Toward Exception Handling Infrastructures for Component-Based Software

Chrysanthos Dellarocas
Adaptive Systems and Evolutionary Software Group
Center for Coordination Science
Massachusetts Institute of Technology
MIT, Room E53-315
Cambridge, MA 02139, USA
Email: dell@mit.edu

Abstract:

This paper argues that component-based software development introduces additional sources of risk because (i) independently developed components cannot be fully trusted to conform to their published specifications and (ii) very often, software failures are caused by systemic patterns of interaction that cannot be localized to any individual component. It articulates the need for a separate exception handling infrastructure to address these issues. The proposed approach creates a clean separation between the normative and exception handling functions in component-based software systems. Components focus on executing their own "normal" problem solving behavior, while an exception handling service focuses on detecting and resolving exceptions in the system as a whole. The exception handling service works by applying a knowledge base of generic and highly reusable exception handling expertise to the particular run-time contexts it faces. The "cost of admission" for this approach is only that individual components implement at least a minimum set of interfaces that require only self-awareness and self-adaptation. This technology can be realized as a standardized middleware service that can add exception handling to any component-based system with appropriate interfaces.

1. The Challenge

Much of the appeal of component-based software development derives from the potential of creating complex systems without having to implement the whole system from scratch - the desired services are provided by independently developed, off-the-shelf components. Components are selected on the basis of their "credentials", that is, published specifications of their capabilities and constraints. Such specifications are, by necessity, incomplete and imprecise descriptions of a component's behavior (Shaw, 1997). A critical challenge to achieving the vision of component-based software development is ensuring that independently developed components correctly provide the services advertised in their published specifications and obey the resource and performance constraints implied by them. Furthermore, it is important to ensure that component ensembles can operate effectively when, as is increasingly typical for many business domains, the operating environment is complex, dynamic and error-prone.

Until now, the standard approach to this problem has been to "compile in" complicated and carefully coordinated exception handling behaviors into all individual components. This is, however, a fundamentally problematic approach for component-based software, because component users do not have access to the internals of a component. Furthermore, the causes, manifestations and resolutions for most exceptions are inherently systemic and context-sensitive rather than localizable to any particular component. A circular wait deadlock, for example, where several components are all stalled waiting from inputs from each other, is caused not by any individual component but rather by the interaction of several components in a given context. Plausible but incorrect data from one component may only have its impact far downstream in the application. The resolution to a circular wait deadlock, to give another example, is to redesign the pattern of component interconnections (by replacing one of the components by another with different input requirements) rather than to change the behavior of any individual component.

The “component-localized” approach has several serious limitations. Component developers must anticipate all the contexts in which a component may be used. No systematic methodology is available to help developers identify all the possible exception types and appropriate resolution strategies. Making changes in a system’s exception handling behavior is difficult because it potentially requires coordinated changes in several constituent components. The resulting components are much harder to maintain, understand and reuse, because the “normative” behavior of the component has been obscured by a large body of code devoted to handling exceptional conditions. Finally, it is unrealistic to expect that all components will have sophisticated exception handling capabilities built in.

As a response to these challenges, this paper proposes the need for a specialized exception handling infrastructure for component-based software systems. It outlines the principles of such an infrastructure and discusses the implications of such a service for other aspects of component management infrastructure, as well as for component developers.

The Adaptive Systems and Evolutionary Software (ASES) group at MIT is currently engaged in developing an exception handling infrastructure, as described in this paper, in the context of agent-based and workflow software systems. For more information about the activities of our group, the interested reader is referred to our web site at http://ccs.mit.edu/ases

2. An Exception Handling Service for Component-based Software

The challenges outlined in the previous section can be addressed directly by establishing a “division of labor” between normal system operation and exception handling. In this approach, individual components need only implement their normative behavior plus a minimal set of interfaces through which a component can report on its current behavior and modify its operation to at least some extent. A separate exception handling service, itself potentially implemented as a set of components, uses these interfaces plus a knowledge base of generic exception management expertise to detect when things go wrong in the system and take the appropriate corrective actions. This service can be viewed as a kind of “coordination doctor” that one can plug into an existing component-based system; it contains a large knowledge base describing the different ways software systems can fail, actively looks system-wide for symptoms of such “illnesses”, and prescribes specific interventions instantiated from a body of general exception resolution strategies also stored in its knowledge base.

The key idea underlying this approach is the simple but powerful notion that generic and reusable exception handling expertise can be usefully separated from the knowledge used by problem-solving components to do their “normal” work. There is substantial evidence for the validity of this notion. Early work on expert systems development revealed that it is useful to separate regular problem solving from generic heuristics for controlling this activity (Gruber, 1989; Barnett, 1984). 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). Examples of generic exception management expertise are easy to find, and range from very general heuristics (e.g. “backtrack to a different plan for achieving a goal if a previous plan has failed”) to more specific ones (e.g. “if a highly serial process is operating too slowly to meet an impending deadline, increase concurrency by pipelined or parallel operations”).

3. Architectural Overview

In the paragraphs below we will go into more detail in how the components of this approach, i.e. exception detection, diagnosis, and resolution generation are realized.

Exception Detection: The first step in detecting exceptions is, of course, to have some model of the “correct” behavior both for the entire system, as well as for each individual component. These models will be prepared by the target system developers (for the entire system) and should be part of components’ published specification (for each individual component). During design time, the models are mapped to a list of the failure modes that are known to occur for each kind of normative behavior. As a result of this analysis, the system is instrumented with additional *sentinel components*. The purpose of sentinel components is to detect particular failure modes by looking for the appropriate patterns in the behavior of base components. Base components should provide an *introspection interface*, through which sentinels will be able to query components and find out about their current behavior.

Failure mode identification can be greatly facilitated by the existence of a taxonomy of generic component types wherein each generic type has associated with it the different ways that the services provided by components of that type can fail. For each component, we merely identify the type of the component in the taxonomy, and from that we can derive the failure modes that apply. A similar taxonomy is also required for component interconnection patterns. For example, it is typical for components to require as input the output of another component. Previous coordination science research has determined that such “flow” dependencies involve making sure the right thing gets to the right place at the right time shown (Malone, 1994). This immediately implies a set of possible failure modes including an input being late (“wrong time”), of the wrong type (“wrong thing”) and so on. Similar analyses can be done for other kinds of transfer processes (e.g. one-to-many “sharing” dependencies) as well as for generic problem solving processes such as diagnosis, synthesis, market-based coordination and so on.

Exception Diagnosis: During run-time, sentinel components monitor system operation and generate appropriate events when exception manifestations are detected. A key challenge here is the fact that the symptoms revealed by sentinels can suggest a variety of possible underlying causes. A diagnostic engine is triggered to determine the underlying cause of the detected symptoms.

A heuristic classification approach (Clancey, 1984) is well-suited to exception diagnosis. In this approach, potential diagnoses (i.e. underlying exception causes) are arranged into a taxonomy ranging from the very abstract at the top to the very specific at the bottom. The diagnosis mechanism works in a top-down way by iteratively increasing the specificity of a diagnosis based on the symptoms as well as information about the process model being enacted. This is essentially a "shallow model" approach (Chandrasekaran, 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.

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 diagnosed problem. A diagnosis class will often have several potential resolution strategies available. 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. Once a resolution strategy has been selected, it is enacted. Enactment of a resolution strategy might involve undoing/redoing previously completed activities or modifying the structure or behavior of the system. Components should provide a *adaptation interface* through which the exception handling service can inquire about a specific component’s adaptation capabilities and can instruct the component to modify itself (e.g. to undo or redo an operation, to change its resource requirements, etc.) during exception resolution.

User Interface: In highly complex systems, it is unrealistic to expect that automated processes can

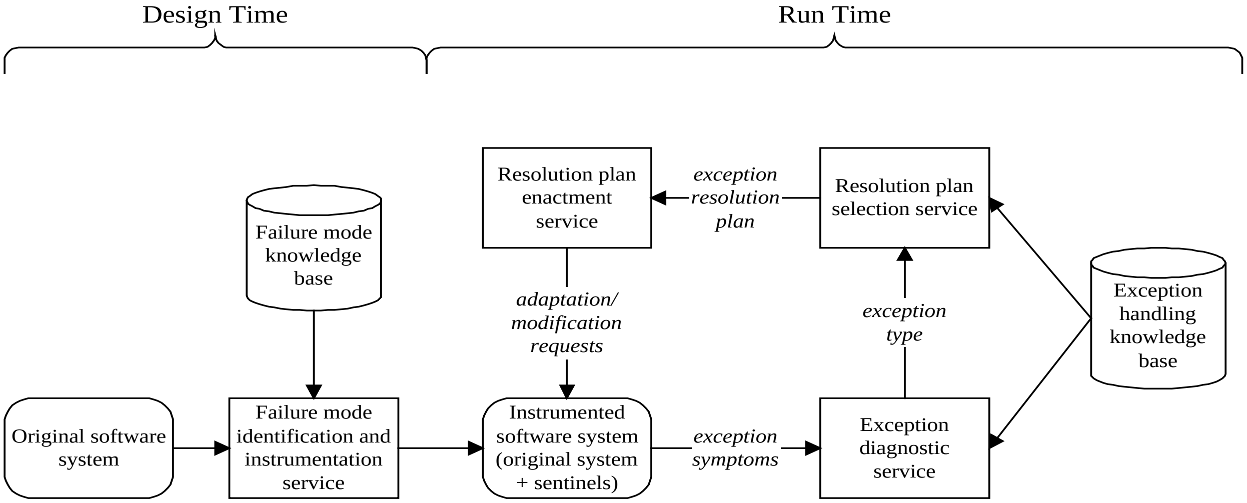
Design Time
Run Time
Resolution plan enactment service
exception resolution plan
Resolution plan selection service
Failure mode knowledge base
adaptation/ modification requests
exception type
Exception handling knowledge base
Original software system
Failure mode identification and instrumentation service
Instrumented software system (original system + sentinels)
exception symptoms
Exception diagnostic service

Figure 1: Summary of proposed exception management approach.

completely detect, diagnose and resolve all possible exceptions. User input might be required in order to finalize the diagnosis of an exceptional condition or the selection of a resolution plan. A successful exception handling infrastructure can help human users better understand and more creatively resolve exceptions, even if they do not use the particular resolutions proposed by the system.

The overall approach is summarized in Figure 1.

4. Implications for Infrastructure and Component Developers

The idea of a separate exception handling infrastructure for component-based software systems has a number of implications, both for other aspects of component management infrastructure, as well as for individual component developers.

Implications for component management infrastructure
The exception handling service described in this paper relies on descriptions of the normal behavior of components, as well as on descriptions of how this behavior might fail. The need to provide this information for each individual component can be greatly facilitated by the existence of standardized taxonomies of common component and connector classes annotated with failure mode information.

Such taxonomies are analogous to taxonomies of professions and skills used in the job market. There are many good reasons for developing such taxonomies other than failure mode analysis. For example, the existence of component class taxonomies would assist component developers to focus their energies on developing "useful" types of components, it would help application developers locate and compare the right components for their applications, etc. A number of academic and industrial projects are focused on developing taxonomies of components (Prieto-Diaz, 1987; Barn 1997) and connectors (Shaw, 1996; Dellarocas, 1997). The novel proposal here is that such taxonomies should be augmented with failure mode information.

Another prerequisite for the successful implementation of an exception handling service is the existence of comprehensive knowledge bases of exception handling expertise. Such knowledge bases should contain information on how to detect, diagnose and resolve exceptional conditions. Currently, such knowledge bases are still an object of ongoing research (Klein, 1997). The Adaptive Systems and Evolutionary Software (ASES) research group at MIT is in the process of developing such a knowledge base.

Implications for component developers
In order for individual components to be able to participate in the exception handling system described in this paper, they must satisfy two requirements:

1. Provide a set of "credentials", that is, a specification of their normal behavior, performance and resource requirements. These credentials are necessary, both for selecting components and for comparison with a component's actual behavior in order to detect exceptional conditions. The need for such specifications is becoming widely accepted in the component-based software engineering community. Although several projects are underway, no standards have emerged yet. One significant effort in this direction is the joint work undertaken by Sterling Software Inc. and Microsoft to define information models based on the Unified Modeling Language (UML, 1997) for storage of components in the Microsoft Repository (Microsoft, 1997).

2. Provide two additional interfaces for communication with the exception handling infrastructure: An *introspection interface*, which allows the exception handling engine to monitor the component's current behavior and progress, and a *adaptation interface*, which allows the engine to ask a component to reconfigure/adapt its behavior as a consequence of an exception resolution strategy. The idea of these two interfaces as a standardized requirement for all software components is novel. Previous research in Distributed Artificial Intelligence suggests that in many cases software agents must have some level of self-awareness and self-adaptation in order to support effective coordination even in the absence of exceptions (Findler, 1988). The intention of our proposal is to define several different levels of sophistication for these interfaces. Component developers would then choose to provide the interfaces at the desirable level of sophistication. More sophisticated introspection and adaptation interfaces would allow better detection, diagnosis and resolution of exceptions but would increase the complexity (and cost) of the component. This way, a component's capability to collaborate with an exception handling infrastructure will become a

differentiating factor in the marketplace of software components.

5. Conclusions

This paper argues that component-based software development introduces additional sources of risk because (i) independently developed components cannot be fully trusted to conform to their published specifications and (ii) very often, software failures are caused by systemic patterns of interaction that cannot be localized to any individual component. The paper articulates the need for a separate exception handling infrastructure to address these issues. The proposed approach is based on the following key features:

- It creates a clean separation between the normative and exception handling functions in component-based software systems. Components focus on executing their own "normal" problem solving behavior, while an exception handling service focuses on detecting and resolving exceptions in the system as a whole.
- It makes use of a taxonomy of domain-specific component and connector types, augmented with failure mode information, in order to instrument a set of base components with additional, exception detecting, sentinel components.
- The exception handling service works by applying a knowledge base of generic and highly reusable exception handling expertise to the particular run-time contexts it faces.
- The "cost of admission" for this approach is only that components implement at least a minimum set of interfaces that require only self-awareness and self-adaptation.
- This technology can be realized as a standardized infrastructural (middleware) service that can add exception handling to any component-based system with appropriate interfaces.

As noted above, these innovations enable easier component development, better exception handling and easier specification of exception handling behavior. These benefits translate in turn into more reliable, predictable and efficient component-based software systems.

Acknowledgments

I'd like to thank Mark Klein for his invaluable contributions to the ideas underlying the paper.

References

Barn, B. (1997) "A classification model for component-based development" TI Technical Journal, April 1997

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

Chandrasekaran, B. and Mittal S. (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.

Dellarocas, C. (1997). "Towards A Design Handbook for Integrating Software Components". Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST'97), Pittsburgh, PA: 3-13.

Findler, N. V. and Lo R. (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.

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

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

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

Microsoft (1997). “Component Description Information Model” July 1997. See http://www.microsoft.com/repository

Prieto-Diaz R. and Freeman P. (1987) “Classifying Software for Reusability.” IEEE Software 4(1): 6-16

Shaw, M. et. al. (1995). “Abstractions for Software Architecture and Tools to Support them.” IEEE Transactions on Software Engineering 21 (4): 314-335.

Shaw, M. (1997). “Software Architecture and Component-based Development”. Keynote address at the 5th International Symposium on Assessment of Software Tools (SAST’97), Pittsburgh, PA.

UML (1997) The Unified Modeling Language, Version 1.1, http://www.rational.com/uml/documentation.html.