# An Exception Handling Framework

Nikolas Nehmer
University of Kaiserslautern
Department of Computer Science
P.O. Box 3049, 67653 Kaiserslautern, Germany
nnehmer@informatik.uni-kl.de
+49 (0) 631 - 205 2644

## Abstract

*In software development processes, exception handling is often considered as an issue of minor importance. Explicit and systematic exception handling strategies and tool-support for reasoning about exception flow are rare. Especially unexpected exceptional events can seriously harm a software system. In this paper a novel exception handling framework is introduced. The framework includes a tool supporting developers in reasoning about exception flow. Based on the exception flow analysis a novel fault containment approach is proposed restricting the impact of uncaught exceptions on the overall system.*

**keywords**: exception handling, graceful degradation, dynamic fault containment, static code analysis, exception propagation
**submission category**: student forum

## 1 Introduction

Complex object-oriented software systems have to cope with an increasing number of exceptional conditions, raised by the environment or by faults persisting in the software itself. Developing fault-free software is nearly impossible and it's unwise to assume that the environment in which software operates always functions correctly [7]. Incorporating fault tolerance mechanisms into the system architecture is essential to meet dependability-related system requirements. Exception handling is one of the most important fault tolerance mechanisms for detecting and recovering from errors, and for structuring fault tolerance activities in a system by separating normal and exceptional control flow structures.

In complex object-oriented systems often more than two-thirds of the application code is devoted to detecting error conditions and to handling these erroneous situations [2]. Unfortunately, experience shows that especially exception handling code carries a high risk of being erroneous. The reason is complex. On the one hand this can be partially attributed to human behavior, i.e. developers being unable or unwilling to rigorously and carefully analyze rare and unlikely events. Furthermore software developers often misuse exception handling [7]. On the other hand, exception handling mechanisms are often error-prone and tool support for reasoning about exceptions is missing. "Real world" examples such as Windows blue-screens or the crash of the Ariane 5 [4] missile illustrate the possible impact of inappropriate exception handling on the overall system dependability.

Although Java is closely related to the ideal exception handling model by Garcia et al. [3] a closer look reveals many issues being the potential source of problems. Uncaught exceptions are a main issue. Furthermore exception handling effectiveness highly depends on the programming discipline, i.e. no compulsory specification of all exceptions possibly propagated by a method, issues related to type subsumption (e.g. implicit catches), long exception propagation paths and sloppy handler design (e.g. empty handlers). This clearly illustrated the need for a systematic and defensive approach to exception handling tackling two main aspects:

- Missing development tools supporting developers in reasoning about exceptions and exception flow
- Potentially catastrophic impact of uncaught exceptions

## 2 Goal Statement

The goal of this research is to tackle the two aspects stated above. Although in programming lan-

guages such as Java compilers help in reasoning about exception flow this support is insufficient. Many problems related to type subsumption or uncaught exceptions have to be detected and resolved by the developers manually during the development process. Even in deployed software systems these problems still persist undetected until they raise system failures. Especially in todays large and complex systems, systems of systems, component architectures or particularly in service oriented architectures reasoning about exception propagation and exception flow without proper tool-support seems to be unmanageable. Tools supporting software developers in this reasoning process by deriving exception propagation information from component code are strongly required and help in designing robust software systems by establishing sound exception handling mechanisms. Furthermore system runtime mechanisms have to be made available to provide means for automatic fault containment by building quarantine areas around "infected" data structures during runtime. Exception propagation beyond certain system boundaries has to be prohibited to contain the impact of uncaught exceptions. Components and data structures affected by exceptions not handled appropriately have to be identified and isolated during runtime. Access to the functionality provided by these data structures has to be restricted dynamically to reduce error propagation.

Accordingly, an exception handling framework including two highly interdependent and closely related approaches is proposed:

- Development of a graph-based static exception analysis tool for Java integrated into the Eclipse development environment
- Development of a runtime system mechanism for Java containing the impact of uncaught exceptions by prohibiting exception propagation and gracefully degrading system functionality

## 3 Approach

*Graph-based Exception Analysis* – Supporting developers by analyzing component code during development-time to detect exception handling related problems is the basic goal of the graph-based exception analysis approach. Static code analysis based on the abstract syntax tree (AST) of component code is applied. An AST is a logic tree (finite, labeled, directed) representation of a program's source code. A transparent representation of all exceptions possibly encountered within an arbitrary system scope is the outcome of such analysis. In object oriented programming languages methods are a reasonable scope for exception handling analysis. Exceptions encountered within a scope comprise exceptions explicitly raised by "throw"-statements, exceptions raised as the result of system operations, exceptions explicitly propagated from method calls and the set of uncaught exceptions implicitly propagated from enclosed scopes. This approach is based on existing exception flow analysis approaches [5, 10, 6, 9, 1].

To support static and dynamic reasoning about exception flow in the context of object oriented software development a general model of exception handling structures is mandatory – the general exception model (GEM). The model correlates exception handling structures in object-oriented languages that define exceptions as objects to their occurrence in a program's calling hierarchy. The goal of the model is to provide a unified basis for discussing problems related to the design, implementation, and maintenance of exception-handling structures, and for the analysis that can help alleviate these problems. The model is focused on the description of possible exception flows in a program. The general exception model is easily applicable to the Java programming language. The GEM is adapted from work by Robillard/Murphy [6] and Schaefer/Bundy [9].

A GEM-instance represents the information required to identify a program's exception flow. A data structure intuitively suitable to represent this information is a directed labeled graph – the exception propagation graph (EPG). The graph can be derived from a programs abstract syntax tree. The EPG represents a program's calling hierarchy and correlates exception structures (i.e. possibly raised exceptions and exception handlers) to their occurrence in the calling hierarchy. To properly reflect exception flow in the graph structure the graph is annotated. Nodes and edges are extended with exception path information (EPI) identifying all possible exception propagation paths from lower level nodes to the root node (system entry point).

Most modern integrated development environments such as Eclipse use abstract syntax trees (AST) as internal object representations of program code. Navigation and editing program code for example are performed based on this AST-representation. The proof of concept for the approach briefly described above is based on Eclipse framework's DOM representation (Eclipse's AST) of Java code. The prototypical exception analyzer is realized as an Eclipse plug-in and is tightly integrated into the Eclipse platform providing precise exception flow information for program code. The exception analyzer prototype is presently being

implemented.

*Fault Containment and Graceful Degradation in the Presence of Uncaught Exceptions* – Todays exception handling mechanisms exclusively located at the application level are highly error-prone. A default exception handling mechanism located at the runtime level guarding the application from the impact of uncaught exceptions is missing. An exception handling framework (exception guard) prohibiting exception propagation beyond well-defined system points and gracefully degrading system parts affected by uncaught exceptions is proposed. The framework is based on the EPG gained from the static code analysis.

Components representing a service are a reasonable scope for prohibiting exception propagation. A vital feature of any sound exception handling mechanism is the differentiation between internal exceptions to be handled inside the scope and the external exceptions which are propagated outside the scope [8]. All external exceptions potentially signaled by a component (i.e. propagated beyond a component's scope) should be declared in the component interface description. All possible return values – normal as well as exceptional – are part of the contract between caller and callee.

For every exception raised during runtime a system's EPG can be used to determine if an appropriate handler exists within the same scope without propagating the exception up the call stack. Internal exceptions identified as uncaught exceptions are converted into default external exceptions, declared in the component signature by default. Propagating well-defined external exceptions instead of unexpected internal exceptions leads to a well-defined and predictable service behavior. In today's systems service results under exceptional conditions might be unspecified and therefore unexpected, hard to interpret and recover from for callers.

Component parts affected by uncaught exceptions have to be gracefully degraded. Access to these component parts is either restricted or prohibited. Possible approaches are wrappers or the runtime injection of constraints. "Erroneous" object instances are identified by combining static information represented by the EPG with runtime information. Exception raising system states are analyzed by investigating call stack information and object states. This information can be used to identify potential error-raising conditions and prevent future exception occurrences.

## 4 Conclusion and Outlook

In this paper a novel approach to systematic exception handling is proposed. Static code analysis is applied to derive a graph-based exception flow representation by correlating a program's calling hierarchy to potential exception occurrences. Uncaught exceptions are identified during runtime, uncontrolled exception propagation is prohibited and affected system parts are gracefully degraded. The exception analyzer and a prototypical implementation of the exception guard are subject of ongoing work. Systematic fault injection into sample applications will be used to evaluate the framework comparing application behavior with and without support by the framework.

## References


[1] B.-M. Chang, J.-W. Jo, K. Yi, and K.-M. Choe. Interprocedural exception analysis for java. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 620–625, New York, NY, USA, 2001. ACM.

[2] F. Cristian. *Software Fault Tolerance*, chapter Exception Handling and Tolerance of Software Faults, pages 81–107. John Wiley & Sons, 1995.

[3] A. F. Garcia, C. M. F. Rubira, A. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *The Journal of Systems and Software*, 59(2):197–222, 2001.

[4] J.-M. Jézéquel and B. Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[5] D. Malayeri and J. Aldrich. Practical exception specifications. In C. Dony, J. L. Knudsen, A. B. Romanovsky, and A. Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 200–220. Springer, 2006.

[6] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, 2003.

[7] A. Romanovsky and B. Sandén. Except for exception handling .... *Ada Lett.*, XXI(3):19–25, 2001.

[8] A. B. Romanovsky. Exception handling in component-based system development. In *COMPSAC '01: Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, page 580, Washington, DC, USA, 2001. IEEE Computer Society.

[9] C. F. Schaefer and G. N. Bundy. Static analysis of exception handling in ada. *Softw. Pract. Exper.*, 23(10):1157–1174, 1993.

[10] S. Sinha, A. Orso, and M. J. Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 336–345, Washington, DC, USA, 2004. IEEE Computer Society.