.1 Examples

The well-formedness definition of Section 4.2 is valid only as long as a process contains no exception handlers. If handlers exist, it is possible that the execution characteristics of a task are changed through the execution of a handler. Recall that, due to the replacement model semantics, it is possible that a compensatable task is replaced by a noncompensatable exception handler, which may lead to an incorrect process structure. The following examples will serve as a starting point for the further investigation of these problems.

Consider the four examples for process structures in Fig. 7. The process of example (a) has a correct ordering of tasks if the exception handlers are not taken into consideration. The handlers, however, have inappropriate execution characteristics, leading to nonatomicity if one of them is executed. The noncompensatable handler attached to a compensatable task, for instance, leads to unrecoverable effects because it cannot be undone if the pivot task should fail. The same effect occurs for the retriable task at the end of process (a). It is replaced by a nonretriable handler, which again can lead to nonatomicity if the exception handler fails.

Example (b) shows the reverse effect, i.e., the correction of an ill-structured process through the introduction of exception handlers. The compensatable task at the end of the sphere has an exception handler which is retriable. Hence, it is guaranteed that the sphere completes after a successful termination of the pivot, even if the compensatable activity should fail. Note that the reverse case is not possible: A retriable activity *before* a pivot cannot be tolerated, even if it has a compensatable exception handler.

Another important observation is that *all* exception handlers attached to a task have to be retriable in order to "repair" a nonretriable task. Hence, a case like example (c) is not correct since, if the third task fails with exception $E2$, the nonretriable handler is selected. If this handler fails, the process is blocked and no rollback is possible.

Example (d) shows incorrect behavior due to the fact that a sphere is aborted at the wrong time. The third task has an exception handler which propagates the exception up to the

next higher level in the invocation hierarchy. Here, it is caught by a higher-level exception handler. In the case that this handler decides to abort the sphere, atomicity is violated since the pivot activity has already been executed. This is an important example since it shows that the correctness of the exception handling mechanism has to take into account the state of a sphere at the time an exception handler is invoked and a sequence of exception handler invocations over several levels.

7.2 Notation

Our goal is now to derive a well-formedness criterion for atomic spheres enriched with exception handling elements. We start by introducing the necessary notation and will then present conditions that have to hold for correct process definitions.

Definition 3 (New notion of spheres). *We start by giving a modified version of Definition 1. The main difference is the inclusion of exception-handling related aspects.*

- *A* sphere *S is a quadruple $(T, H, <, e)$. T is the set of* component tasks, *which may be activities, blocks, or subprocesses. H is the set of exception handlers used in the block. $< \subset T \times T$ is the control flow relation. It is derived from the* activator *part of the guards attached to each task. $e \subset ((T \cup H) \times E \times T)$ is the* exception handling relation, *where E denotes the domain of all exception types. Hence, the exception handling relation relates tasks to exception handlers based on the exceptions that may occur. Note that this relation can be determined based on the declarations which have to be provided by the process designers. Note also that exception handlers can raise exceptions as well. Hence, they are also included in the relation.*
- *As a shortcut, we will denote with T^+ the union of tasks and exception handlers, $T \cup H$.*
- *The function* prop, *mentioned in Definition 1, is extended to work on T^+ instead of T. Likewise do the test predicates, i. e.,* atomic *has now T^+ as its input set.*

Definition 4 (Transitive task execution characteristics). *The examples (a), (b), and (c) show that the execution characteristics of a task can change if an exception handler is executed. To capture this fact, we introduce a new function $prop^* : T^+ \rightarrow 2^{PROP}$ and a new set of test predicates $compensatable^*$, $retriable^*$, etc., which are based on $prop^*$. $prop^*$ is defined recursively:*

1. *If t has no associated exception handler, then $prop^*(t) := prop(t)$*
2. *If t has a set of associated exception handlers H_t, then*

   a)

   $$(\forall h \in H_t : retr^*(h)) \Rightarrow (prop^*(t) := prop(t) \cup \{retriable\}),$$

   i.e., if all exception handlers of a task are retriable, the task is considered to be retriable.

   b)

   $$(\exists h \in H_t : \neg comp^*(h)) \Rightarrow (prop^*(t) := prop(t) \setminus \{compensatable\}),$$

   i.e., if at least one exception handler of a task is noncompensatable, the task has to be considered noncompensatable.

7.3 Correctness

The correctness criterion for fault-tolerant processes is enclosed in the notion of well-formedness given below.

7.3.1 Well-Formed Process Specifications

A well-formed process specification has to suffice the following three requirements:

1. **Atomicity of components.** The first necessary condition for block atomicity is that either all component tasks have to be atomic or quasi-atomic or the block has to be quasi-atomic. There is one exception to this rule. If a task is retriable, it is allowed to be nonatomic. We assume that the reexecution of a retriable activity after a failure cleans up possible effects of the previous erroneous execution.
2. **Flex structure under replacement.** As a second necessary condition, we postulate that a sphere must adhere to the well-formedness definition of Section 4.2, with the difference that well-formedness must hold for the transitive task execution characteristics as they were defined in Definition 4. This correctness requirement captures cases such as the ones described in the examples (a), (b), and (c) of Fig. 7.
3. **No abort after critical point.** A sphere that contains noncompensatable activities can be rolled back only until a certain point in its execution is reached. This point, which is defined by the successful termination of the first noncompensatable activity, is called a *critical point* [13]. The specification of a sphere can have multiple critical points due to parallel branches.

   It has to be avoided that the sphere is aborted after the critical point since rollback is not possible anymore. Obviously, there are two cases in which an abort of a sphere is possible:

   a) An exception is raised in the sphere and the handler dealing with this exception aborts the sphere.
   b) Handling an exception in the sphere leads to an exception propagation and the resulting handling of the exception on the next higher level leads to an abort of the sphere.

   Hence, the atomicity of a sphere can only be guaranteed if, after a critical point, no exceptions can be raised that could ultimately lead to an abort of the sphere. An algorithm for testing this condition will be given in the next section.

TABLE 1
Deriving Execution Characteristics of Complex Tasks

Execution characteristic	Definition	Structure
atomic		Well-formed structure (see section 7)
quasi-atomic	Rollback-method provided	
compensatable	Compensating method provided	All components compensatable
commit_deferrable		All components commit_deferrable
retriable		
guaranteed success		All components retriable

7.3.2 Deriving Execution Characteristics of Complex Tasks

If a sphere contains blocks or subprocesses, it is necessary to compute the task execution characteristics of these complex tasks in order to access the correctness of the sphere. The characteristics of a block are based either on the characteristics of its component tasks or on specifications given by the process designers. Table 1 defines, for each execution characteristic, the necessary conditions which they have to hold. A process is compensatable, for instance, if either all of its components are compensatable or if a compensating method has been provided which allows us to compensate the whole process without regard to its components. Quasi-atomicity, on the other hand, can only be achieved through definition, by specifying a rollback method. A property that is only based on structure is the deferrable commit and only if all components of a block are commit-deferrable can the commit of the block can be deferred.

8 A Validation Algorithm for Well-Formed Atomic Spheres

We will now describe a validation algorithm for atomic spheres that tests for the three necessary conditions described in the last section.

8.1 Preliminaries

Given that we want to validate a sphere S, the algorithm assumes the existence of the following information:

- The set of component tasks T_S. We assume that each task t is represented by a tuple (E_t, C_t), where E_t is the set of exceptions it can possibly raise and C_t is the set of task execution characteristics of the task which have either been declared or are (for complex tasks) derived according to Section 7.3.
- The set of exception handlers H. Each exception handler h is represented by a tuple (C_h, T_h), where C_h is the set of task execution characteristics and T_h is the set of possible terminations (out of *resume*, *abort*, and *propagate*) of this handler. This information can be derived from the specifications of the exception handlers. We denote with T^+ the union of exception handlers and tasks.
- The control flow relation $<_S \subset (T_S \times T_S)$ describes the relative order of the component tasks. It can be derived from the activators, which are part of the process specifications.
- The *internal* exception handling relation for the component tasks, $e_i \subset (T^+ \times E \times H)$. It defines which exception handler has to be invoked if a particular exception is signaled by a particular task. Note that this relation does not cover exceptions raised by signal proxies since those exceptions are not handled internally, but have to be treated by exception handlers of the sphere.
- The *external* exception handling relation of the sphere. It declares the exceptions the sphere may signal (if a signal proxy is executed or if an exception handler propagates its exception) and the exception handlers to be invoked. It is represented as $e_e \subset E \times H$.
- To simplify matters, we will omit indices whenever no ambiguities are possible because of the context. For instance, instead of $<_S$, we will often use $<$ to denote the control flow relation if it is clear which sphere is meant.

8.2 Algorithm

The validation algorithm is graph-based and runs through five phases, which are executed according to the graphical representation of Fig. 8. In the following, they are described in detail.

Step 1. Constructing the internal exception handling graph. The *internal exception handling graph* $G_I(S)$ for a sphere S is a directed, multicolored graph that has as its

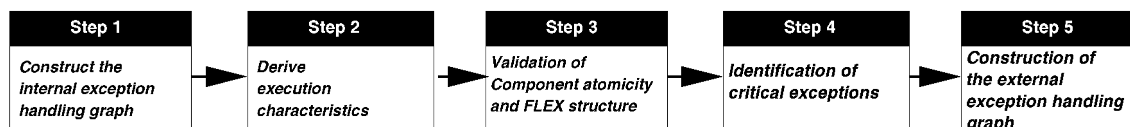

Fig. 8. Steps of the validation algorithm.

TABLE 2
Example Sphere

Name	Type	Characteristics	Activator	Exceptions	Termination
T1	Activity	atomic,compensatable	*initial*	E1	-
T2	Activity	atomic,retriable	*finished(T1)*	E1,E2	-
P1	Signal Proxy	-	*finished(T2)*	E3	-
H1	Handler	atomic,compensatable	-	E2	abort
H2	Handler	atomic,compensatable	-	-	abort,propagate
H3	Handler	atomic,compensatable	-	-	resume
H4	Handler	atomic,compensatable	-	-	abort

Components of the sphere

Task	Exception	Handler
T1	E1	H1
T2	E1	H1
T2	E2	H2
H1	E2	H2

Internal exception handling

Exception	Handler
E2	H3
E3	H4

External exception handling

nodes the elements of T^+, i.e., the tasks and exception handlers used in S. The graph has two types of edges, reflecting the control flow relation $<$ and the internal exception handling relation e. For each element (t_1, t_2) of $<$, a $<$-edge is inserted between t_1 and t_2. Likewise, for each element (t, e, h) of e, an e-edge is inserted between t and h. This edge is labeled with e. The graph is not allowed to have loops.

An example sphere is described in Table 2. It consists of the two activities T1 and T2 and the signal proxy P1. Two exception handlers, H1 and H2, are attached to the activities. Furthermore, two exception handlers, H3 and H4, are attached to the sphere itself. The properties of the various tasks and handlers can be seen from the table. The other two tables describe the internal and external exception handling relations.

The internal exception handling graph for this sphere is shown in Fig. 9. Note that no edge is leaving P1 since its exception impacts the external exception handling of the sphere, not its internal exception handling.

Step 2. Deriving execution characteristics. In a first step, the $prop^*$ relation is computed by first computing the execution characteristics of composite tasks (according to Section 7.3) and then deriving the transitive execution characteristics (according to Definition 4). For the first part, each composite task has to be analyzed recursively.

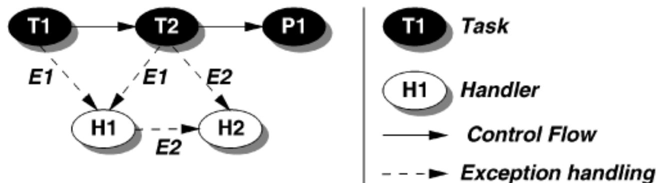

Fig. 9. Internal exception handling graph for the example.

For the second part, the subgraph containing only e-edges is traversed in reverse order, starting with exception handlers that throw no exceptions themselves and ending with plain tasks. During the traversal, the transitive execution characteristics for each node can be computed as a function of the characteristics of its successors.

In our example, it is easy to see that the $prop^*$ function equals the $prop$ function. Either of the two conditions given in Definition 4 are given since all exception handlers are compensatable.

Step 3. Validating component atomicity and FLEX structure. Now that the $prop^*$ function is known, the first two necessary conditions of Section 7 can be validated. In addition, the critical points have to be identified.

To achieve this, the graph is traversed depth-first from left to right. Each visited node is first checked for atomicity or quasi-atomicity. The structure rules for well-formed spheres are checked by appropriately marking visited tasks and ensuring that no path contains nonretriable activities after the first noncompensatable activity occurred. It is also checked that parallel branches have the same type. Critical points (i.e., a noncompensatable task directly following compensatable ones) are marked when they are visited.

In the example, T2 is a critical point since it is the first noncompensatable task. It is easy to verify that component atomicity and FLEX structure are not violated because each task is atomic, $T1$ is compensatable, and $T2$, as well as $P1$, is retriable.

Step 4. Identifying critical exceptions. Now, we modify the graph in order to validate the third correctness criterion of Section 7. First, all nodes are removed that do not follow a critical point (this includes the critical points themselves). The removed nodes are those from which a rollback is possible. Hence, the remaining nodes enforce

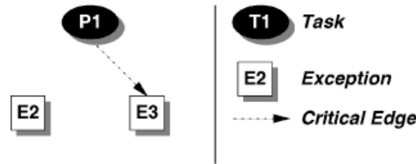

Fig. 10. Modified graph for the example.

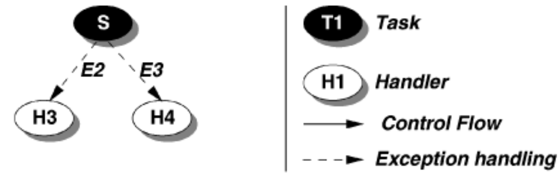

Fig. 11. External exception handling graph for the example.

a successful termination of the sphere. Due to the removal of tasks, there may be exception handlers which are not reachable anymore. Those are removed as well, together with all edges connected to them.

In a second step, we add as new nodes all *exceptions* that may be signaled by the sphere itself. New edges are inserted. These so-called c-edges (critical edges) reflect the external exception handling relation. They are generated by the following rules:

1. If an exception handler h is reached by an e-edge labeled with exception E and if h has propagation as a possible termination, we draw a c-edge from h to the node for E.
2. If a signal proxy p can raise an exception of type E, we draw a c-edge from p to the node for E.

Now we can derive the set of *critical exceptions* for S. A critical exception is represented by a node which has an incoming c-edge. The rationale behind the introduction of critical exceptions is that they are signaled by the sphere while it is in a critical state. Hence, correctness requires that the handling of a critical exception cannot lead to an abort of the sphere.

In our example, the removal of nodes leads to the elimination of the nodes for T1 and T2 since T2 is the critical point. The exception handlers H1 and H2 are also removed since they were only reachable from the removed tasks. The resulting graph, together with the newly introduced external exception information, is shown in Fig. 10. The two external exceptions of the sphere, E2 and E3, are added, as well as a c-edge from P1 (the signal proxy) to E3. Hence, the only critical exception in this example is E3.

Step 5. Constructing the external exception handling graph. In a last step, the impact of handling the critical exceptions is evaluated. To this end, we construct the *external exception handling graph* based on the external exception handling relation. It contains, as nodes, the sphere S itself and all exception handlers which are transitively reachable from S through e-nodes.

The external graph for the example is shown in Fig. 11. S signals two possible exceptions, E2 and E3, which are caught by the handlers H3 and H4.

All handlers which cannot be reached through a path starting with a critical exception (as computed in the previous step) can be removed from the graph. The termination options of the remaining handlers have to be examined. If they include an abort, the correctness of the sphere cannot be guaranteed since it is possible to have an execution of the sphere in which an exception which occurs after a critical point leads to an abort of the sphere. In the example, it is easy to see that correctness is violated since the handler H4, which is invoked for the critical exception E3, aborts the sphere.

If, in Step 5, no critical handler with an abort option is detected, the correctness of the sphere is validated.

8.3 Complexity of the Algorithm

We will now evaluate the complexity of the algorithm described above. To this end, we analyze the different steps.

- **Step 1.** It is easy to see that the size of the internal exception handling graph is linearly dependent on the number of tasks and exception handlers in the sphere since it has exactly these elements as nodes. Inserting the edges has a complexity which is bound by $n * m$, where n is the number of tasks and exception handlers and m is the number of edges listed in the internal exception handling relation, which we assume is represented as a table.
- **Step 2.** If the reverse traversal of the graph is performed breadth-first, each node has to be visited exactly once. Hence, the complexity of this step is linear.
- **Step 3.** All validations of Step 3 (atomicity, FLEX-structure, critical points) can be performed in one depth-first left-to-right traversal of the graph. Hence, the complexity is linear.
- **Step 4.** In this step, nodes and edges are removed and inserted based on information which can be found directly in the process descriptions. Hence, what is needed is a traversal of the graph and a lookup of task properties for each node. The complexity of this is bound by n^2 if n is the number of nodes in the graph.
- **Step 5.** This step requires a traversal of the external exception handling table and the generation of the external graph based on this table. The complexity is bound by $n^2 + m^2$, where n is the size of the table and m is the size of the exception information for the exception handlers. The size of the former, as well as the size of the latter, is bound by the number of exceptions in the system.

The considerations above show that the algorithm for the correctness validation of spheres described above has a complexity which is $\mathcal{O}(n^2)$, where n is the number of tasks, exceptions, and exception handlers used in the sphere.

8.4 Practical Considerations

The validation algorithm described above allows us to control the correctness of sphere definitions and is thus an

important tool for the construction of fault-tolerant processes. Process designers can invoke the algorithm to verify the correctness of their specifications and receive hints on possible improvements. These hints can be easily derived from the algorithm. In the simplest form, the algorithm could present the internal and external exception handling graphs to the process designers, giving them the chance to comprehend the decision of the system and allowing them to detect possible improvements themselves. A more sophisticated mechanism could make propositions for improvements autonomously. If, for instance, a critical exception is detected that leads to an exception handler which aborts the sphere, the system can provide the designer with a set of possible changes which include:

1. Changing the exception returned by the respective task or signal proxy,
2. Changing the termination options of the handler which incorrectly aborts the sphere, or
3. If a chain of multiple handler invocations leads to the abort, changing exceptions or termination options of one or more of the handlers in the chain.

The user can then select the most appropriate out of the options presented.

A general problem of the correctness notion underlying the algorithm is its restrictiveness. The algorithm evaluates the worst case, detecting all possible violations of atomicity. Such a violation can, however, be caused by an exception that is very unlikely to occur, which means that a process is rejected although the probability that atomicity will be violated is very small. For practical reasons, modifications to the algorithm are possible that take a more pragmatic approach.

1. Ignore exceptions in the validation process. The user can instruct the system to check only the normal process without considering the exception handling mechanism. Given that exceptions are unlikely to occur, this allows us to validate a basic level of correctness without the guarantee that the system can enforce atomicity in all cases. Instead of compile-time checks, the system can detect atomicity violations caused by exception handling at runtime and signal a specific exception in this case which calls for human intervention. Relaxing the mechanism in this way can be valuable during the test phase of a new process since it allows us to detect which violations are likely to occur.
2. Allow to ignore certain violations explicitly. Ignoring all correctness violations caused by exceptions may be a too relaxed a solution in some cases. Instead, an interactive mechanism is possible which presents the user with the correctness violations found, asking them to correct the failures, but permits us to install a process even if some of the failures are not fixed. This weakening of correctness has to be confirmed by the user explicitly and the runtime mechanisms described before have to be in place in order to detect violations at runtime.
3. Statistical support: Given that the process support system contains a history component which logs all relevant information during process execution and allows us to analyze the stored data, it is possible to provide statistical support for the decisions of the modeler. If, for instance, a process is composed of predefined components which have already been used in other processes, the system will have information about the probability of certain exceptions and the typical decisions of the exception handlers. This information can serve as the basis for deciding which correctness violations can be ignored (because they are very unlikely to occur) and which have to be corrected. The history information can also be used to gradually improve process specifications, for instance, by analyzing a process after a number of instances have been executed and finding exceptions which are raised very frequently. An exception that is raised often should be incorporated in the normal control flow. Likewise, branches in a process which are almost never executed can be changed into exceptions in order to keep the process as near to the actual application semantics as possible.

8.5 Related Work

Several research projects have focused on the integration of databases and nondatabases into distributed computing environments, including work on extended transaction models (ETM) in distributed object management (DOM) [12], [23], [24], where a framework for flexible transaction structures in workflows was developed, and the ConTracts project [48], which focused 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 [14] and [15], 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 toward flexible recovery has been done in the context of advanced transaction models [11], [19], [18], [24], [21], [49], [48]. In particular, the EXOTICA project [5] investigated the mapping of advanced transaction models to workflow description languages and showed how some of the concepts used in transaction management can be used in workflow environments. The approach presented here differs from this work mainly because of its strong focus on modeling language aspects and because we do not assume a transactional environment.

There is no support for exception handling in currently available commercial WFMS. The problem of recovery from activity failures has, however, been considered in a number of research projects. Leymann [38] introduces the notion of a *sphere of joint compensation*, which is a subset of a process' activities. If one activity of a sphere fails, the whole sphere has to be backed out (either by compensating each step that succeeded so far or by executing a special higher-level compensating activity). The special properties of the concept (spheres do not have to form a connected graph, and multiple spheres can overlap) leads to very complex and incomprehensible semantics. Eder and Liebhart [17] describe recovery facilities in WAMO, a research WFMS.

Processes are treated as *workflow transactions*. Component activities can be declared *vital* for a process. A running process is aborted and compensated if a vital activity fails. Failures are classified into two categories (system failures and semantic failures). A further distinction of exception types is not possible. Thus, the system does not support the specification of different handling strategies for different classes of failures. Resumption of failed processes is not possible.

It is important to note the differences between our notion of exception handling and the recent work on adaptive workflows and workflow evolution [35]. Our work focuses on the handling of expected exceptions and on the seamless integration of exception handling in the execution environment. Workflow evolution concepts go far beyond the application of programming language concepts to workflow systems. Instead, they try to cope with unexpected exceptions by making the workflow system adapt to new or changed environments. These issues are beyond the scope of our work.

The work on exception handling in programming languages [25], [40], [43], [47], [52] has provided the basics for our approach. While it provides mechanisms 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.

A problem we did not consider in this work is the correct handling of multiple exceptions that occur concurrently in multiple parts of a system and can only be handled together. Specification and failure resolution techniques that can be used for this case have been developed in the context of complex concurrent and real-time distributed systems [34], [42], [51]. They do mostly rely on a notion of *atomic actions* that is very similar to the notion of atomic spheres used in workflow contexts.

9 Conclusions

The solutions presented allow the elegant specification of fault-tolerant workflow processes and will, along with the corresponding system support, result in cleaner process specifications and less overhead when designing fault-tolerant workflow processes. 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 back-out ensure the necessary flexibility of backward navigation, while exception handlers guarantee forward progress.
- *Reusability.* 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 reused since exception handlers are registered with the system and can thus be applied in various processes. Note that this applies even in the case of semantic exceptions. While the definition of whether a given situation is regarded as a semantic exception or not is highly context-dependent, the handling will often be similar between cases so that reusability can still be exploited. 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.
- *Consistency.* The combination of exception handling and atomicity provides strong semantics for exception handling, as they are needed if database management systems and other transactional resources are to be integrated into a distributed process.

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.

We see the validation mechanism as an important contribution toward the rapid development of fault-tolerant processes since it provides process designers with the ability to verify the correctness of their specifications, freeing them from the need for lengthy simulations or test suites. In this regard, our validation mechanism can be seen as a first step toward more sophisticated verification mechanisms for distributed processes which are able to assess other properties such as resource utilization or temporal behavior.

References

[1] G. Alonso et al., "WISE: Business to Business E-commerce." *Proc. IEEE Ninth Int'l Workshop Research Issues on Data Eng.,* Mar. 1999.

[2] G. Alonso and C. Hagen, "Geo-Opera: Workflow Concepts for Spatial Processes," *Proc. Fifth Int'l Symp. Spatial Databases (SSD '97),* June 1997.

[3] G. Alonso et al., "Distributed Processing over Stand-Alone Systems and Applications," *Proc. 23rd Int'l Conf. Very Large Databases (VLDB '97),* 1997.

[4] G. Alonso et al., "Towards a Platform for Distributed Application Development," *Workflow Management Systems and Interoperability,* A. Dogac et al., eds., 1998.

[5] G. Alonso et al., "Advanced Transaction Models in Workflow Contexts," *Proc. Int'l. Conf. Data Eng.,* Feb. 1996.

[6] G. Alonso and C. Mohan, "Workflow Management: The Next Generation of Distributed Processing Tools," *Advanced Transaction Models and Architectures,* S. Jajodia and L. Kerschberg, eds., Kluwer Academic, 1997.

[7] G. Alonso and H.J. Schek, "Research Issues in Large Workflow Management Systems," *Proc. NSF Workshop Workflow and Process Automation in Information Systems,* A. Sheth ed., pp. 126–132 May 1996. http://optimus.cs.uga.edu:5080/activities/NSF-workflow/proceedings.html.

[8] T. Anderson and P.A. Lee, *Fault Tolerance—Principles and Practice.* Prentice Hall Int'l, 1981.

[9] P. Arnold, "Integration eines erweiterten Transaktionsmodells in ein Workflow-Modell," master's thesis, ETH Zürich, Institut für Informationssysteme, 1996.

[10] P.A. Bernstein and E. Newcomer, *Principles of Transaction Processing.* Morgan Kaufmann, 1997.

[11] Y. Breitbart et al., "Merging Application-Centric and Data-Centric Approaches to Support Transaction-Oriented Multi-System Work-Flows," *SIGMOD Record*, vol. 22, no. 3, Sept. 1993.

[12] A. Buchmann et al., "A Transaction Model for Active Distributed Object Systems," *Database Transaction Models for Advanced Applications*, A. Elmagarmid, ed., pp. 123–158, 1992.

[13] C. Canamero, "Validierung der Atomarität in Workflow- und Prozessunterstützungssystemen," master's thesis, ETH Zürich, 1998.

[14] Q. Chen and U. Dayal, "A Transactional Nested Process Management System," *Proc. 12th Int'l Conf. Data Eng. (ICDE '96)*, 1996.

[15] Q. Chen and U. Dayal, "Failure Handling for Transaction Hierarchies," *Proc. 13th Int'l Conf. Data Eng. (ICDE '97)*, 1997.

[16] C.T. Davies, "Data Processing Spheres of Control," *IBM Systems J.*, vol. 17, no. 2, pp. 179–198, 1978.

[17] J. Eder and W. Liebhart, "Workflow Recovery," *Proc. First IFCIS Int'l Conf. Cooperative Information Systems (CoopIS '96)*, June 1996.

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

[19] A.K. Elmagarmid et al., "A Multidatabase Transaction Model for Interbase," *Proc. Int'l Conf. Very Large Data Bases*, pp. 507–518, 1990.

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

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

[22] D. Georgakopoulos, M. Hornick, and A. Sheth, "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure," *Distributed and Parallel Databases*, vol. 3, no. 2, pp. 119–153, 1995.

[23] D. Georgakopoulos and M.F. Hornick, "A Framework for Enforcable Specification of Extended Transaction Models and Transactional Workflows," *Int'l J. Intelligent and Cooperative Information Systems*, Sept. 1994.

[24] D. Georgakopoulos, M.F. Hornick, and F. Manola, "Customizing Transaction Models and Mechanisms in a Programmable Environment Supporting Reliable Workflow Automation," *IEEE Trans. Knowledge and Data Eng.*, 1996.

[25] J.B. Goodenough, "Exception Handling: Issues and a Proposed Notation," *Comm. the ACM*, vol. 18, no. 12, pp. 683–695, Dec. 1975.

[26] The Open Group, *Distributed TP: The XA Specification*, Open Group Technical Standard, 1992.

[27] C. Hagen and G. Alonso, "Backup and Process Migration Mechanisms in Process Support Systems." Technical Report 304, ETH Zurich, Ins. of Information Systems, Aug. 1998. http://www.inf.ethz. ch/department/IS/iks/publications/ha98d.html.

[28] C. Hagen and G. Alonso, "Flexible Exception Handling in the OPERA Process Support System," *Proc. Int'l Conf. Distributed Computing Systems*, May 1998.

[29] C. Hagen and G. Alonso, "Beyond the Black Box: Event-Based Inter-Process Communication in Process Support Systems," *Proc. 19th Int'l Conf. Distributed Computer Systems (ICDCS 99)*, June 1999.

[30] C. Hagen and G. Alonso, "Highly Available Process Support Systems: Implementing Backup Mechanisms," *Proc. 18th IEEE Symp. Reliable Distributed Systems*, Oct. 1999.

[31] *Bulletin IEEE Technical Committee on Data Eng., Special Issue on Workflow and Extended Transaction Systems*, M. Hsu ed., IEEE Computer Soc., June 1993.

[32] *Bulletin IEEE Technical Comittee on Data Eng., Special Issue on Workflow Systems*, M. Hsu, ed., IEEE Computer Soc., Mar. 1995.

[33] S. Jablonski and C. Bussler, *Workflow Management.* Int'l Thomson Computer Press, 1996.

[34] P. Jalote and R.H. Campbell, "Atomic Actions for Software Fault Tolerance Using CSP," *IEEE Trans. Software Eng.*, vol. 12, no. 1, 1986.

[35] "Towards Adaptive Workflow Systems," *Proc. Computer Supported Cooperative Work (CSCW 98') Workshop*, M. Klein, C. Dellarocas, and A. Bernstein, eds. 1998. http://ccs.mit.edu/klein/cscw98/.

[36] *Dependability: Basic Concepts and Terminology*, J.C. Laprie ed., Springer-Verlag, 1992.

[37] Y. Leu, A.K. Elmagarmid, and N. Boudriga, "Specification and Execution of Transactions for Advanced Database Applications," *Information Systems*, vol. 17, no. 2, pp. 171–183, 1992.

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

[39] F. Leymann and W. Altenhuber, "Managing Business Processes as an Information Resource," *IBM Systems J.*, vol. 33, no. 2, pp. 326–348, 1994.

[40] D.L. Parnas, "Response to Detected Errors in Well-Structured Programs," technical report, Computer Science Dept, Carnegie-Mellon Univ., 1972.

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

[42] A. Romanowsky, "Pratical Exception Handling and Resolution in Concurrent Programs," *Computer Languages*, vol. 23, no. 1, 1997.

[43] R.W. Sebesta, *Concepts of Programming Languages*, third ed. Addison-Wesley, 1996.

[44] *Proc. NSF Workshop Workflow and Process Automation in Information Systems*, A.Sheth, ed., May 1996. http://optimus.cs.uga.edu:5080/activities/NSF-workflow/proceedings.html.

[45] G.L. Steele, *Common Lisp: The Language*, second ed. Digital Press, 1990.

[46] B. Stroustrup, *The C++ Programming Language*, second ed. Addison-Wesley, 1991.

[47] P. van Zee, M. Burnett, and M. Chesire, "Retire Superman: Handling Exceptions Seamlessly in a Declarative Visual Programming Language," *Proc. IEEE Symp. Visual Languages*, Sept. 1996.

[48] H. Waechter and A. Reuter, "The ConTract Model," *Transaction Models for Advanced Database Applications*, A. Elmagarmid, ed., pp. 219–263, Morgan-Kaufmann, 1992.

[49] G. Weikum and H.J. Schek, "Concepts and Applications of Multilevel Transactions and Open Nested Transactions," *Database Transaction Models for Advanced Applications*, A.K. Elmagarmid, ed., Morgan Kauffman, 1991.

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

[51] J. Xu, A. Romanowsky, and B. Randell, "Coordinated Exception Handling in Distributed Object Systems: From Model to System Implementation," *Proc. 18th Int'l Conf. Distributed Computing Systems (ICDCS '98)*, pp. 12–21, May 1998.

[52] S. Yemini and D.M. Berry, "A Modular Verifiable Exception-Handling Mechanism," *ACM Trans. Programming Languages and Systems*, vol. 7, no. 2, pp. 214–243, Apr. 1985.

Claus Hagen received his degree in computer science from the University of Erlangen-Nürnberg, Germany, in 1995, and his PhD degree in computer science from the Swiss Federal Institute of Technology (ETH), Zürich, in 1999. He is a senior information systems architect at Credit Suisse, a major Swiss bank. He is a member of the IEEE Computer Society.

Gustavo Alonso received a degree in telecommunications engineering from Madrid Technical University in 1989 and the MS and PhD degrees in computer science from the University of California at Santa Barbara in 1992 and 1994, respectivily. Previously, he was with the IBM Almaden Research Laboratory in San Jose, California. Currently, he is an assistant professor in the Department of Computer Science at the Swiss Federal Institute of Technology (ETH) in Zürich, and also leads the Information and Communication Systems Research Group. His research interests include cluster computing, databases, workflow management, scientific applications of database and cluster technology (earth sciences, astrophysics, and biology), transaction management, as well as replication, availability, and scalability in databases and clusters.