TECHNICAL REPORT

A Knowledge-Based Approach to Handling Exceptions in Workflow Systems

by

Mark Klein

CCS WP #203  SWP # 4034
April 1998

CENTER FOR COORDINATION SCIENCE

Massachusetts Institute of Technology
Sloan School of Management
Cambridge, Massachusetts
A Knowledge-Based Approach to Handling Exceptions in Workflow Systems

by

Mark Klein

CCS WP #203  SWP # 4034

April 1998
A Knowledge-Based Approach to Handling Exceptions in Workflow Systems

Mark Klein  
Center for Coordination Science (CCS)  
MIT Sloan School of Management  
One Amherst Street E40-169  
Cambridge MA 02139 USA  
+1 (617) 253-6796  
m_klein@mit.edu

ABSTRACT

This paper describes a novel knowledge-based approach for helping workflow process designers and participants better manage the exceptions (deviations from an ideal collaborative work process caused by errors, failures, resource or requirements changes etc.) that can occur during the enactment of a workflow. This approach is based on exploiting a generic and reusable body of knowledge concerning what kinds of exceptions can occur in collaborative work processes, how these exceptions can be detected, and how they can be resolved. This work builds upon previous efforts from the MIT Process Handbook project and from research on conflict management in collaborative design.

Keywords

process exception handling, re-design, adaptation

THE CHALLENGE

A critical challenge for workflow systems is their ability to respond effectively when "exceptions" occur (Strong 1992); (Suchman 1983); (Grudin 1994); (Mi and Scacchi 1991); (Karbe and Ramsberger 1990) (Kreiefelts and Woetzel 1987) (Chiu, Karlapalem et al. 1997). We can consider an exception to be any departure from a process that achieves the process goals completely and with maximum efficiency. Exceptions can arise from changes in resources, organizational structure, company policy, task requirements or task priority. They can also include incorrectly or tardily performed tasks, resource contentions between two or more distinct processes, unanticipated opportunities to merge or eliminate tasks, conflicts between actions taken in different process steps and so on.

Exceptions can be frequent and extremely disruptive (Saastamoinen 1995). They often are not detected until some task actually becomes late, at which point they are typically handled as “fires”, are kicked up to higher management layers for resolution, and can cause cascading exceptions as normal work is shoved aside. Exceptions often do not have standardized preferred processes for handling them so they can be addressed inconsistently and with uneven effectiveness. If not detected and handled effectively, exceptions can thus result in severe impacts on the cost and schedule performance of process enactment systems.

Workflow systems are currently ill-suited to dealing with exceptions. These systems typically institutionalize a more or less idealized preferred process. When exceptions do occur we are often forced to “go behind the workflow system’s back”, making the system more of a liability than an asset. Workflow models can, of course, include conditional branches to deal with anticipated exceptions. Current process modeling methodologies and tools ((Grover and Kettinger 1995) (Harrington 1991)
(Kettinger, Guha et al. 1995) (Davenport 1993) (Hammer and Champy 1993)) do not, however, make any provision for describing exception handling procedures separately from "main-line" processing. Inclusion of exception handling branches, therefore, can greatly complicate process models and obscure the "preferred" process, making it difficult to define, understand, and modify. Up-front prescription of exception handling can also reduce or eliminate the discretion workflow participants in precisely the cases most likely to profit from individual attention. Current workflow modeling methods provide, in addition, no support for uncovering what kinds of exceptions can occur in a given process model, and how they can be resolved.

This paper describes a knowledge-based approach to meeting these challenges. The sections below will discuss how this approach works, what it contributes to previous research in this area, and how we plan to extend this work in the future.

A KNOWLEDGE-BASED APPROACH TO EXCEPTION HANDLING

The approach described here integrates and extends two long-standing lines of research: one addressing coordination science principles about how to represent and utilize process knowledge, another addressing how artificial intelligence techniques can be applied to detecting and resolving conflicts in collaborative design settings:

One component is a body of work pursued over the past five years by the Process Handbook project at the MIT Center for Coordination Science (Malone, Crowston et al. 1993; Dellarocas, Lee et al. 1994; Malone and Crowston 1994) (Malone, Crowston et al. 1997). The goal of this project is to produce a repository of process knowledge and associated tools/techniques that help people to (among other things) better redesign organizational processes, learn about organizations, and automatically generate software. The Handbook database continues to grow and currently includes over 4000 models covering a broad range of business processes. We have developed a mature Windows-based tool for editing the Handbook database contents, as well as a Web-based tool for read-only access. Both are being actively used by a highly distributed set of scientists, students and sponsors from government and industry. A key insight from this work is that a repository of business process templates, structured as a taxonomy, can help people design qualitatively more innovative processes more quickly by allowing them to retrieve, contrast and customize interesting examples, make "distant analogies", and utilize "recombinant" (mix-and-match) design techniques (Herman, Klein et al. 1998).

The other key component of this work is nearly a decade of development and evaluation of systems for handling multi-agent conflicts in collaborative design (Klein 1989; Klein 1991; Klein 1993) and collaborative requirements capture (Klein 1997). This work resulted in principles and technology for automatically detecting, diagnosing and resolving design conflicts between both human and computational agents, building upon a knowledge base of roughly 300 conflict types and resolution strategies. This technology has been applied successfully in several domains including architectural, local area network and fluid sensor design. A key insight from this work is that design conflicts can be detected and resolved using a knowledge base of generic and highly reusable conflict management strategies, structured using diagnostic principles originally applied to medical expert systems. Our experience to date suggests that this knowledge is relatively easy to acquire and can be applied unchanged to multiple domains.

The work described in this paper integrates and extends these two lines of research in an innovative and, we believe, powerful way. The central insights underlying this integration are that (1) workflow exceptions can be handled by generalizing the diagnostic algorithms and knowledge base underlying design conflict management (a conflict, after all, is just one subclass of process exception), and (2) the exception handling knowledge base can be captured as a set of process templates that can be retrieved,
compared and customized using the principles embodied in the Process Handbook. The result of this integration is an approach that allows workflow designers and participants to better take advantage of insights collected from a wide range of experts and domains when trying to determine what exceptions can occur in their process, as well as how such exceptions can be detected, diagnosed and resolved. These points will be discussed in detail in the following sections.

**Detecting Exceptions**

The first step is for a workflow designer to determine, for a given "ideal" workflow, the ways that the process may fail and then "instrument" the workflow so that these failures can be detected. This can be done via inheritance of failure modes down a process taxonomy.

A process taxonomy can be defined as a hierarchy of process templates, with very generic processes at the top and increasingly specialized processes below. Each process can have attributes, e.g. that define the challenges for which it is well-suited. Note that process specialization is different from decomposition, which involves breaking a process down (i.e. "decomposing it") into subactivities. While a subactivity represents a part of a process; a specialization represents a "subtype" or "way of" doing the process (Malone, Crowston et al. 1997).

Consider as an illustration a small process taxonomy consisting of the following templates (Figure 1):

![Figure 1. An Example of a Generic Process Taxonomy with Failure Modes.](image)

These templates are annotated with the ways in which they can fail, i.e. with their characteristic exception types. Our work to has revealed that a wide range of exception types exist (Klein 1997). Exceptions result, in general, from violations of some (implicit or explicit) assumption underlying a workflow model and can include changes in resources, organizational structure, policies, task requirements or task priority. They can also include incorrectly performed tasks, resource contentions between two or more distinct processes, unforeseen opportunities to merge or eliminate tasks, conflicts between actions taken in different process steps and so on.

Failure modes for a given process template can be uncovered using failure mode analysis (Raheja 1990). It is typical, for example, for process steps to require outputs produced by other steps. The processes for managing such ifow dependencies need to make sure that the right thing gets to the right place at the right time (Malone and Crowston 1994). This immediately implies a set of possible failure modes including an input being late (i.e. wrong time), of the wrong type (i.e. wrong thing) and so on. Similar analyses have been done for other process templates, such as resource sharing, diagnosis, order fulfillment, and so on.
We are now ready to see how the failure modes for a given workflow process can be identified. Consider the following process:

![Workflow Diagram]

**Figure 2. An Example Workflow Process.**

This workflow consists of a subprocess for allocating design tasks (performed in this case by a human manager), a subprocess for allocating shared resources such as mainframe computer time to design groups (handled on a first-come first-serve basis), followed by subprocesses where the different subcomponent designs are consolidated and then sent on to be manufactured, delivered, and inspected by the customer.

To identify failure modes we need only identify the generic process templates that match (components of) the workflow model. The potentially applicable exception types will then consist of the union of the failure modes inherited from the matching templates. We can see, for example, that the “distribute shared design resources” subprocess in Figure 2 is a subtype of the generic “pull-based sharing” process template in Figure 1, since the resources are “pulled” by their consumers rather than “pushed” (i.e. allocated) by their producers. This template includes among its characteristic failure modes the exception called “poaching”, wherein resources go disproportionately to lower-priority tasks because agent(s) with lower priority tasks happen to reserve them first. The “consolidate sub-designs” subprocess is a specialization of the “manage fit” template and thereby inherits the “design conflict” failure mode. The “deliver product” subprocess is a specialization of the “manage flow” template, with characteristic exceptions such as “item delayed”, “item misrouted” and so on. All the subprocesses also inherit the characteristic failure modes from the generalizations of these matching templates, such as “responsible agent is unavailable”, and so on.

The workflow designer can select, from this list of possible exception types, the ones that seem most important in his/her particular context. He/she might know, for example, that the “deliver product” process is already highly robust and that there is no need to augment it with additional exception handling capabilities.

For each exception type of interest, the workflow designer can then decide how to augment the workflow models in order to detect these exceptions. While processes can fail in many different ways, such failures have a relatively limited number of different manifestations, including missed deadlines, violations of artifact constraints, exceeding resource limits, and so on. Every exception type includes pointers to “exception detection” process templates in the Handbook repository that specify how to detect the symptoms manifested by that exception type. These templates, once interleaved into the workflow by the workflow designer, play the role of isentinels” that check for signs of actual or
impending failure. The template for detecting the “resource poaching” exception, for example, operates by comparing the average priority of tasks that quickly receive shared resources against the average priority of all tasks. The “item delayed”, “agent unavailable”, and “item misrouted” exceptions can all be detected using time-out mechanisms. The “design conflict” exception type can be detected by techniques such as constraint propagation and geometric feature overlap detection, depending on the nature of the conflicts being looked for.

**Diagnosing Exceptions**

The next step is to figure out how to react when an exception actually occurs during the enactment of the workflow process. Just as in medical domains, selecting an appropriate intervention requires understanding the underlying cause of the problem, i.e. its *diagnosis*. A key challenge here, however, is that the symptoms revealed by the exception detection processes can suggest a wide variety of possible underlying causes. Many different exceptions (e.g. “agent not available”, “item misrouted” etc.) typically manifest themselves, for example, as missed deadlines.

We have found that a heuristic classification approach (Clancey 1984) is well-suited to this challenge. This approach works by traversing a diagnosis taxonomy. Exception types can be arranged into a taxonomy ranging from highly general failure modes at the top to more specific ones at the bottom; every exception type includes a set of defining characteristics that need to be true in order to make that diagnosis potentially applicable to the current situation (Figure 3):

![Figure 3. A Subset of the Exception Type Taxonomy.](image)

When an exception is detected, the responsible workflow participant traverses the exception type taxonomy top-down like a decision tree, starting from the diagnoses implied by the manifest symptoms and iteratively refining the specificity of the diagnoses by eliminating exception types whose defining characteristics are not satisfied. Distinguishing among candidate diagnosis will often require that the user get additional information about the current exception and its context, just as medical diagnosis often involves performing additional tests.

Imagine, for example, that we have detected a time-out exception in the “transport product” step. The diagnoses that can manifest this way include “agent unavailable”, “item misrouted”, and “item delayed”. The defining characteristics of these exceptions are:

- **agent unavailable**: agent responsible for task is unavailable (i.e. sick, on vacation, retired ...)
- **item misrouted**: current location and/or destination of item not match original target destination
- **item delayed**: item has correct target destination but is behind original schedule
The user then has a specific set of questions that he/she can ask in order to narrow down the exception diagnosis. If the appropriate information is available on-line, then answering such questions and thereby eliminating some diagnoses can potentially be automated.

We have found that a relatively small set of question types get used again and again when describing the defining characteristics for different exception types. Examples include questions about the status of a task, the status of a resource, the rationale for a task (e.g. its underlying goals) and so on. We have formalized this set of questions into what we call the “query language” (Klein 1989) (Klein 1993). We are working towards the goal of defining a fully capable query language that will be simple enough to allow substantial automation of the exception diagnosis process, i.e. where most or all questions are answerable by software systems. This has the advantage of reducing the “cost of admission” for computer-based agents as participants in robust workflow systems; these agents need not have sophisticated individual exception handling capabilities but need merely be able to respond to a basic set of queries (an “action language”, as we shall see, will also be necessary).

Heuristic classification is a "shallow model" (Chandrasekaran and Mittal 1983) form of diagnosis because it is based on compiled empirical and heuristic expertise rather than first principles. This approach is appropriate for domains, such as medical diagnosis, where complete and consistent behavioral models do not exist. This, I would argue, is also true for workflows with human and complex software agents. An important characteristic of heuristic classification is that the diagnoses represent hypotheses rather than guaranteed deductions: multiple diagnoses may be suggested by the same symptoms, and often the only way to verify a diagnosis is to see if the associated prescriptions are effective.

Resolving Exceptions
Once an exception has been detected and at least tentatively diagnosed, one is ready to define an prescription that resolves the exception and returns the workflow to a viable state. This can be achieved, in our approach, by selecting and instantiating one of the generic exception resolution strategies that are associated with the hypothesized diagnosis. These strategies are processes like any other, are captured in a portion of the process taxonomy, and are annotated with attributes defining the preconditions (expressed using the query language) that must be satisfied for that strategy to be applicable. We have accumulated roughly 200 such strategies to date, including for example:

- IF a subprocess fails, THEN try a different process for achieving the same goal
- IF a highly serial process is operating too slowly to meet an impending deadline, THEN pipeline (i.e. releasing partial results to allow later tasks to start earlier) or parallelization to increase concurrency
- IF an agent receives garbled data, THEN trace back to the original source of the faulty data, eliminate all decisions that were corrupted by this error, and start again
- IF an agent may be late in producing a time-critical output, THEN see whether the consumer agent will accept a less accurate output in exchange for a quicker response
- IF multiple agents are causing wasteful overhead by frequently trading the use of a scarce shared resource, THEN change the resource sharing policy such that each agent gets to use the resource for a longer time
- IF a new high-performance resource applicable to a time-critical task becomes available, THEN reallocate the task from its current agent to the new agent
- IF an agent in a serial production line fails to perform a task, THEN re-allocate the task to an appropriately skilled agent further down the line

Since an exception can have several possible resolutions, each suitable for different situations, we use a procedure identical to that used in diagnosis to find the right one. Imagine, for example, that we want a
resolution for the diagnosis “agent unavailable”. We start at the root of the process resolution taxonomy branch associated with that diagnosis (Figure 4):

![Diagram of the Resolution Process Taxonomy]

Figure 4. A Fragment of the Resolution Process Taxonomy.

Three specific strategies are available, with the following preconditions and actions:

**wait till agent available**: IF the original agent will be available in time to complete task on current schedule THEN wait for original agent

**find new agent with same skills**: IF another agent with the same skills is available, THEN assign task to that agent

**change task to meet available skills**: IF task can be performed a different way using currently available agents THEN do so

The system user can prune suggested strategies based on which preconditions are satisfied, and enact or customize a strategy selected from the remainder, e.g. using the Process Handbook techniques mentioned above (Herman, Klein et al. 1998). Note that the substantial input may be needed from the user in some cases in order to instantiate a strategy into specific actions.

We have identified a small core set of meta-level operations that have proven adequate to expressing all exception resolution strategies we have encountered. These operations constitute what we call the “action language” and include such primitives as “try different plan for goal”, “insert/delete process steps”, “insert/delete resource assignment” and so on. As with the query language, a compact, well-defined and fully expressive action language raises the possibility of more fully automating the exception resolution process and reduces the “cost of admission” for including computer-based agents in robust workflow systems.

**Summary**

Our exception handling approach can be summarized as follows (Figure 5):
A "preferred" workflow model is checked at design time, using a generic process taxonomy augmented with failure mode information, to see the ways it can fail. It is then augmented with "sentinels" that check for manifestations of these exceptions. When the process is enacted, these sentinels flag any exception manifestations ("symptoms") that they encounter. The notified workflow participant can then use the Handbook’s knowledge base of exception types and associated resolution strategies to uncover the underlying cause for the problem, and select a strategy for responding appropriately. The user then enacts the strategy, making changes that allow the workflow process to continue.

The approach described herein therefore avoids, as we can see, the key problems that have traditionally faced workflow designers and users:

- it helps workflow designers determine what kinds of exceptions can occur in a given workflow process (as well as how they can be detected and resolved),
- it allows us to give workflow participants individual discretion in handling exceptions, guiding them to possible explanations and resolutions as needed
- it does so without complicating and obscuring the "preferred" process models with conditional branches devoted to exception handling

CONTRIBUTIONS OF THIS WORK

The ideas described in this paper constitute, we believe, a substantive and novel contribution to previous efforts on exception handling, which have been pursued in the context of workflow (Kunin 1982; Kreifelts and Woetzel 1987; Auranaki and Leppanen 1989; Karbe and Ramsberger 1990; Strong 1992; Mi and Scacchi 1993) manufacturing control (Parthasarathy 1989; Katz 1993; Visser 1995), model-based fault diagnosis (deKleer and Williams 1986; Krishnamurthi and Jr. 1989; Birnbaum, Collins et al. 1990; Friedrich, Gottlob et al. 1990), planning (Broverman and Croft 1987; Birnbaum, Collins et al. 1990) (Sussman 1973) (Goldstein 1975), and failure mode analysis research (Raheja 1990). The workflow and manufacturing work, in general, has not evolved to the point of constituting a computational model, has been applied to few domains (mainly software engineering and flexible manufacturing cell control) and/or has addressed a small handful of exception types. The planning work, by contrast, has developed a range of computational models but they are only applicable if the planning technology was used to develop the original work process. This is typically not the case for workflow settings where processes are defined by people rather than planning tools. Model based fault diagnosis approaches use a single
generic algorithm to uncover the causes of faults in a system without the need for a knowledge base of failure modes and resolution heuristics. This approach is predicated, however, on the availability of a complete and correct model of the system’s behavior, which is possible for some domains (e.g. the analysis of electrical circuits) but not for many others including, I would argue, most collaborative work settings that include human beings and/or complex computer systems as participants. Model-based fault diagnosis also typically assumes that resolution, once a fault has been diagnosed, is trivial (e.g. just replace the faulty component) and thus does not provide context-specific suggestions for how to resolve the problem. Current work on failure mode analysis describes a systematic process, but the actual work must be done by people based on their experience and intuitions. This is potentially quite expensive, to the extent that this analysis is rarely done, and can miss important failure modes due to limitations in the experience of the analyst(s) (Raheja 1990).

Our approach improves on these previous efforts in several important ways:

- **It constitutes a comprehensive computational approach** based on general and well-tested principles from AI and coordination science. It is applicable to processes defined by either humans or planning systems, and to processes where complete behavioral models are not available. It in fact subsumes previous work in this area by providing a framework for selecting and using whatever exception resolution algorithms appeared best suited to the particular problem at hand.
- **It is based on a substantial and growing knowledge base**. We have captured, to date, over 4000 generic process templates, 100 exception types, and 200 exception resolution strategies. Previous work has identified only a small handful of such diagnoses and strategies, raising questions about whether enough generic exception handling expertise is available to be useful.
- **It introduces a novel technique for identifying failure modes**, based on an annotated knowledge base of generic process templates, that can substantially reduce analysis costs and risks relative to current approaches.
- **While it has the potential for substantial automation**, it can also be used to enhance, rather than replace, human creativity in responding to exceptions, by suggesting potential explanations and resolutions and allowing users to customize resolutions.

**FUTURE WORK**

The collection of exception handling expertise we are acquiring has not been applied, in its current form within the Process Handbook, to support substantive real-world exception handling (see section 2 above for a discussion of our experience with previous versions of this knowledge base). One key next step, therefore, is to evaluate and extend the Handbook-based incarnation of these ideas in the context of a real or simulated workflows in multiple domains.

We also intend to explore the use of more sophisticated diagnosis algorithms, e.g. that take account of previous failed resolutions in determining a diagnosis, and are capable of handling multiple simultaneous exceptions.

Another direction will be to use the knowledge base of exception resolution strategies to allow workflow designers to more effectively define exception resolution behavior up-front, thereby exploring the tradeoff between the flexibility of run-time resolution and the predictability of design-time prescription of resolution behavior.

ACKNOWLEDGMENTS

I’d like to thank Chris Dellarocas, Abraham Bernstein and the other members of the MIT Adaptive Systems and Evolutionary Software research group, as well as the members of MIT Center for Coordination Science, for their invaluable contributions to the ideas underlying this paper.

REFERENCES


