Error Handling in Process Support Systems

Fabio Casati[1] and Gianpaolo Cugola[2]

[1] Hewlett-Packard Laboratories,
1501 Page Mill Road, 1U-4, Palo Alto, CA, 94304, USA.
casati@hpl.hp.com
[2] Politecnico di Milano
Dipartimento di Elettronica e Informazione,
Via Ponzio 34/5, I-20133 Milan, Italy.
cugola@elet.polimi.it

Abstract. Process Support Systems (PSSs) are software systems supporting the modeling, enactment, monitoring, and analysis of business processes. Process automation technology can be fully exploited when predictable and repetitive processes are executed. Unfortunately, many processes are faced with the need of managing exceptional situations that may occur during their execution, and possibly even more exceptions and failures can occur when the process execution is supported by a PSS. Exceptional situations may be caused by system (hardware or software) failures, or may by related to the semantics of the business process.
In this paper we introduce a taxonomy of failures and exceptions and discuss the effect that they can have on a PSS and on its ability to support business processes. Then, we present the main approaches that commercial PSSs and research prototypes offer in order to capture and react to exceptional situations, and we show which classes of failure or exception can be managed by each approach.

1 Introduction

Mature business organizations are characterized by a high level of standardization in the set of activities carried out by their employees to pursue the organization's business mission (i.e., their *business processes*). This is true in particular for companies engaged in e-business activities, where business processes are executed in very high volumes and are automated for the most part. More human-oriented, flexible and dynamic organizations follow looser, adaptive process, but some notion of company-wide process still exists.

The quality of the business process affects the quality of the products and services delivered by the organization. As a consequence, in the last decades a lot of effort has been put in identifying techniques and methodologies to increase business process quality (and hence to provide better services at lower operating costs). In the area of information technology this efforts lead to the development of two different classes of tools: *Workflow Management Systems* (WfMSs) [1, 2] and *Process-centered Software Engineering Environments* (PSEEs) [3–5]. WfMSs

are oriented to supporting generic business processes, while PSEEs have been specially conceived to support software development processes.

It is interesting to observe that, even if they were developed by different communities (WfMSs by people originally working in the area of Information Systems and Databases, while PSEEs by people working in the area of Software Engineering) modern WfMSs and PSEEs share more commonalities than differences. In particular, they are affected by similar problems that initially limited their adoption. As a consequence, in this paper we will not distinguish among the two classes of tools (this is becoming a common approach in the last few years) and we will refer to both of them with the common term of *Process Support Systems* (PSSs).

PSSs support business organizations in modeling, automating, monitoring, and measuring their business process. Usually, a PSS provides a *Process Description Language* (PDL), used to develop a model of the business process. This model may be used to consolidate the process knowledge, to support process assessment, measurement, and refinement, to communicate the business rules within the organization, and, most importantly, to *automate* (i.e., *enact*) and *monitor* business process executions. During process model enactment, the PSS uses the rules and constraints expressed in the model to automate the activities that can be carried out without the intervention of human agents, and to guide and support people in carrying out the activities that require their intervention. Furthermore, PSSs offer tools to monitor and analyze process executions, in order to detect inefficiencies and hence improve the process.

Process coordination and automation technologies are becoming widespread in both e-businesses as well as in traditional enterprises, due to the need of reducing operating costs and performing high-quality services. While current PSS technology does contribute to achieving these objectives, it still lacks the flexibility and robustness needed to adapt to the rapidly evolving business and IT environment and to handle exceptional events that may occur during process enactment [6–8]. Exceptional events may range from failures in the underlying infrastructure to unforeseen changes in the external environment that require a deviation from the planned course of actions.

In this paper we classify exceptional events that may occur during process model enactment, we analyze the problems that these events may generate, and we describe the possible approaches to efficiently handle them, possibly with minimal or no human intervention.

2 Some Preliminary Definitions

As noted in Section 1, in the last decades two different communities have worked to similar problem developing different tools (i.e., WFMSs and PSEEs), but also a different terminology[1]. In this section we give some preliminary definitions

[1] This is demonstrated by the considerable time that the two authors, coming from these different communities, have spent in order to synchronize concepts and terminology

of the terms we will use in the remainder of the paper. We also give a quick overview of the basic architecture of a PSS.

Process Description Language (PDL). Each PSS provides a *Process Description Language* (PDL), used to develop a model of the business process.

Process model. It is a static description of the expected business process expressed in a PDL[2]. A process model is typically composed of *activities* (or *tasks*), that represent work items to be executed by a human or automated *resource*. In addition, the process model include the description of the execution dependencies among activities.

Observe that some PDLs allow process engineers to describe the expected process together with the activities to be pursued to cope with undesired (but foreseen) events. As an example some PDLs provide explicit linguistic constructs for exception handling (see Section 4 for further details on this topic). Here we use the name "process model" to indicate the model of the expected business process, without considering any undesired event.

Process Support System (PSS). It is a software application that supports the specification, automation, and monitoring of business processes. Typical examples of PSSs are Workflow Management Systems and Process-centered Software Engineering Environments.

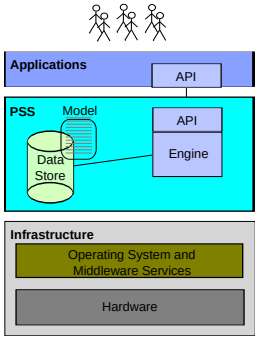

Fig. 1. The run-time architecture of a typical PSS.

A typical PSS (see Figure 1) is composed of an *engine*, which enacts a *process model* by scheduling activities and assigning them to the appropriate human or automated resource. A *Data Store* holds process definition and execution data as well as a description of the process resources. Usually, the data store is implemented by taking advantage of a DBMS, but PSSs exist that use the standard file system as their data store. Through an Application Programming Interface (API) the engine is able of controlling the execution of

[2] People working in the area of WFMSs often use the term *workflow schema*.

external applications, including the graphical user interface of the PSS itself. All these components run on top of a certain operating system and middleware, which provides advanced communication services to let the different components communicate and synchronize.

Observe that real PSSs have a much more complex architecture, including other components like process definition and monitoring tools, worklist managers, and so on. Sometimes they provide a distributed engine, or they use several DBMS distributed over a LAN to implement the data store. Since similar details are not relevant for the remainder of the paper they have not been included in Figure 1.

Actual process. It is the actual business process as it is performed in the real world. At each time instant, it may be described by the history of the activities that were performed to carry out the business process from the time it was started. It is a dynamic entity (i.e., it changes each time a new action is performed).

Observed process. During process model enactment, a PSS has a partial view of the actual process. The PSS, in fact, is only aware of the actions that the users perform under its control. All other actions are invisible to the PSS. This partial view of the actual process owned by the PSS is called the "observed process". At each time instant, it may be described by a history of the activities that the users performed under the PSS control to carry out the business process from the time it was started. It is the result of the enactment. Like the actual process, it is a dynamic entity.

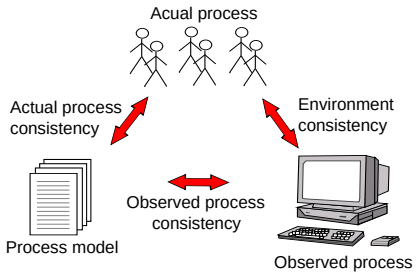

Fig. 2. The consistency relations among process model, actual process, and observed process.

Figure 2 shows the consistency relations that joins process model, actual process, and observed process when the process proceeds as expected. The actual process is consistent with the process model, i.e., it proceeds as described in the process model, without violating any of the constraints described there. Similarly, the observed process is consistent with the actual process. This means that the PSS has a correct view of the actual process. Finally, the observed process is

consistent with the process model, which means that the PSS is enacting the process model and it is not violating any of the constraints stated in the model. As discussed in the following section, this ideal situation is not easy to achieve and maintain.

3 Undesired Events and their Possible Effects

Business processes can be composed of many complex activities that need to be managed and synchronized, thus composing a very complex workflow that involves human and automated resources for a long time.

It is very common that during process execution something undesired happens. It may be something related to the underlying system, like a crash of a server, or a network fault; it may be something related to the PSS, like a crash in one of the applications invoked by the PSS to execute some process step; or it may be something related to the process itself, like a unforeseen situation that need to be managed before process could proceed. In all these cases the PSS must offer the right mechanisms to face the undesired event and to overcome it. Here we first classify the possible *undesired events*. Next, we show the techniques that can be leveraged to handle the different kinds of undesired events.

3.1 A Classification of Undesired Events

As a first initial classification it is useful to distinguish between *failures* and *exceptions*.

- Failures are system or network errors that originate from the PSS, from the applications invoked by the PSS, or from the underlying infrastructure on top of which the PSS is built and executed. As an example, they can be failures of the hardware that runs the PSS, crashes in the data store, or crashes of one of the applications controlled by the PSS engine. Failures affect the information system that supports the process.
- Exceptions are unexpected situations that are not part of the normal behavior of the process, and that require a deviation from the process model to be managed[3].

  Every time an event not captured by the process model occurs, we say that an exception has occurred. As an example, in some circumstances it may be necessary to change the order of execution of some activities, or to violate some timing constraints, or to let a unauthorized user perform a critical task because the person that was assigned to that task is not available.

[3] Observe that, as mentioned in Section 2, in this paper we do not consider the part of the process definition that copes with undesired (but foreseen) events as being part of the process model

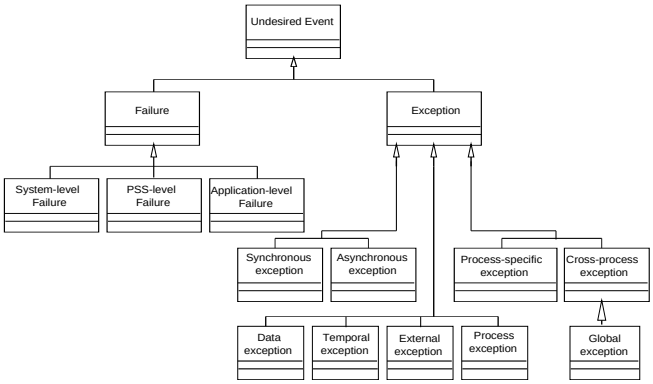

Fig. 3. A classification of undesired events.

A Taxonomy of Failures Depending on the layer of the PSS that originate the problem (see Figure 1), we distinguish among *infrastructure-level*, *PSS-level*, and *application-level* failures.

- Infrastructure-level failures originate from the hardware or from the operating system and middleware on top of which the PSS runs. Examples of this kind of failures are disk crashes, network faults, and operating system errors.
- PSS-level failures originate from the PSS. In particular, they may affect the PSS engine, the PSS data store, or the process model enacted by the PSS. Typical examples of this kind of failures are data store failures, or deadlocks, crashes of the PSS engine, or errors in the process model (e.g., a bug in the way an application is invoked or controlled through the engine API).
- Application-level failures affect the applications that run under control of the PSS to execute process steps. As an example, it may happen that an application invoked by the PSS engine crashes or it is not able, due to a bug, to save the results of the work the user has accomplished.

A Taxonomy of Exceptions Our analysis identified three main characteristics of exceptions, that have to be considered in understanding how they can be modeled: the *synchronicity*, the *scope*, and the *origin*.

Synchronicity. Exceptions may be *synchronous* or *asynchronous* with respect to the progression of the process. Synchronous exceptions occur at the start or completion of tasks and processes, while asynchronous ones may occur at any time during process execution. For instance, the output data of a task may return an unexpected null value. An example of asynchronous exception is instead the cancellation of an interview in a candidate interview process, which may happen at any time during the execution, and not only at the

start or completion of a task. Synchronous exceptions can further be characterized by their localization: *localized* exceptions may be only caused by the execution of one (or few) tasks, while *sparse* exceptions may be caused by the execution of several tasks in a process, and therefore may occur at several stages during the execution of the process. For instance, a data constraint violation exception is localized if only one task may modify those data, and sparse otherwise.

Scope. Exceptions may be *process-specific*, i.e. they may be related to one specific process instance, or they may be *cross-process*, i.e. they may be related to and affect several process instances. Interview cancellations and deadline expirations are examples of process-specific exceptions. Instead, a hiring freeze would be cross-process, since it would affect every running instances of the *candidate interview* process. Exceptions can span over process boundaries both in their detection and in their recovery:

- *detection*: the occurrence of the exceptional situation may depend on the state of several instances, such as when a customer rents two or more cars in overlapping periods.
- *recovery*: managing the exceptional situation may require actions in several process instances, such as in the car rental example discussed above.

A distinguished kind of cross-process exception is represented by *global* exceptions, i.e. anomalous, generic situations that may possibly affect every process, and for which the reactions may be defined at the PSS level. The unavailability of a resource is an example of a global exception. The appropriate reaction may be defined at the global (PSS) level and possibly refined for specific processes if different policies need to be adopted.

Origin. Exceptions may be classified according to what generates them:

Data exceptions are raised by modifications to process relevant data. For instance, a modification to the trip's cost may cause an overdraft on the customer's account.

Temporal exceptions are raised at the occurrence of a given timestamp (e.g. a deadline for a task), periodically (e.g. every night at 7pm), or as a defined interval has elapsed since a reference event (e.g., 20 minutes after a fire alarm).

External exceptions are explicitly notified to the process engine by humans or external applications. An email by the candidate requesting the cancellation of the interview is an example of external exception.

Process exceptions are raised by state changes in a process instance or task execution. For instance, the firing of an already active task may be perceived as an exceptional situation.

Figure 3 summarizes through a UML model [9] the resulting classification of undesired events.

3.2 Possible Effects of Undesired Events

Typically, the effect of an unmanaged failure or exception is an action that breaks the consistency relationships shown in Figure 2. This often leads to the

impossibility of continuing executing the business process under the control of the PSS.

To better clarify the possible effects of undesired events not adequately managed by the PSS, we introduce two new terms: we call *deviation* an action in the actual process that breaks one of the consistency relationships of Figure 2 and *inconsistency* the resulting state. Depending on the relationship broken we distinguish among (see Figure 4):

Actual process deviation. It is an action performed in the actual process that is not described in the process model or that violates some of the constraints expressed in the process model. Actual process deviations break the consistency relation between the actual process and the process model leading to an *actual process inconsistency*.

Observed process deviation. It is an action performed by the PSS that is not reflected in the process model. Observed process deviations break the consistency relation between the observed process and the process model, leading to an *observed process inconsistency*.

Environment deviation. [4] It is an action performed in the actual process or in the PSS that breaks the consistency relation between the actual process and the observed process. It typically occurs when someone performs an action that is relevant for the business process out of the PSS control. It leads to an *environment inconsistency*.
Observe that an environment inconsistency is definitely something to avoid. When an environment inconsistency occurs, the PSS has an incorrect or incomplete view of the actual process and, consequently, it cannot correctly support the actual process anymore.

Usually, actual process deviations are the result of an exception. To cope with an exception, in fact, the actual process must deviate from the process model, thus leading to an actual process deviation[5]. Moreover, since in general PSSs cannot deviate from the process model (i.e., observed process deviations are not possible), the effect of an actual process deviation is often an environment deviation. To avoid similar situations, it may be necessary for the PSS to deviate from the process model (i.e., to perform an observed process deviation), to continue mirroring the actual process even in presence of an actual process deviation. Model-relaxing approaches (see Section 4.2) can be adopted to pursue this goal.

4 Recovery Approaches

The ability to adequately manage undesired events without disruptions to the running business processes is crucial for modern PSSs that support mission-

[4] The name "environment deviation" follows from the common habit of calling "environment" the sum of the actual process plus the PSS supporting it [10].

[5] We observe again that we assume that the process model only includes the description of the normal behavior of a process. Mechanisms to specify exception handling behaviors as part of the process model will be introduced later in the paper.

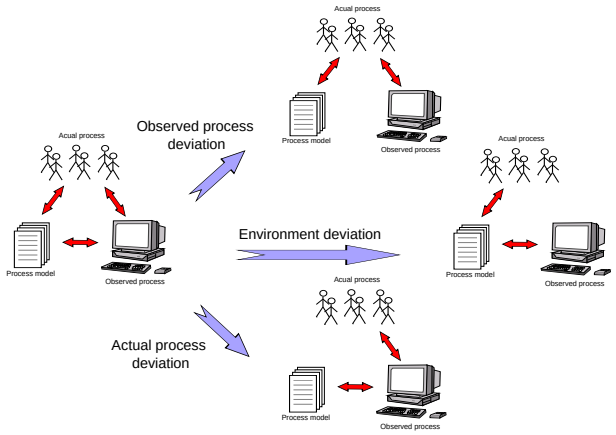

Fig. 4. Deviations and inconsistencies.

critical operations. Consequently, to be successful in the market, a PSS has to offer a set of mechanisms to react to the different classes of undesired events identified in Section 3.

In principle, different *recovery approaches* may be followed to react to failures and exceptions. We identify three possible classes of approaches, depending on the layer that implements them (see Figure 1): *infrastructure-level*, *PSS-level*, and *application-level* approaches. For its relevance, we also consider a fourth class of approaches that we call *process-model-level*. The next sections analyzes the different approaches.

4.1 Infrastructure and Application-Level Approaches

While exceptions are directly related to the existence of a PSS and can be adequately managed only by the PSS, failures (and particularly infrastructure and application-level failures) can be managed through general approaches involving the infrastructure and the application levels only.

Infrastructure and application-level approaches to failure handling have in fact been developed in several different contexts and most of them become mature years before the first PSS was developed. They include mechanisms like hardware redundancy to reduce the impact of hardware crashes, advanced network protocols and middleware mechanisms to cope with network faults, or software redundancy (e.g., using replicated servers) to cope with application failures.

While these mechanisms are designed to react either to infrastructure-level or to application-level failures, they provide little help to handle PSS-level failures and exceptions. In fact, these require a knowledge of the process model in order to be properly captured and managed.

Since this paper focuses on PSSs, analyzing these generic failure handling techniques in details is out of the scope of this work. Instead, in the following we focus on PSS and process-model-level approaches, described next.

4.2 PSS-Level Approaches

The PSS has a direct visibility of the process model and is responsible for scheduling tasks and for launching and managing applications, so it can provide powerful approaches to handle both failures and exceptions.

PSS-Level Failure-Handling Approaches. PSSs may implement mechanisms to cope with both infrastructure-level and application-level failures. As an example, the PSS may offer mechanisms to handle failures of the data store or of the middleware layer. Even PSS failures can be managed at the PSS level, by offering mechanisms to restore the last consistent state reached before the failure when the PSS is restarted (typically, this is obtained by taking advantage of a DBMS used to permanently store the PSS state).

In general, failure management at the PSS level implies the use of a combination of "undoing" and "redoing". As an example, network failures are often managed by retrying the communication until a success is reached, while to manage an application failure it is often necessary to undo the operations made by the activity that called the failed application, before trying to redo the entire activity. Similar approaches are adopted by a few research prototypes like SPADE [11–13], whose elementary tasks are implemented as transactions in the object-oriented DBMS that stores process instance data, thus allowing the engine to roll-back any task affected by a failure. A similar approach is adopted by Apel [14, 15].

The main problem with undoing and redoing in PSSs is that often the tasks that have to be undone and redone are very complex. They could have side effects that cannot be undone and they could involve activities that cannot be controlled by the PSS (e.g., manual tasks). In other word, undoing and redoing require a knowledge of the application domain that the PSS alone cannot have. This motivations induced researchers and practitioners to introduce process-model-level approaches, which involve process modelers in modeling the recovery process together with the standard process.

PSS-Level Exception-Handling Approaches. In Section 3 exceptions have been defined as unexpected events that were not planned in the process model. Exceptions usually result in actual process deviations that, if not adequately managed, may result in environment deviations.

Current PSSs adopt two different approaches to cope with the need of deviating from the process model to react to exceptions: either they provide mechanisms to change the process model on-the-fly (*model-changing approaches*), or they provide mechanisms to explicitly deviate from the model without the need of modifying it (*model-relaxing approaches*).

With the first approach the model is changed before the deviation occurs so neither an actual process deviation, nor a PSS deviation is required to face the exceptional event. With the second approach the PSS adopts some kind of "relaxed" interpretation of the model to continue mirroring the actual process even in presence of an actual process deviation. In practice, the PSS reacts to an actual process deviation by performing a PSS deviation, thus avoiding an environment deviation. Figure 5 compares the two approaches.

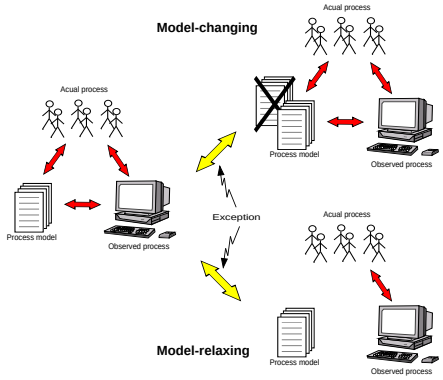

Fig. 5. PSS-level exception-handling approaches.

Model-Changing Approaches. In general, we may distinguish between:

- *Ad-hoc changes*, which are modifications applied to a single running process instance, which do not involve a change in the process model itself.
- *Bulk changes*, which refer instead to modifications of the process model collectively applied to a subset (or to all) the running instances of a process.

For instance, assume that a new agreement between Italy and the US requires Italian tourists traveling to the US to previously request and obtain a visa from the US consulate. If the travel reservation process (Figure 6-a) has not yet been modified according to the new law, then its execution does not lead to the successful completion of the business process.

Indeed, since the new law will affect several running instances (in principle all the running instances plus all the instances that will start in the future), a bulk approach is required. The travel reservation process model should be modified as described in Figure 6-b, where all travel reservations for Italian citizens traveling to the US will include a task for requesting the visa to the US consulate. Moreover, the process modeler has to choose how to manage currently running process model instances. In fact, although the modified process model of Figure 6-b can correctly support all new reservations, it may not be suited for completing the processing of reservations which are in progress. Intuitively,

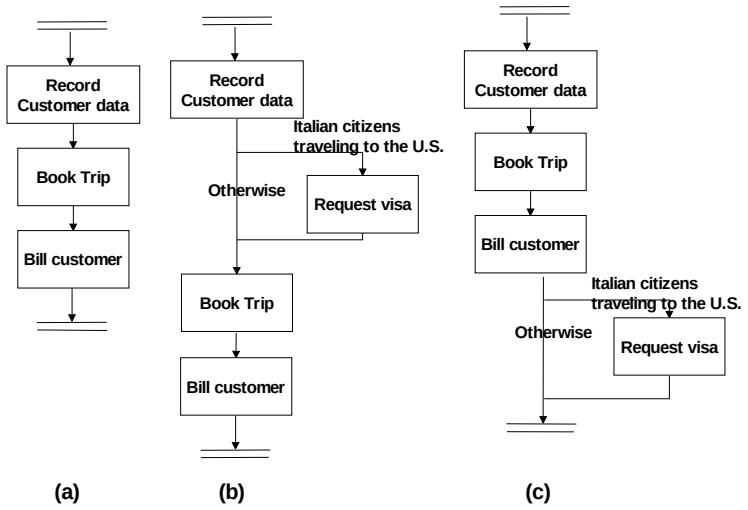

Fig. 6. The Travel Reservation Process. (a): initial version; (b): modified version; (c): ad-hoc version for managing instances that cannot be migrated to the correct version

all instances that are not concerned with Italian citizens going to the US or that are in their early stages (i.e., where task "record customer data" has not been completed yet) can *migrate* to follow the modified process definition shown in Figure 6-b. The other instances will have to be handled in an ad-hoc way, for instance by migrating to the process model shown in Figure 6-c, that still allows to achieve the goals of the process, although it is not the most appropriate and efficient way to execute it. In the context of PSS exceptions, *migrating* a process instance from a process definition (schema) S_1 to a schema S_2 means that the process engine will now schedule and assign tasks based on S_2 [16].

This and similar issues affecting bulk changes have been addressed by several research papers, such as [16–21, 11, 22]. Many of these approaches are based on the use of *migration rules*, to specify the future behavior of running instances once an exception has been detected. A migration rule identifies a subset of the running instances of a given process and specifies the schema to which instances in this subset should be migrated. For instance, in *eFlow* [23], rules have the form `IF <condition> THEN MIGRATE TO <model>`. The condition is boolean expression that identifies a subset of the running instances, while `<model>` denotes the destination schema. An example of migration rule is: `IF (task_state(Book_Trip)=not_started) THEN MIGRATE TO "New_travel_req"`. The set of rules must define a partition over the set of running instances, so that each instance is migrated to at most one schema. Instances that do not satisfy any rule condition are not migrated.

This idea of explicitly stating the rules to migrate process model instances with an ad-hoc language has been further extended with process-model approaches (see Section 4.3), which leverages reflective PDLs to allow process modelers to define, as part of the process model itself, when and how the model should be changed during its execution.

Ad-hoc changes could be seen as a particular kind of bulk changes, where only one instance is migrated. For instance, InConcert [24] allows the process responsible to manually reassign a task to a different resource or to start the execution of an arbitrary task in the flow.

A different approach is taken by Endeavors [25], a distributed, extensible PSS infrastructure, which allows the object-oriented definition and specialization of activities, artifacts, and resources associated with a software development process. In Endeavors, process models can be changed very easily even by standard users. These model changes affect the currently running instance and they may affect all the new instances, also. The easiness through which process models can be changed is presented as the main mechanism both to cope with unforeseen situations and to solve the problem of which running instance should be affected by a new process model. Since ad-hoc changes can be easily performed they can be used to migrate any running instance as required, while new instances will automatically use one of the modified process models as decided by the process modelers at launch time.

Model-Relaxing Approaches. The key idea behind model relaxing approaches is to allow the PSS to explicitly deviate from the process model to cope with unforeseen situations. PSSs adopting this approach provide flexible mechanisms for process model enactment that allows users to perform process tasks even if they violate some of the constraints stated in the model. As an example, in Sentinel [26] tasks are characterized by a set of preconditions that implicitly determine their ordering. Users are allowed to invoke a task even if one of its preconditions is not satisfied and the PSS is able to track this event and to mark the data items that could be possibly affected by this deviation, thus supporting users to analyze the effects of PSS deviations, if necessary.

A similar approach is adopted by Prosyt [6], while [27] and [28] present a goal-oriented language: PEACE, which formalizes parts of a process model using an autoepistemic logic, thus allowing users' beliefs to be modeled, and allowing the PSS to reason about the differences between the user beliefs and the actual process. A PEACE process model may describe a wide range of process states and transitions and the right transition may be chosen based on the actual beliefs of the PSS with respect to the actual process.

Model relaxing approaches are best suited to cope with unforeseen events that require an immediate answer by the system and that is most likely that will not occur again in the future. In similar situations, ad-hoc process model changing approaches could be adopted also, but they require much more effort in order to change the model as required. In fact, model changing is a time consuming activity that requires the intervention of the process modeler. It is

not reasonable to change the model each time some minor, unexpected event happens. In this situations, model relaxing approaches show all their power.

4.3 Process-Model-Level Approaches

The key idea behind process-model-level approaches is to involve process modelers, who have a precise knowledge of the application domain, in failure and exception handling. PSSs adopting these approaches give process modelers the chance to specify, as part of the process model, either the actions for undoing process tasks, or the actions to perform in case of failures and exceptions, or even the actions for modifying the process model itself when necessary.

In the first case, the weakness of standard PSS-level approaches to failure handling described in Section 4.2 are addressed by giving process programmers the ability of explicitly modeling application specific techniques for undoing or redoing critical tasks. In the second case, process modelers specify directly what to do when a undesired event occurs. In this case we talk of *expected exceptions* to refer to the set of predictable deviations from the normal behavior of the process whose handling has been explicitly coded has part of the model itself. This kind of undesired events are typically detected by the system and possibly managed with no human intervention. The need for the third approach comes from the consideration that traditional model-changing approaches do not offer a way to choose in an process-specific way the kind of allowed changes and the way these changes have to be applied.

Transactional Approaches As mentioned in Section 4.2, the main weakness of standard PSS-level approaches to failure handling is the difficulty of undoing complex business tasks without a precise knowledge of the process semantics. For instance, actions such as "send a letter" cannot be undone by restoring a previous database state. This problem can be managed by transactional approaches.

Transactional process models allow the definition of *regions* in the process model that should be executed atomically. At the request of the process responsible (or depending on the failure codes returned by task executions) the execution of the region can be aborted and rolled back to its entry point. Process execution can then be resumed by retrying the same path or by following an alternative route. Rollback of process regions is typically performed by executing a *compensating task* for each completed task, in the reverse order of their forward execution. A compensating task is an activity that semantically undoes the effect of another task. For instance, a task that reserves a car is compensated by a task that involves calling the car rental company to cancel the reservation. A similar model is for instance supported by *ConTracts* [29]. Several other PSSs implements similar approaches to failure handling. This section reviews the most relevant ones.

WAMO [30, 7], WIDE [8], TREX [31], and Crew [32] extend the above approaches by providing more flexibility in the backward compensation/forward execution process: they allow the definition, for each task, of the point to which

execution should be rolled back in case of failure and the specification of whether execution should be re-started or aborted from there; furthermore, based on the task properties or on predicates defined over process data, a task involved in a partial rollback and forward recovery may or may not be compensated or re-executed.

Commercial systems do not typically provide this kind of functionality. However, the Exotica project [33, 34] has developed a tool that provides process designers of IBM FlowMark (an earlier version of MQ Workflow [35]) with an extended process definition language, allowing the implementation of advanced transaction models such as sagas and flexible transactions. Specifications in the extended model are then translated into FDL (FlowMark Definition Language) by properly inserting additional "compensating" paths after each task, to be conditionally executed upon a task failure (captured by means of the task return code).

Transactional approaches are effective in handling synchronous exceptions. However, they lack the required flexibility for handling generic exceptional situations. In addition, they are restricted in the class of allowed reactions, since they basically only allow a partial or total process rollback: in many exceptional situations, rolling back a process instance is not the appropriate reaction, and it is an extreme and expensive solutions in terms of lost work. Finally, current approaches can only capture process and data events.

Approaches Based on Explicitly Modeling Exceptions. These approaches allow process designers to explicitly model how to capture and react to *predictable* deviations from the normal behavior of the process. By explicitly modeling them, these exceptional behaviors can be managed by the PSS without the need for human intervention. There are two main sub-categories in this approach: one is based on *event nodes*, while the other is based on *Event-Condition-Action rules*.

Event Node Approaches. In the event node approach, the process meta-model includes a particular kind of node, often called *event node* (or event step), which is able to capture asynchronous events and to activate the successor node when the event is detected. There are many variations of this approach, depending on the types of events that can be captured and on how they can affect the process in which the task is defined. *Staffware* [36] and *eFlow* [23] are examples of PSSs that use this approach.

Event nodes can typically capture several types of events (for instance, in the case of *eFlow* they can capture all four kinds of originating events defined in Section 3.1), they can specify filter over (exceptional) events of interest, and they can capture event parameters into process variables. Hence, they are capable of *capturing* the exception. In the event node approach, the reaction is performed by activating a path in the process flow (the path connected in output from the event node). For instance, a node capturing candidate withdrawals in a candidate hiring process is an example of exception handling achieved through event nodes. However, as mentioned in the taxonomy of exception, this is not the only way

an exception can be handled: in fact, an exception often requires a partial or global rollback of the process instance, or the notification of the problem to a selected user. In addition, process instances may have to be suspended in order to properly handle the exception. Hence, in order to be fully effective, the event node approach requires the PSS to provide specialized tasks that enable the specification of these kinds of behaviors.

A limitation of the event node approach is that every exception requires the definition of an event node (that captures the exceptional situation) and of the tasks that perform the corrective actions. Hence, when many exceptions need to be managed, the process model may become very complex. Consider for instance exceptions related to deadline expirations. If deadlines need to be specified for each node (as it often happens), then the process model becomes unmanageable. In general, event nodes are effective in managing all types of exceptions except cross-process and global ones. In fact, they can capture synchronous and asynchronous events; however, they must be specified within a process model (it is not possible in current PSSs to define "global" event nodes).

Rule-Based approaches. A few commercial systems and research prototypes (e.g., COSA [37], InConcert [24], WIDE [8] and ADOME-WfMS [38]) provide a rule-based language for the specification of exceptional behaviors. Rules typically take the form of event-action or event-condition-action statements, where the event defines when the rule should be activated, the condition (if present) verifies that the occurred event actually corresponds to an exception that must be managed, while the action handles the exception, by invoking the primitives provided by the rule language.

In order to exemplify these concepts, we now show how exceptional behaviors are specified by means of rules in the WIDE PSS (the other systems follow a similar or simplified model). The WIDE process definition language includes a rule language for defining expected exceptions, called *Chimera-Exc* [8]. In Chimera-Exc rules, triggering events belong to one of the four classes mentioned above (data, temporal, external, or process); the condition is a predicate over process data whose evaluation determines whether the action part should be executed or not, and may in addition return some bindings, passed to the action part in order to target the reaction over specific objects; finally, the actions can send notifications to selected resources, start, suspend or terminate the execution of tasks and process instances, reassign tasks to different resources, or rollback process instances. For example, the rule *negativeBalance* shown below is activated as the value of variable *balance* is modified in an *accountMgmt* process, and notifies a process responsible if the resulting balance is negative.

```
define trigger negativeBalance for accountMgmt
 events     modify(accountMgmt.balance)
 condition  accountMgmt(A), A.balance<0,
            occurred (modify(accountMgmt.balance), A),
 actions    notify (A.responsible, "Overdraft for customer" +
                    A.customerName)
end
```

With the rule-based approach, exception handling strategies can be defined at different levels of abstraction. For instance, a process definition language could allow rules to be defined at the task, process, or PSS level. Rules associated to a given task are triggerable only when the task is active; rules associated with a given process are triggerable when the instance is active, while global rules are always active. This kind of structuring allows the definition of exception handling strategies at the global level, valid for all processes and tasks. Exceptional behaviors can then be overridden (or integrated) at the local level. In general, many variations of the rule-based approach are possible, depending on the expressive power of the event, condition, and action language, and on the rule invocation and execution semantics.

In general, the rule-based approach is very powerful. It can handle both synchronous and asynchronous events, it can manage global and local exceptions, it can capture different kinds of originating events, and can handle localized as well as sparse exceptions. Its drawback lies in the intrinsic complexity of the rule language: in fact, it is yet another formalism that is needed to fully specify the process behavior, and its semantic is very subtle, so that it is easy to generate unforeseen and undesired behaviors.

Approaches Based on Explicit Modeling of the Meta Process The main weakness of generic model-changing approaches is their lack of flexibility and the impossibility of accurately controlling the expressive power of model changing. In general, the kind of changes allowed to a process model and the way these changes have to be applied and deployed are process-specific and cannot be specified one for all. *Reflective PDLs* solve this problem.

Generally speaking, reflective languages give programmers the ability of specifying how programs have to be changed at run-time. In process programming this means adding to the PDL some special constructs that allow process modelers to specify when and how the model have to be changed. We say that process programmers are able of specifying the *meta-process* as part of the process itself [21].

Several PSS adopt a reflective language. SPADE adopts a PDL called SLANG [39], which is based on Petri-Nets. SLANG allows both running process model instances and the process model itself to be treated as process data items that may be modified by the PSS in the way specified by the running process model. Similarly, OASIS [22] provides an object-oriented, reflective framework for the definition, customization, and evolution of software process meta-models and of the software process models that are their instances. In developing this framework, the authors started from the consideration that every process modeling approach relies on some specific set of abstractions that define a process meta-model. They observed that the ability to provide a uniform model of both the process model and the process meta-model is essential to capture complex processes and to manage the customization and evolution of process models and their meta-models.

5 Conclusions

In this paper we have discussed the problem of managing failures and exceptions in business processes. We have presented a taxonomy that classifies the different kinds of exceptional situations that may occur during PSS-supported process executions, and we have shown approaches that enable their handling, possibly with minimal or no human intervention. Exception handling techniques have been mostly developed in the academia, and are recently starting to be included in commercial PSSs that support mission-critical applications. In particular, they are increasingly needed and leveraged in e-business applications.

We expect that many of the techniques presented in this paper will be applied to the area of *e-services*, and specifically to e-service composition. Frameworks and platforms for developing and managing (composite) e-services will be the next battleground for large software vendors such as Sun, Microsoft, IBM, BEA, and HP. Indeed, the adaptation of PSS technology to support the robust and reliable composition of e-services is often named as one of the main opportunity for such vendors in order to achieve competitive advantage. The ability of automatically handling failures and exceptions will be paramount in this area, due to need of supporting high volume, low cost, and zero latency service delivery.

References

1. D. Georgakopoulos, H. Hornick, and A. Sheth, "An overview of workflow management: from process modeling to workflow automation infrastructure," *Distributed and Parallel Databases*, vol. 3, 1995.
2. H. Stark and L. Lachal, *Ovum Evaluates: Workflow.* Ovum ltd., September 1995.
3. A. Finkelstein, J. Kramer, and B. Nuseibeh, eds., *Software Process Modelling and Technology.* Research Studies Press Limited (J. Wiley), 1994.
4. A. Fuggetta and C. Ghezzi, "State of the art and open issues in process-centered software engineering environments," *Journal of Systems & Software*, vol. 26, July 1994.
5. V. Ambriola, R. Conradi, and A. Fuggetta, "Assessing process-centered environments," *ACM Transactions on Software Engineering and Methodology*, vol. 6, July 1997.
6. G. Cugola, "Tolerating deviations in process support systems via flexible enactment of process models," *IEEE Transactions on Software Engineering*, vol. 24, November 1998.
7. J. Eder and W. Liebhart, "Contributions to exception handling in workflow management," in *Proceedings of the EDBT Workshop on Workflow Management Systems*, (Valencia, Spain), Mar. 1998.
8. P. Grefen, B. Pernici, and G. Sanchez, *Database Support for Workflow Management: the WIDE Project.* Kluwer Academic Publishers, 1999.
9. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.
10. G. Cugola, E. Di Nitto, A. Fuggetta, and C. Ghezzi, "A framework for formalizing inconsistencies in human-centered systems," *ACM Transactions On Software Engineering and Methodology (TOSEM)*, vol. 5, July 1996.

11. S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza, "SPADE: an environment for Software Process Analysis, Design, and Enactment," in *Software Process Modelling and Technology* (A. Finkelstein, J. Kramer, and B. Nuseibeh, eds.), Research Studies Press Limited (J. Wiley), 1994.
12. S. Bandinelli, A. Fuggetta, C. Ghezzi, and S. Grigolli, "Process Enactment in SPADE," in *Proceedings of the Second European Workshop on Software Process Technology*, (Trondheim (Norway)), Springer-Verlag, September 1992.
13. S. Bandinelli, M. Braga, A. Fuggetta, and L. Lavazza, "The architecture of the SPADE-1 process-centered SEE," in *Proceedings of the 3rd European Workshop on Software Process Technology*, LNCS 772, (Villard de Lans (Grenoble), France), February 1994.
14. S. Dami, J. Estublier, and M. Amiour, "Apel: a graphical yet executable formalism for process modeling," in *Process Technology* (E. Di Nitto and A. Fuggetta, eds.), Kluwer Academic Publishers, January 1998.
15. J. Estublier, P. Y. Cunin, and N. Belkhatir, "Architectures for process support system interoperability," in *5th International COnference on Software Process*, (Chicago, Illinois, USA), pp. 137–147, June 1998.
16. F. Casati, S. Ceri, B. Pernici, and G. Pozzi, "Workflow Evolution," *Data and Knowledge Engineering*, vol. 24, pp. 211–238, Jan. 1998.
17. F. Casati, *Models, Semantics, and Formal Methods for the Design of Workflows and Their Exceptions*. PhD thesis, Politecnico di Milano - Dipartimento di Elettronica e Informazione, Milano, Italy, Dec. 1998.
18. S. Ellis, K. Keddara, and G. Rozenberg, "Dynamic change within workflow systems," in *Proceedings of the ACM Conference on Organizational Computing Systems (COOCS '95)*, (Milpitas, California), 1995.
19. C. Liu, M. Orlowska, and H. Li, "Automating handover in dynamic workflow environments," in *Proceedings of the 10th International Conference on Advanced Information Systems Engineering CAiSE'98*, (Pisa, Italy), June 1998.
20. M. Reichert and P. Dadam, "ADEPT$_{flex}$ - supporting dynamic changes of workflows without losing control," *Journal of Intelligent Information Systems*, vol. 10, pp. 93–129, Mar. 1998.
21. S. Bandinelli, A. Fuggetta, and C. Ghezzi, "Process model evolution in the SPADE environment," *IEEE Transactions on Software Engineering*, vol. 19, December 1993.
22. P. Jamart and A. van Lamsweerde, "A reflective approach to process model customization, enactment, and evolution," in *Proceedings of the Third International Conference on the Software Process (ICSP3)*, (Reston, Virginia), pp. 21–32, IEEE Computer Society Press, October, 10-11 1994.
23. F. Casati, L. Jin, S. Ilnicki, and M. Shan, "eflow: an open, flexible, and configurable system for service composition," in *Proceedings of the Workshop on E-Commerce and Web Information Systems*, (Milpitas, CA, USA), June 2000.
24. R. Marshak, "InConcert workflow," Tech. Rep. 20,3, Workflow Computing Report, Patricia Seybold Group, 1997.
25. G. A. Bolcer and R. N. Taylor, "Endeavors: A process system integration infrastructure," in *Proceedings of the Fourth International Conference on Software Process (ICSP4)*, (Brighton, UK), December 2-6 1996.
26. G. Cugola, E. Di Nitto, C. Ghezzi, and M. Mantione, "How to deal with deviations during process model enactment," in *Proceedings of the 17th International Conference on Software Engineering*, (Seattle (Washington - USA)), April 1995.

27. S. Arbaoui and F. Oquendo, "Peace: Goal-oriented logic-based formalism for process modelling," in *Software Process Modelling and Technology* (A. Finkelstein, J. Kramer, and B. Nuseibeh, eds.), Research Studies Press, J. Wiley, 1994.
28. S. Arbaoui and F. Oquendo, "Managing inconsistencies between process enactment and process performance states," in *Proceedings of the 8th International Software Process Workshop*, (Wadern (Germany)), March 1993.
29. A. Reuter, K. Schneider, and F. Schwenkreis, "Contracts revisited," in *Advanced Transaction Models and Architectures* (S. Jajodia and L. Kerschberg, eds.), New York: Kluwer Academic Publishers, 1997.
30. J. Eder and W. Liebhart, "The Workflow Activity Model WAMO," in *Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIs'95)*, (Wien, Austria), May 1995.
31. R. van Stiphout, T. D. Meijler, A. Aerts, D. Hammer, and R. le Comte, "TREX: Workflow transaction by means of exceptions," in *Proceedings of the EDBT Workshop on Workflow Management Systems*, (Valencia, Spain), Mar. 1998.
32. M. Kamath and K. Ramamritham, "Failure handling and coordinated execution of concurrent workflows," in *Proceedings of the 14th International Conference on Data Engineering(ICDE'98)*, (Orlando, FL, USA), Feb. 1998.
33. G. Alonso, M. Kamath, D. Agrawal, A. E. Abbadi, R. Gunthor, and C. Mohan, "Failure handling in large scale workflow management systems," Tech. Rep. RJ9913, IBM Almaden Research Center, Nov. 1994.
34. G. Alonso, D. Agrawal, A. E. Abbadi, M. Kamath, R. Gunthor, and C. Mohan, "Advanced transaction model in workflow context," in *Proceedings of the 12th International Conference on Data Engineering(ICDE'96)*, (New Orleans, LA, USA), Feb. 1996.
35. IBM, *MQ Series Workflow - Concepts and Architectures*, 1998.
36. Staffware Corporation, *Staffware Global - Staffware for Intranet based Workflow Automation*, 1997. Available at http://www.staffware.com/home/whitepapers/data/globalwp.htm.
37. Baan Company N.V. - COSA Soultions, *COSA Reference Manual*, 1998.
38. D. Chiu, K. Karlapalem, and Q. Li, "Exception handling with workflow evolution in "adome-wfms": a taxonomy and resolution techniques," in *Proceedings of the First Workshop on Adaptive Workflow Systems*, (Seattle, Washington, USA), Nov. 1998. Available at http://ccs.mit.edu/klein/cscw98/paper06.
39. S. Bandinelli, E. Di Nitto, and A. Fuggetta, "Supporting cooperation in the spade-1 environment," *IEEE Transactions on Software Engineering*, vol. 22, December 1996.