Exception Handling in Agent Systems[1]

Mark Klein
MIT Center for Coordination Science
Email: m_klein@mit.edu

Chrysanthos Dellarocas
MIT Sloan School of Management
Email:dell@mit.edu

Abstract

A critical challenge to creating effective agent-based systems is allowing them to operate effectively when the operating environment is complex, dynamic, and error-prone. In this paper we will review the limitations of current “agent-local” approaches to exception handling in agent systems, and propose an alternative approach based on a shared exception handling service that is “plugged”, with little or no customization, into existing agent systems. This service can be viewed as a kind of “coordination doctorî; it knows about the different ways multi-agent systems can get “sick”, actively looks system-wide for symptoms of such ìillnessesî, and prescribes specific interventions instantiated for this particular context from a body of general treatment procedures. Agents need only implement their normative behavior plus a minimal set of interfaces. We claim that this approach offers simplified agent development as well as more effective and easier to modify exception handling behavior.

The Challenge: Exception-Capable Agent Systems

A critical challenge to creating effective agent-based systems is allowing them to operate effectively when, as is typical for many domains ranging from manufacturing to office work to military information gathering, the operating environment is complex, dynamic, and error-prone (Suchman 1983; Auramaki and Leppanen 1989; Karbe and Ramsberger 1990; Strong 1992; Mi and Scacchi 1993). In such domains, we can expect to utilize a highly diverse set of agents; some have fairly sophisticated coordination capabilities, but many will be simple encapsulations of legacy applications. New tasks, agents and other resources can be expected to appear and disappear in unpredictable ways. Communication channels can fail or be compromised, agents can ìdieî (break down) or make mistakes, inadequate responses to the appearance of new tasks or resources can lead to missed opportunities or inappropriate resource allocations, unanticipated agent inter-dependencies can lead to systemic problems like multi-agent conflicts, ìcircular waitî deadlocks, and so on. All of these departures from “ideal” collaborative behavior can be called *exceptions*. The result of inadequate exception handling is the potential for systemic problems

[1] This paper appears in the Proceedings of the Third Intrernational Conference on Autonomous Agents, Seattle, WA, May 1-5, 1999. The authors gratefully acknowledge the support of the DARPA CoABS Program (contract F30602-98-2-0099) while preparing this paper.

such as clogged networks, wasted resources, poor performance, system shutdowns, and security vulnerabilities.

In this paper we will review the limitations of current “agent-local” approaches to exception handling in agent systems, and propose an alternative “shared service” approach that offers simplified agent development as well as more effective and easier to modify exception handling behavior. Initial versions of this service have been developed and tested in the multi-agent collaborative design conflict management domain; we will describe our preliminary results as well as our future plans.

Contributions and Limitations of Current Work

Current approaches to agent exception handling have serious limitations in terms of agent development cost and the effectiveness of system-wide exception handling behavior. The standard approach has been to ìcompile inî complicated and carefully coordinated exception handling behaviors into all problem-solving agents. This is, however, fundamentally problematic, since the causes, manifestations and resolutions for agent system exceptions are inherently systemic and context-sensitive rather than localizable to any particular agent. A circular wait deadlock, for example (where several agents are all stalled waiting for inputs from each other) can only be detected as a *pattern* of agent interactions, and can only be resolved by changing that pattern (e.g. by replacing one agent with another that has different input requirements). Agent developers must thus anticipate all the contexts in which the agent may be used, but this is extremely difficult. No systematic methodology is available, however, to help developers identify all relevant exception types and resolution strategies. Making changes in exception handling behavior is difficult because it potentially requires coordinated changes in multiple agents. Agents become much harder to maintain, understand and reuse because the relatively simple normative behavior of an agent becomes obscured by a potentially large body of code devoted to handling exceptional conditions. Finally, it is unrealistic to expect that all agents will have sophisticated exception handling capabilities built in. In many cases we will have to be able to operate with agents whose design incorporates only the most basic capabilities.

A few efforts have done some preliminary exploration of the use of distinct exception handling services. This work has occurred predominantly in the context of business process enactment (Kreifelts and Woetzel 1987; Auramaki and Leppanen 1989; Karbe and Ramsberger 1990; Strong 1992; Mi and Scacchi 1993), manufacturing control (Parthasarathy 1989; Katz 1993; Visser 1995) and planning (Broverman and Croft 1987; Birnbaum, Collins et al. 1990). The process enactment and manufacturing work, in general, has either not evolved to the point of constituting a computational model, or has been applied to a very limited range of domains (e.g. just software engineering or flexible manufacturing cell control) and exception types (e.g. just inappropriate task assignments). The planning work, by contrast, has developed a range of computational models but their ability to redesign a multi-agent work process in response to an exception is contingent upon the planning approach having been used to develop the original work process. This requirement is difficult or impossible to satisfy in an environment where the work process emerges dynamically via the interaction of multiple heterogeneous agents.

Our Approach: A Shared Exception Handling Service

Our approach transcends the limitations of current approaches by creating a shared exception handling service that can be “plugged”, with little or no customization, into existing agent systems to add the ability to function in exception-prone environments. This service can be viewed as a kind of “coordination doctorî; it knows about the different ways multi-agent systems can get “sick”, actively looks system-wide for symptoms of such ìillnessesî, and prescribes specific interventions instantiated for this particular instance from a body of general treatment procedures. Agents need only implement their normative behavior plus a minimal set of interfaces that assume only that each agent can report on its own behavior and modify its own actions to at least some extent. This vision is realized by building on four key innovations:

- We define a clear division of labor. Problem solving agents focus on executing their own ìnormalî problem solving behavior, while the exception handling agents focus on detecting and resolving exceptions in the agent ensemble as a whole.
- The exception handling service applies a knowledge base of generic exception handling detection, diagnosis and resolution expertise which can be applied to a wide range of domains.
- The ìcost of admissionî is only that agents understand a standard language providing at least a basic level of self-awareness and self-modifiability, comparable to what is required of agents capable of reasonably sophisticated coordination in exception-free contexts.
- This service can be implemented as a set of standard agents that can be “plugged” in to any agent system whose agents support the language interfaces described above.

We describe our approach in more details in the paragraphs below.

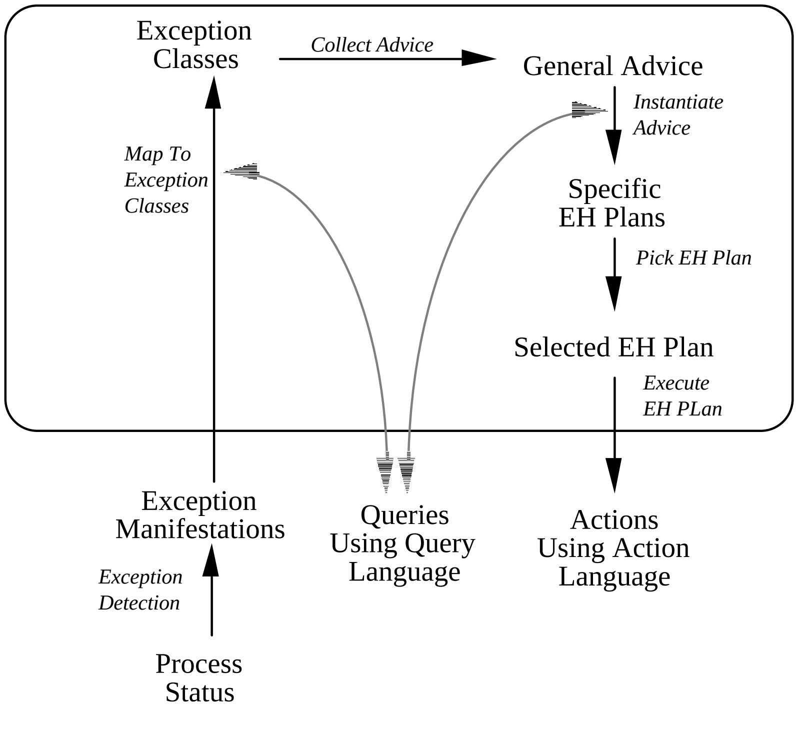
Generic Exception Handling Expertise: The key element underlying our approach is the simple but powerful notion that generic and reusable exception handling expertise can be usefully separated from the knowledge used by agents to do their ìnormalî work. There is substantial evidence for the validity of this claim. Early work on expert systems development revealed that it is useful to separate domain-specific problem solving and generic control knowledge (Barnett 1984; Gruber 1989). Analogous insights were also confirmed in the domains of collaborative design conflict management (Klein 1991) and in preliminary work on process exception management (Klein 1997). Generic exception management strategies are easy to find (Klein 1989). Some examples include:

- if an agent plan has failed, backtrack to a different plan for achieving the same goal
- if a highly serial process is operating too slowly to meet an impending deadline, and the subtasks have only serial dependencies, use pipelining (i.e. releasing results for earlier subtasks before later subtasks are completed) to increase concurrency
- if an agent receives garbled data, trace the problem 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, 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, 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, consider reallocating the task from its current agent to the new agent
- if an agent in a serial production line fails to perform a task, try to re-allocate the task to an appropriately skilled agent further down the line

It is our experience that such strategies are easy to acquire from a wide range of research literature sources, as well as by generalizing from the vast range of exception handling cases we all encounter. We have identified about 300 strategies to date; more details will be given below.

Heuristic Classification: A useful metaphor for organizing such expertise, we have found, is to treat exception handling (EH) as a heuristic classification process (Clancey 1984) analogous to that used in medical diagnosis. In this approach, an exception manifestation (i.e. symptom), once detected, is mapped to candidate diagnoses in a pre-defined taxonomy of generic underlying causes; generic strategies associated with these diagnoses are then instantiated into candidate exception resolution plans, one of which is then selected and executed:



The approach thus instantiates generic exception handling expertise into specific situations. The EH service communicates with agents using pre-defined languages for learning about the exception(s) (the query language) and for describing exception resolution actions (the action language). Agents can take any form as long as they are capable of responding appropriately to at least a minimum subset of these query and action languages.

Exception Detection: The first step in detecting exceptions is, of course, having a model of what the ìcorrectî behavior for the multi-agent system is. When an agent is introduced into a multi-agent system, therefore, it must register at least a rudimentary model of its normative behavior with the exception handling service. This model is mapped to a list of the failure modes that are known to occur for that kind of normative behavior, and sentinels are generated to detect those modes.

Failure mode identification is done making use of a taxonomy of generic problem solving processes wherein each process is annotated with the different ways it can fail. When a new agent is registered, we merely identify the generic processes corresponding to that agentís behavior, and derive the applicable failure modes from that. For example, it is typical for agents to require the output of another agent. The processes for managing such ìflowî 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 (ìwrong timeî), of the wrong type (ìwrong thingî) and so on. Similar analyses can be done for other generic processes, e.g. resource sharing, diagnosis, synthesis, market-based coordination and so on. We are building for this purpose upon the process taxonomy being developed in the context of the MIT Process Handbook. The Handbook is a substantive (3700+ entity) and growing repository of coordination mechanisms and other problem solving processes (Malone, Crowston et al. 1993; Dellarocas, Lee et al. 1994; Malone and Crowston 1994) which has been under development for roughly the past five years in the MIT Center for Coordination Science.

Our work to date in performing failure mode analysis has revealed a wide range of exception types (Klein 1997). Exceptions in general involve violations of some either implicit or explicit assumption underlying a collaborative work process (e.g. stability of resources, correctness of output etc.) and can include change in resources, organizational structure, agent system policies, task requirements or task priority. They can also include incorrectly performed tasks, missed due dates, resource contentions between two or more distinct processes, unexpected opportunities to merge or eliminate tasks, conflicts between actions taken in different process steps and so on.

Every failure mode can have associated with it a script that searches for the pattern of agent behavior corresponding to that failure. These scripts, once instantiated, play the role of ìsentinelsî that alert the exception handling service when the condition they were created to detect has occurred. A typical sentinel, for example, may check for a task becoming late, violation of resource limits, circular wait patterns, and so on. To define these scripts, we build upon pattern matching tools developed in previous work (Klein 1997).

Exception Diagnosis: The diagnosis mechanisms works by traversing a taxonomy of possible exception diagnoses based on the presenting symptoms as well as information about the process model being enacted. This is a "shallow model" approach (Chandrasekaran and Mittal 1983) 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 first-principle-based behavioral models do not exist. 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.

The diagnosis hierarchy, in our current model, is structured as a decision tree wherein the system starts at the top most abstract diagnosis and attempts to refine it to more specific diagnoses by traversing down the tree and selecting the appropriate decision branches by asking questions, expressed as query language statements, of the relevant problem solving agents. For example, if the system is assessing whether the diagnosis of “circular wait deadlock” applies, it may ask agents for which other agents they are waiting for inputs from. This traversal can result in more than one candidate diagnosis, since multiple causes may be suggested by the same symptoms.

Exception Resolution: Once one or more candidate diagnoses for an exception have been identified, the next step is to generate, using a knowledge base of generic exception resolution strategies, specific plans for resolving the problem. A diagnosis class will often have several potential resolution strategies. Since they may not all be applicable for a particular exception, a decision tree procedure identical to that used to select diagnoses is used to find the generic strategies for a given diagnosis. Strategies are represented as executable script templates whose actions are described using the action language. Every template has "slotsî which are filled with context-specific values, found using query language queries, to produce specific exception resolution plans. The resolution strategy “backtrack to untried plan for goal”, for example, would include slots for the goal and plan that are filled in by asking the affected agent what goal is was trying to achieve, and what other plans are available for achieving that same goal. Typically, many possible candidate plans can be generated for a given exception. We can backtrack, for example, in as many ways as there are alternative plans. In our previous work we have found that a relatively small collection of domain-independent heuristics (e.g. ìpick the plan that makes the smallest changeî) has been effective in producing a useful ranking of candidate exception resolution plans.

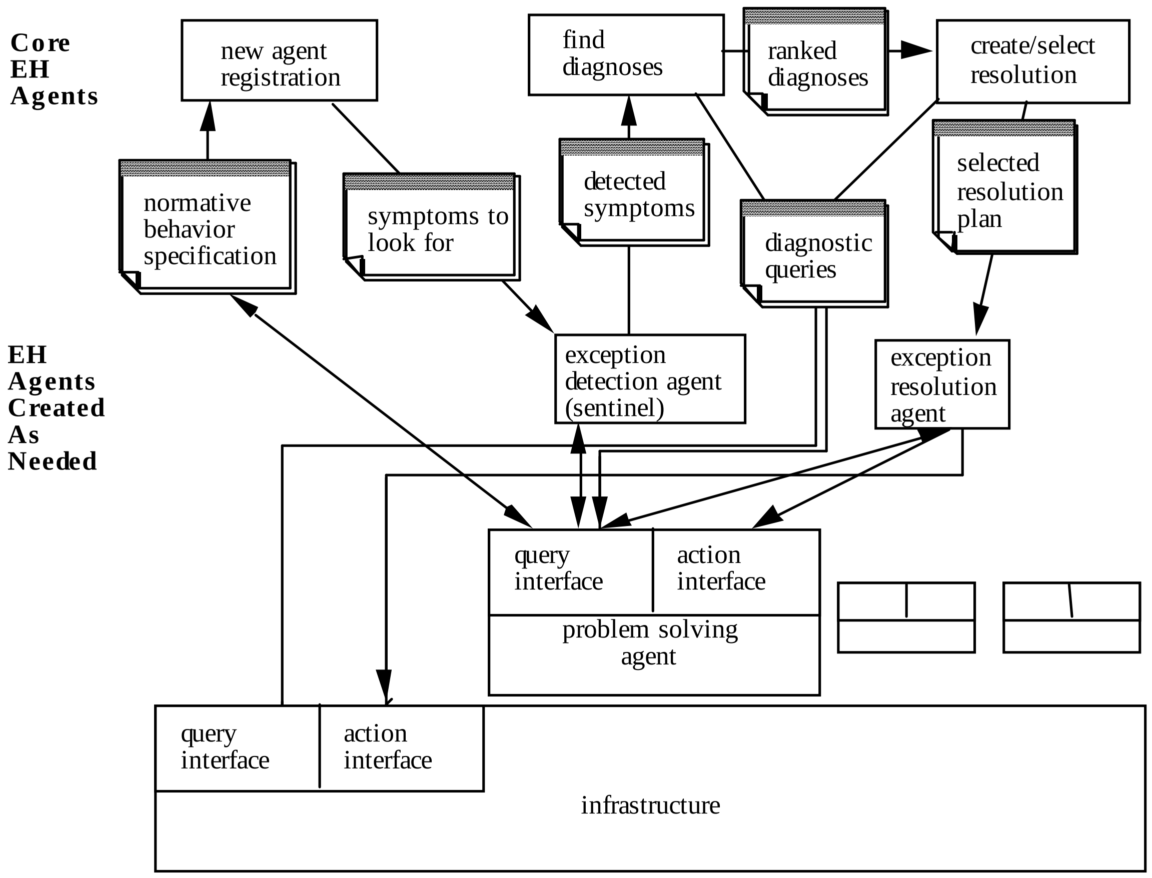
The Query and Action Languages: As we have seen, the query and action languages represent the medium by which the exception handling service interacts with the problem solving agents to detect, diagnose and resolve exceptions. The query language is used to get agent state information, and the action language is used to modify it.

The query language we use builds upon that developed in earlier systems (Klein 1989; Klein 1993) extending it to include queries concerning normative agent behavior models. The query language is relatively large, and we will make the effort to consolidate it into a smaller set of critical query types. The action language, in contrast, consists we have found of a relatively small set of operators (Klein 1989). These include changing the process model (re-ordering, deleting or adding new tasks; changing the resources allocated to a task; canceling tasks) and changing the work package contents.

Our experience to date has shown that agents do not have to understand all of the query or action language primitives in order to benefit from the exception handling service, but the more primitives they can understand, the more effective the exception handling service is likely to be. This is because the more generic exception diagnoses and resolution strategies tend to require only the simplest and easiest to implement queries and actions, but the more sophisticated (and presumably more effective) diagnoses and resolutions use the more ìadvancedî primitives.

The query and action languages can be viewed as representing a ìprice of admissionî to our approach. These languages only require, however, that the individual problem solving agents be able to describe their own behavior as well as a modify their own actions; the exception handling service is responsible for understanding how local knowledge and actions can be coordinated to produce a globally effective exception response. Previous DAI research suggests, moreover, that for many cases we want our agents to have roughly that level of self-awareness and self-modifiability in order to support effective coordination even in the absence of exceptions (Findler and Lo 1988).

System Architecture: The capabilities described above can be implemented straightforwardly as agents that can simply be plugged-in to an existing agent system with suitable interfaces:



This architecture consists of exception handling agents, problem solving agents as well as the agent systems infrastructure, all of which must support at least the base level query and action languages. When a new agent is created, the ìnew agent registrationî agent takes a description of its normative behavior and creates sentinels (exception detection agents) as necessary to look for evidence of dysfunctional behavior. Should a sentinel detect such symptoms, this information is sent to a ìdiagnosisî agent which produces a ranked set of candidate diagnoses. These in turn are sent to the resolution agent which defines a resolution plan instantiated in the form of a ìresolutionî (exception resolution) agent. We can have redundant copies of these agents, thereby increasing performance and addressing potential failures in the exception handling agents themselves.

Human in the Loop: While the architecture above has been presented as a fully automated one, in at least some cases it will make sense to include a human ìexecutive managerî in the loop. Our previous work in this area, for example, used human input to modify the ranking of diagnoses and resolution plans proposed by the exception handling service, and thereby direct the system in the direction the human users considered more appropriate (Klein and Lu 1991). We have found that the exception handling service can help human users better understand and more creatively resolve exceptions, even if they did not use the particular resolutions proposed by the system.

Evaluation: Contribution to Improving Agent-Based Systems

The ideas described in this paper have already been substantially validated through nearly a decade of development and evaluation of successful systems for resolving multi-agent exceptions in the collaborative design (Klein 1989; Klein 1991; Klein 1993) and collaborative requirements capture (Klein 1997) domains. This led to the development of the basic heuristic classification approach, software tools for exception diagnosis and resolution, a substantive standardized language for communication between agents and the exception handling service, a highly expressive rationale capture language (Klein 1993), as well as a substantive and growing knowledge base of exception resolution heuristics. More recent work has begun applying these ideas to a broader range of exception types (Klein 1995; Klein 1996; Klein 1997). The current contents of the exception handling knowledge base can be characterized as follows:

Aspect	*Number*	*Examples*
conflict detection strategies	~10	• check for violated resource budget • check for inconsistent parameter constraints
query operators in standardized agent communication language	~100	• what is the rationale for the decision? • is the parameter constraint relaxable?
action operators in standardized agent communication language	~10	• relax constraint • try different plan for goal
exception diagnoses	~100	• agent constraints too ambitious • excessive serialization in work process
exception resolution strategies	~300	• relax constraints, maximizing summed utilities • pipeline tasks with serial dependencies
exception plan selection heuristics	~10	• pick most specific resolution plan • abandon low level goals before high level goals

Our results to date suggest that the exception handling service approach enables two classes of important benefits:

- easier agent development: This approach makes it much easier to develop, understand, maintain and reuse problem solving agents, since developers can focus on their normative behavior without having to build in responses to all possible exceptions. This greatly reduces the cost of achieving the transparent agent interoperability that underlies the appeal of agent systems. Another advantage is that this approach does not rely on the existence of powerful exception-handling support facilities in every agentís implementation language.
- easier to specify effective exception handling behavior: We are less likely to miss important failure modes, and will probably use better exception resolution practices, by taking

advantage of a systematically accumulated knowledge base of exception handling ìbest practicesî. It will also be much easier, we believe, to specify and modify systemic exception-handling expertise if it is treated as a functional unit rather than captured as a series of carefully designed interlocking behaviors spread over myriads of diverse agents.

These benefits translate into more reliable, predictable and efficient agent-based system operation.

Future Work

We plan to follow two inter-related tracks in our future work: (1) further development of the exception handling knowledge base and underlying diagnostic technology, and (2) further evaluation of this technology in both simulated and "real-world" testbed contexts.

Technical issues we currently consider important include extending the diagnostic approach to be able to handle multiple simultaneous exceptions in a coordinated way (Wu 1990), as well as reducing as much as possible the size of the query/action languages that agents need to understand in order to interact effectively with the exception handling service. We also plan to explore ìmodel-basedî diagnostic approaches (Genesereth 1982; Kleer, Macworth et al. 1990) which have been applied with good results to explaining faults in that subclass of systems where complete behavioral models exist

Our evaluation plan consists of a graded series of experiments, occurring first in a simulated agents testbed (where we have the maximum flexibility in designing the test scenarios), and then transitioning to an externally developed testbed (to assess and demonstrate the ability to extend a pre-extending agent system with our exception handling technology). The simulated testbed will evaluate agent system behavior using such heuristics as problem solving time, effectiveness of resource utilization, the understandability of the agent ensemble behavior to human observers, ability of problem solving agents to work in multiple ensemble contexts w/o modification, and the ability to control the tradeoff between exception handling and problem solving effort. We currently plan to do our first tests in the manufacturing logistics domain. The second testbed will enable a "technology integration experiment" wherein we explore integration of our exception handling technology into a agent system not originally designed for that purpose. This will allow us to assess and demonstrate the ability of our technology to be "plugged in" to other testbeds, help us identify the knowledge base and query/action language enhancements needed, if any, and provide insights into how integration can best be done. We can therefore view this as a ìfinal rehearsalî for adoption of our technology by agent system developers outside of our project team. We are currently considering, for this purpose, the MIT AI Lab's "Intelligent Room", a large real-time agent-based information gathering/presentation system (Kautz, Selman et al. 1994; Coen 1997; Coen 1997).

References

Auramaki, E. and M. Leppanen (1989). Exceptions and office information systems. Office Information Systems: The Design Process. Proceedings of the IFIP WG 8.4 Working Conference, Linz, Austria.

Barnett, J. A. (1984). "How Much Is Control Knowledge Worth? A Primitive Example." Artificial Intelligence **22**(1): 77-89.

Birnbaum, L., G. Collins, et al. (1990). Model-Based Diagnosis of Planning Failures. AAAI.

Broverman, C. A. and W. B. Croft (1987). "Reasoning About Exceptions During Plan Execution Monitoring." Aaai-87 **1**: 190-195.

Chandrasekaran, B. and S. Mittal (1983). "Deep Versus Compiled Knowledge Approaches To Diagnostic Problem Solving." Int. J. Man-Machine Studies: 425-436.

Clancey, W. J. (1984). "Classification Problem Solving." Aaai: 49-55.

Coen, M. (1997). Building Brains for Rooms: Designing Distributed Software Agents. Proceedings of IAAI-97.

Coen, M. (1997). Towards Interactive Environments: The Intelligent Room. Proceedings of HCI-97.

Dellarocas, C., J. Lee, et al. (1994). Using a Process Handbook to Design Organizational Processes. *Proceedings of the AAAI 1994 Spring Symposium on Computational Organization Design*, Stanford, California.

Findler, N. V. and R. Lo (1988). An Examination of Distributed Planning in the World of Air Traffic Control. Readings in Distributed Artificial Intelligence. A. H. Bond and L. Gasser. California, Morgan Kaufmann**:** 617--627.

Genesereth, M. R. (1982). Diagnosis Using Hierarchical Design Models.

Gruber, T. R. (1989). "A Method For Acquiring Strategic Knowledge." Knowledge Acquisition **1**(3): 255-277.

Karbe, B. H. and N. G. Ramsberger (1990). Influence of Exception Handling on the Support of Cooperative Office Work. Multi-User Interfaces and Applications. S. Gibbs and A. A. Verrijin-Stuart, Elsevier Science Publishers**:** 355-370.

Katz, D. M., S. (1993). Exception management on a shop floor using online simulation. Proceedings of 1993 Winter Simulation Conference - (WSC '93), Los Angeles, CA, USA, IEEE; New York, NY, USA.

Kautz, H., B. Selman, et al. (1994). An Experiment in the Design of Software Agents. Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), Seattle, WA.

Kleer, J. d., A. K. Macworth, et al. (1990). Characterizing Diagnoses.

Klein, M. (1989). Conflict Resolution in Cooperative Design. Computer Science. Urbana-Champaign, IL., University of Illinois.

Klein, M. (1991). “Supporting Conflict Resolution in Cooperative Design Systems.” IEEE Systems Man and Cybernetics **21**(6).

Klein, M. (1993). “Capturing Design Rationale in Concurrent Engineering Teams.” IEEE Computer **26**(1): 39-47.

Klein, M. (1993). “Supporting Conflict Management in Cooperative Design Teams.” Journal on Group Decision and Negotiation **2**: 259-278.

Klein, M. (1995). “Conflict Management as Part of an Integrated Exception Handling Approach.” AI in Engineering Design Analysis and Manufacturing (AI EDAM) **9**: 259-267.

Klein, M. (1996). “Core Services for Coordination in Concurrent Engineering.” Computers in Industry.

Klein, M. (1997). “An Exception Handling Approach to Enhancing Consistency, Completeness and Correctness in Collaborative Requirements Capture.” *Concurrent Engineering Research and Applications*(March).

Klein, M. (1997). Exception Handling in Process Enactment Systems. Cambridge MA, MIT Center for Coordination Science.

Klein, M. and S. C.-Y. Lu (1991). Insights Into Cooperative Group Design: Experience With the LAN Designer System. Sixth International Conference on Applications of Artificial Intelligence in Engineering (AIENG '91), Uk.

Kreifelts, T. and G. Woetzel (1987). Distribution and Exception Handling in an Office Procedure System. Office Systems Methods and Tools, IFIP WF 8.4 Working Conference on Methods and Tools for Office Systems.

Malone, T. W., K. Crowston, et al. (1993). Tools for inventing organizations: Toward a handbook of organizational processes. 2nd IEEE Workshop on Enabling Technologies Infrastructure for Collaborative Enterprises (WET ICE), Morgantown, WV, USA.

Malone, T. W. and K. G. Crowston (1994). “The interdisciplinary study of Coordination.” *ACM Computing Surveys* **26**(1): 87-119.

Mi, P. and W. Scacchi (1993). Articulation: An Integrated Approach to the Diagnosis, Replanning and Rescheduling of Software Process Failures. 8th International Conference on Knowledge-Based Software Engineering.

Parthasarathy, S. (1989). Generalised process exceptions-a knowledge representation paradigm for expert control. Proceedings of the Fourth International Conference on the Applications of Artificial Intelligence in Engineering, Cambridge, UK, Comput. Mech. Publications; Southampton, UK.

Strong, D. M. (1992). “Decision support for exception handling and quality control in office operations.” Decision Support Systems **8**(3).

Suchman, L. A. (1983). “Office Procedures as Practical Action: Models of Work and System Design.” ACM Transactions on Office Information Systems **1**(4): 320-328.

Visser, A. (1995). “An exception-handling framework.” International Journal of Computer Integrated Manufacturing **8**(3): 197-203.

Wu, T. D. (1990). Efficient Diagnosis of Multiple Disorders Based on a Symptom Clustering Approach.