Graceful Degradation in the Presence of Exceptions

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

Abstract

In today's software development processes, exception handling is mostly considered as an issue of minor importance. Furthermore, explicit and systematic exception handling strategies and tool-support for reasoning about exception flow and exception compatibility between components are very rare. Nevertheless exceptions not handled appropriately can seriously harm a software system – uncaught exceptions for example usually lead to a crash or shutdown of the whole system. In this paper a novel approach to systematic exception handling using appropriate tool-support is presented. Static code analysis is used to detect and identify exception handling related problems. Identifying and isolating affected data structures during runtime is a major part of this approach. A generic system-mechanism restricting error propagation to contain the fault's impact on the overall system is introduced. The overall objective is to achieve graceful degradation in the presence of exceptions of any kind.

keywords: exception handling, graceful degradation, dynamic fault containment, static code analysis, exception propagation
submission category: regular paper

1 Introduction

Exception-handling is a programming language construct or computer hardware mechanism designed to handle the occurrence of conditions that change the normal flow of program-execution. The condition raising the control-flow change is called an exception. In general exceptions are used for signaling error (exceptional) conditions.

Unfortunately, the presence failures[2] is a fact of life today's software has to deal with. Modern programming languages such as C#[9], C++[23] or Java[7] provide explicit exception handling mechanisms as programming language constructs to handle these exceptional conditions and to ease the difficulty of developing robust software systems. Exception-handling mechanisms provide means for raising exceptional conditions explicitly and means for specifying blocks of code to handle one or more of these exceptional conditions[6]. Handling exceptions includes a variety of different means; from sending simple error notification messages to client applications to complex forward error recovery mechanisms. In general, exception-handling mechanisms are intended to make developers design and build more robust and dependable software systems by separating code regions that handle unusual system behavior on the one hand from code that deals with normal processing on the other hand. This separation of concerns leads to consequences for system design. As exceptions are not necessarily handled at the place of their occurrence, exceptions can flow to an arbitrary point within the system and even beyond system boundaries if they are not handled appropriately. Developers must be aware of every exception type possibly raised at any program point or that can be propagated to it from lower-level modules. Some exception-handling techniques provided by current programming languages like Java help developers in this reasoning process by adding exception information to the method declarations, for example. Common compilers make use of exception declarations to check whether the corresponding exception handlers are provided by a client application.

Anyhow, this kind of exception-handling support provided by a multitude of programming languages is insufficient for several reasons.

- Unspecific throws – Modern object-oriented languages allow sub-typing of exceptions which may lead to unspecific throws.
- Implicit catches – The classification of exceptions into type hierarchies complicate reasoning about exception structures and may lead to implicit catches[17].

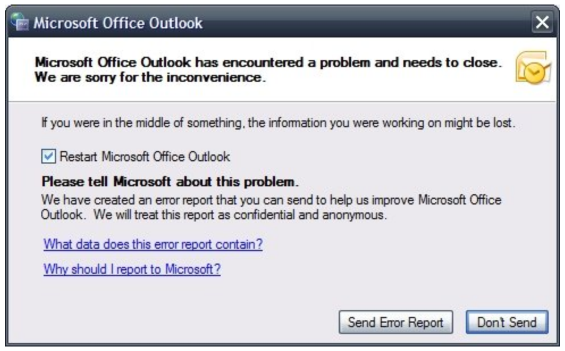
- Exception handling "ad hoc" – developers have to deal with exception handling without tool or infrastructure support. A default exception handling mechanism at the programming language runtime level is missing.
- Uncaught exceptions – In some programming languages like Java checked and unchecked exceptions are distinguished. Unchecked exceptions bypass the compiler-check and consequently developers of client modules are not forced to handle these exceptions types. This fact may lead to uncaught exceptions. If exceptions are not handled appropriately and hence flow through the system until they reach the system's entry point the program's execution will be aborted immediately. Crashing the entire software system is the usual fate of uncaught exceptions.

Uncaught exceptions may have the most fatal and obvious impact on the system behavior. But beyond that, there are further shortcomings regarding exception handling mechanisms in today's programming languages. The list below mentions a few:

- Stack trace propagation harms trust (e.g. in service oriented architectures based on today's web service technology)
- No clear responsibilities for exception handling
- Exception hierarchy is futile (often only applicable to academic examples)
- Without strict programming discipline, improper handling of exceptions will jeopardize the stability of the system
  - *Throw* – *catch* without handling – *rethrow* the same exception or a superclass (wrapping exceptions often only applicable to academic examples)
  - *Catch* {} with an empty handler
  - Decorating methods with "*throws Exception*" where Exception is the superclass of all exceptions
- Only a few reasonable handling patterns for exceptions (decentralized error handling leads to redundant code at different places e.g. user notification)
  - User notification
  - Graceful degradation
  - Application specific error handling
- Unstable interfaces (e.g. adding exceptions)
- Scalability issues (especially in systems of systems)

"Real world" examples such as Windows blue-screens or the crash of the Ariane 5[15, 12] rocket illustrate the possible impact of inappropriate exception handling on the overall system dependability. The impact reaches from data or content loss in office or mail applications to catastrophic crashes of air carriers or erroneous medical equipment compromising human life. This clarifies the need for a systematic and defensive approach to exception handling. In the following some meaningful examples are given.

Everyday life software - The first example that comes in mind when thinking of inappropriate exception handling is software used in everyday life such as windows, office applications or computer games. In windows uncaught exceptions often end up in the well known blue-screen leaving up the user with a listing of affected memory areas and a textual hint on how to restart the computer. In office applications or applications like computer games uncaught exceptions often just terminate the main application (including side effects such as loosing data) and a simple stack trace is displayed on a pop-up window. It seems that the exception handling strategy used in both examples does not improve the system overall dependability. An illustrating screen shot is given below in figure 1.



Microsoft Office Outlook has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

☑ Restart Microsoft Office Outlook

Please tell Microsoft about this problem.
We have created an error report that you can send to help us improve Microsoft Office Outlook. We will treat this report as confidential and anonymous.

What data does this error report contain?

Why should I report to Microsoft?

Figure 1. Microsoft Office Error

Therac 25 - The Therac 25[14] was a medical linear accelerator used as a radiation therapy machine for the treatment of cancer in several US and Canadian hospitals. It was involved in at least six known accidents between 1985 and 1987, in which at least six patients were given massive overdoses of radiation, three of them died afterwards. The failure of the radiation therapy machine is based on one of the most momentous software errors in history. The machine offered two operational modes - electron mode and photon (or x-ray) mode.

When operating in the photon mode, the machine was designed to rotate four components into the path of the electron beam - especially a target, which converted the high energy electron beam into low energy x-rays. The accidents occurred when the high-power electron beam was activated for x-ray therapy, without the target having been rotated into place. Consequently, a potentially lethal dose of radiation was applied to the patients. Each incident had a similar pathology. The operator initiated the treatment, but received an error message indicating that no dose had been applied. Accustomed to the machine's quirky behavior, operators were used to retry by pressing the "try again"-button several times. In fact, the radiation machines were applying a dose on each trial with radiation levels sometimes 30 to 100 times higher than originally desired. The root cause of this disaster was a complex chain of events. The software failures were mostly due to dynamic issues resulting from concurrency aspects including a race condition initiated by the operator's quick handling of the machine. Nevertheless, inappropriate exception handling displaying an error message indicating that no dose had been applied (which in fact was absolutely wrong), contributes to the chain of events.

Ariane 5 crash - On June 4th, 1996, the maiden flight 501 of the European Ariane 5[15, 12] launcher crashed about 40 seconds after takeoff. This crash was arguably one of the most expensive software errors in history causing a loss of roughly 0.5 Billion $. An international inquiry board analyzed the accident, identifying the root cause as a chain of events - an out-of-range data condition in a software module that was not even needed during the crash (reusing Ariane 4 software) raised an uncaught exception. This uncaught exception lead to a subsequent automatic shutdown of critical subsystems which lead to the catastrophic crash of the carrier. Again inappropriate exception handling caused an accident.

Exception handling is a very complex and poorly tool-supported task for software developers. Although in programming languages such as Java compilers help in reasoning about exception flow, many problems related to type-subsumption, unchecked exceptions or uncaught exceptions have to be detected and resolved by the developers manually during the development process and even still persist undetected in deployed software systems. Especially in today's 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. For some reason, exception handling is often not considered high priority when building systems. This is because architectures can be viewed as having two sides – positive and negative. The positive side includes algorithms and functions; the negative side includes handling exceptional conditions. Perhaps it is because exception handling does not deliver tangible functionality to users. System mechanisms supporting software developers in this reasoning process and in designing robust software systems by establishing sound exception handling mechanisms are strongly required.First of all, developers need tool- and infrastructure-support that helps deriving exception propagation information from component code. Secondly, 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.

In this paper a novel and systematic approach to coping with some of the shortcomings mentioned above is presented. Especially reasoning about exception propagation within applications using a graph-based static code analysis approach is emphasized. The aim of the graph-based analysis is to identify exception related problems such as uncaught exceptions and long exception propagation paths. The information extracted from the code analysis should be used for static code repair as well as for dynamic fault containment purposes collecting and isolating "infected" data structures during runtime as introduced in this paper. A default exception handling mechanism based on the concept of graceful degradation is proposed. The concepts presented in this paper are illustrated using the Java programming model but can easily be generalized.

The reminder of the paper is structured as follows. In Section 2 related work in the field of exception analysis is presented and discussed. Section 3 is threefold. It deals with a novel graph-based static code analysis approach and presents a generic approach to dynamic fault containment. Furthermore some basic principles and concepts defining the underlying exception model are introduced. Section 4 briefly summarizes and highlights the contribution of the paper. Furthermore it gives an outlook on future work.

2 Related Work

Various static analysis tools and concepts have been proposed to address problems related to exception handling in different programming languages making different assumptions on execution types and programming model. In this related work section Java related approaches analyzing synchronous exceptions are discussed. Of course, any Java compiler has limited exception handling analysis capabilities as well. For space reasons the discussion is limited to more advanced approaches.

Ryder et al.[20] have developed the Java Exception Static Profiler (JESP), a tool-suit meant to statically analyze the frequency of exception handling construct-occurrences in Java programs. Their studies show a frequent usage of exception handling constructs in Java programs and emphasizes the need for a more systematic approach to handling exceptions. Sinha and Herrold's[22] study on the occurrence of try and throw statements in a set of seven different Java programs confirm this statement.

Robillard and Murphy[18] have developed JEX, a tool for static code analysis of Java exception flows. The Jex-tool extracts information about the exception structure in Java programs, providing views of the exception structure and exception handlers that are present. Their approach includes checked exceptions as well as unchecked exceptions. Jex aims at providing exception information at any point in the program hierarchy and therefore is designed at statement level.

Chang et al.[5, 13] have introduced an interprocedural exception analyzer based on the set-based framework[8]. Their approach estimates uncaught exceptions independently of exceptions declared in the method header. Their analysis determines the types of exceptions propagated by a method and proves to be more precise than the one offered by the Java compiler. The analysis, however, does not include the flow of exceptions and is limited to checked exceptions. In [4] Chang et al. have presented a tool to visualize exception propagation on method level in Java programs. For a selected method the visualization displays textual information on uncaught checked exceptions.

The set-based approaches presented in the related work section are mostly limited to the analytical part supporting developers in reasoning about exception propagation and building more dependable and robust software. Static code analysis is used to detect drawbacks related to exception handling - solving the problems is still left up to the developer. The graph-based concepts proposed in this paper go one step further and introduces a default exception handling mechanisms as a fall-back strategy to cover uncaught exceptions or exceptions handled inappropriately on the application level.

3 Exception flow analysis and fault containment

Software developers have to be aware of type-subsumption, potentially raised exceptions, exception handlers and their interrelationship within the system's calling hierarchy at any arbitrary program point. Exception flow, exception dependencies or exception handling responsibilities within a system and even beyond system boundaries have to be transparently identifiable. The desired exception transparency can only be achieved by computer-aided support mechanisms. The required tool-support would serve software developers during the whole development process of a software system, especially during implementation and for failure analysis. For exception flow analysis several aspects are of great importance – implicitly and explicitly raised exceptions, exception handlers, the exception hierarchy, the calling hierarchy of a component and especially the interrelationship of these entities. This information can directly be derived from a program's source code. A program's abstract syntax tree (AST) is a logic tree representation (finite, labeled, directed tree) of a program's source code. Consequently, static code analysis based on an abstract syntax tree representation is used to exactly relate possible exception occurrences and corresponding exception handlers to code blocks in the calling hierarchy of a software system. A graph representation of the information given above can be derived by parsing a program's abstract syntax tree. This graph structure can be used to gain exception-flow information. A static analysis of this graph can be applied to detect exception related problems like uncaught exceptions, long propagation paths, unspecific catches or exception incompatibilities between components.

Today's exception handling mechanisms exclusively located on the application level are highly error prone as illustrated in the introduction. Consider an uncaught unchecked arithmetic exception raised in a subcomponent (as it was the case in the Ariane 5 accident). This exception would normally flow to the system entry point and terminate the system runtime. Consequently, some general fall-back mechanisms have to be established to back-up customized application level exception handling mechanisms. In general, these default exception handling mechanisms will not substitute customized application specific exception handlers but restrict error impact in the overall system if customized exception handlers are inappropriate or just missing. In the example given above the uncaught exception should be detected immediately, caught automatically and access to the data structures affected by the exception should be prevented. Generic exception handling functionality has to be provided by frameworks or has to be integrated into programming language runtimes and compilers directly. These runtime mechanisms have to cope with problem cases such as uncaught exceptions based on a generically applicable mechanism. Automatic fault containment is one approach of generically handling errors and restricting their impact on the overall system dynamically during runtime.

The problem cases detected by a static code analysis can be tackled in different ways. Obviously, the problems can be solved by ruling out the root cause and adapting or changing the source code. This approach is mostly applicable during development (optionally during maintenance) for systems with full code access only. But at this point several questions should be raised. Does a developer really want to tackle all the problems by ruling out the root cause? Is it reasonable to handle all kinds of exceptions by application specific error handlers? Regarding these questions, at first, checked and unchecked exceptions have to be distinguished conceptually. In Java for example checked exceptions are usually handled in some way because the compiler enforces checked exceptions to be treated - either locally or by upward propagation. For some reason, exception handling is often not considered high priority in a software development process. Accordingly, even checked exceptions are not handled appropriately resulting in runtime problems described above (long propagation paths, imprecise catches, etc.). In general unchecked exceptions such as runtime exceptions (e.g. null pointer exceptions or array out of bounds exceptions) or system errors (e.g. io-exceptions) can hardly be handled appropriately on the application level. Hence, this type of exceptions is mostly ignored by developers regarding exception handling. Nevertheless an uncaught runtime exception or system error will also lead to the problems discussed earlier such as system crashes. To avoid these extensive consequences in both cases (for checked and unchecked exceptions) exceptions should be analyzed and a systematic and automatic default system-level exception handling mechanism (i.e. on an abstraction layer below the application level such as the programming language runtime) has to be provided.

A systematic and automatic exception handling mechanism located on the system level calls for a generic exception handling approach. Consequently, this means that recovery mechanisms such as application specific forward recovery that require semantics are inappropriate. Approaches based on semantic context information are hardly applicable to the general case. A generic exception handling mechanism gracefully degrading system functionality by avoiding uncaught exceptions and improving fault containment by identifying and isolating affected code blocks is required. When an exception is raised during runtime the affected data structures have to be identified and access to these data structures has to be restricted or completely prevented. The question if a raised exception can be handled by application specific handlers and the identification of affected data structures can be supported by using the graph-structure resulting from the exception flow analysis. Access restriction can be achieved by dynamically activating pre-deployed protective wrappers during runtime. The wrappers' duty is to build a quarantine area around affected data structures by restricting client access.

Both approaches – "development"-tools and runtime mechanisms are based on an analysis of the exception flow. Exception flow information can be derived from a system's source code directly. Basically, both approaches are based on a detailed abstract syntax tree analysis to detect exception related problem cases. During the analysis the abstract syntax tree is transformed into a simplified and more abstract graph representation containing exception flow related information - calling hierarchy, exception hierarchy, exceptions and exception handlers and their relationship (as depicted in figure 2).

3.1 Exception Model

The concepts proposed in this paper are based on some assumptions expressed in the underlying exception model briefly described in this section. Basically, the approach presented in this paper is illustrated using Java (not including multi-threading) as a reference programming language. Consequently, the approach is based on the Java programming and exception model but it can easily be adapted to other programming languages. A detailed description of the Java exception and programming model goes beyond the scope of this work. The key assumptions are:

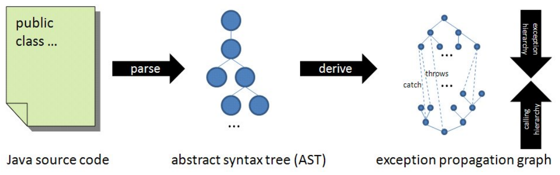


Figure 2. Basic Strategy

- Exceptions as objects (including type subsumption) [1]
- Termination model[3, 17]
- Checked and unchecked exceptions including programmed application specific exceptions, errors and runtime exceptions (aka implicit and explicit exceptions)
- Synchronous exceptions (asynchronous exceptions raised by concurrent threads for example are beyond the scope of this paper)

3.2 Detailed Strategy

This section gives a detailed insight into graph-based exception handling analysis and a dynamic fault containment approach in the presence of exceptions. Exception handling analysis can be regarded as a compile-time issue while fault containment is considered a runtime issue.

3.2.1 Compile-time Issues – Static code analysis

The basic idea is to analyze component code to detect exception handling related problems during compile-time. A transparent representation of possible exceptions raised, handled and handed on at any arbitrary program point is the outcome of such analysis. In object oriented programming languages a reasonable scope for exception handling analysis are methods synthesizing all exceptions encountered within the scope of the method, the exceptions caught within the scope including subtypes and the exceptions thrown or handed on beyond the scope of the method (including uncaught or undeclared exceptions). The static code analysis is based on the abstract syntax tree of component code. Information about exceptions required to identify exception flow includes:

- method call dependencies – aka system calling hierarchy

- the scope and hierarchy of try blocks
- the expression raising the exception
- contains-relationships between method calls, expressions and try blocks
- the exact location in the calling hierarchy where exceptions are raised
- exact location in the calling hierarchy of exception handlers
- the exception identifier and the exception hierarchy to reason about type-subsumption

A graph containing and relating this information – from now on called the exception call graph (ECG) – can be directly derived from a component's abstract syntax tree. The ECG represents a component's calling hierarchy related to possibly raised exceptions and corresponding exception handlers. Unfortunately there is no common AST in Java provided by Sun. The proof of concept for the approach proposed in this paper will be based on eclipse framework's DOM representation of Java code. To properly analyze the ECG regarding exception related problems the ECG has to be extended by exception propagation information. Therefore every node and edge within the ECG is annotated with exception information and exception propagation information identifying the exception's source and the propagation path. The annotated version of the exception call graph (ECG) is known as the exception propagation graph (EPG). Both, ECG and EPG, are directed labeled graphs potentially containing cycles.

To support static and dynamic reasoning about exception flow in object oriented languages a general model of exception handling structures and their occurrence in a program's calling hierarchy is presented (hence referred to as exception call graph model - ECGM). The goal of the model is to provide a unified and sound basis for discussing problems related to the design and implementation of exception handling mechanisms. Furthermore it eases reasoning about exception handling related problems by abstraction. The exception call graph's and the exception propagation graph's structures are based on this model. The model is adapted from the work by Schaefer and Bundy[21] and closely related to the work by Robillard and Murphy[19]. The focus of the model is on the description of all entities required for the modeling of exception propagation in a program. The model is depicted in figure 3 and a detailed description is given below.

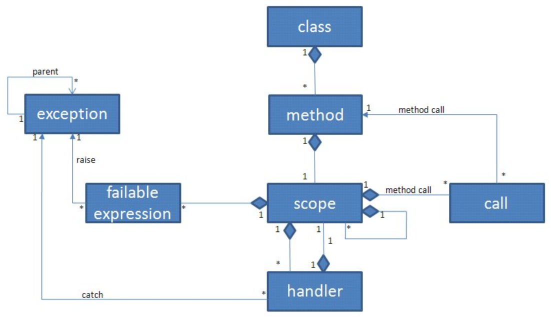


Figure 3. Exception Call Graph Model

In the structural exception call graph model (ECGM) object oriented language constructs are mapped to model entities. Relationships between entities are modeled as directed associations and compositions. Classes and methods are simply mapped to classes and methods in the model where one class is composed of zero to n methods and a method belongs to exactly one class. According to the UML-meta-model all “has-a”-relationships in the model are strong association relationships corresponding to compositions in the UML meta-model. A composition has a strong life cycle dependency between instances of the container entity and instances of the contained entity. A method in the ECGM always corresponds to exactly one scope. A scope is defined as an atomic control flow sphere. Every exception encountered within scope not associated to a handler flows to the boundary of the scope without further modifying the control flow of the program. In the following an expression potentially raising an exception is referred to as failable expression - this includes e.g. arithmetic expressions raising arithmetic exceptions such as division by zero or throw statements raising new exceptions. A scope consists of zero to n method calls and zero to n failable expressions. A scope potentially contains zero to n further guarded scopes. A ”guarded” scope is defined as a scope explicitly preventing (or catching) explicitly defined exceptions from propagating to the enclosing scope. In the exception graph model a guarded scope manifests itself by strongly associating one to n exception handler entities. Handlers and failable expressions are assigned to exactly one scope and are directly associated with an exception entity expressing the exception raised or handled respectively. Every handler contains a new default scope. Exceptions are associated to themselves to represent the exception type hierarchy. A method call is associated to exactly one method. Possibly exceptions propagated by a method call are mapped by the transitive relation between method, scope and all containing entities. To ease the understanding of the model and to support reasoning about exception propagation some definitions are given below:

- *scope s* = (C_s, F_s, S_s, H_s) where C_s is the set of all method calls directly contained in the scope s, F_s is the set of all failable expressions directly contained in the scope s, S_s is the set of all scopes directly contained in the scope s, H_s is the set of all handlers associated with scope s

- (s_D) = the default scope not associated with a handler
- *method m* = (s_D) where s_D is the default scope
- *handler h* = (s_D) where s_D is the default scope
- *failable expression f* := expression potentially raising an exception
- *call c* := control flow transition to a method
- *exception e* := entity representing an exceptional condition
- $E_x = \{e|e \text{ raised in the context of x}\}$

Furthermore, some functions are defined as follows:

$$encounters : S \rightarrow E \tag{1}$$

$$encounters(s) = raises(s) \cup propagates(s) \cup uncaught(s) \tag{2}$$

$$raises : S \rightarrow E \tag{3}$$

$$raises(s) = \bigcup_{\forall f \in F_s} raises(f) \tag{4}$$

$$\text{where } raises(f) = \{e|e \in E_f\} \tag{5}$$

$$propagates : S \rightarrow E \tag{6}$$

$$propagates(s) = \bigcup_{\forall c \in C_s} propagates(c) \tag{7}$$

$$\text{where } propagates(c) = encounters(s_D) \tag{8}$$

$$uncaught : S \rightarrow E \tag{9}$$

$$uncaught(s) = encounters(s) - catch(s) \tag{10}$$

$$catches : S \rightarrow E \tag{11}$$

$$catches(s) = \bigcup_{\forall h \in H_s} chatches(s) \tag{12}$$

$$\text{where } catches(h) = \{e|e \in E_h\} \tag{13}$$

To illustrate how the exception handling mechanisms and the calling hierarchy of a real programming language are mapped onto the general exception call graph model and to ease the understanding of the concepts proposed in the rest of this paper, the Java language constructs are described in terms of the general model. In Java every class and method corresponds to a class and method entity in the exception call graph model. A Java method spans the default outer scope. Every try-block within the method (outer scope) is mapped to a guarding scope associated to the corresponding exception handler entities. Every try-block may contain further try-blocks expressed by nesting scopes in the model. A handler in the model corresponds to a catch-block in Java; the associated exception matches the exception caught by the catch expression. A handler spans a new default scope possibly containing further guarded scopes (aka try-blocks). A Java method or try-block in general contains method calls or failable expressions. Failable expressions in Java are expressions implicitly or explicitly raising exceptions. Explicitly raised exceptions correspond to throw statements; implicitly raised exceptions correspond to exceptions that can be raised by the runtime as shown in the Java Language Specification[7]. A method call propagates all exceptions possibly encountered within a method‘s scope and a method scope's child scopes that are uncaught. The mapping of Java language constructs to the exception call graph model is informally summarized in the following:

- Java class → ECGM class
- Java method → ECGM method + ECGM scope
- Java try-block → ECGM scope
- Java catch → ECGM handler
- Java exception → ECGM exception
- Java potentially exception raising expression → ECGM failable expression
- Java method call → ECGM method call
- Java finally-block → ECGM scope
- Java throw statement → ECGM failable expression

Basically, the ECG's and EPG's structure is based on the exception call graph model. Graph node types correspond to the entities described by the model. Graph edges are modeled by associations within the ECGM. Deriving the ECG from the abstract syntax tree is a straight forward approach. The abstract syntax tree is parsed and the required information is extracted. According to the mapping rules described above step by step the statements in the abstract syntax tree are parsed and new nodes with corresponding directed edges are inserted. The exception hierarchy is directly derived from the abstract syntax tree as well. Finally, the ECG represents a program's calling hierarchy consisting of class, method and scope nodes mainly. Method calls are represented by directed edges to the corresponding method nodes. Exceptions raised explicitly or implicitly or caught are embodied by directed edges to the adequate exception nodes.

To support reasoning about exception propagation the ECG containing the required exception and call structure of a program as described above is analyzed and annotated with exception path information (epi). Of course, the exception path information contains the exception type as well as the exception source. If an exception is propagated the exception path information includes the exception propagation path according to the calling hierarchy. According to [4] the exception path information is formally described as:

$$\textit{epi} = \textit{exception} \times \textit{propagation path} \tag{14}$$

$$\begin{aligned}&\text{where } \textit{exception} = \textit{exception identifier} \times \textit{expression}\\ &\text{and } \textit{propagation path} = \textit{method*}\end{aligned} \tag{15}$$

To annotate the exception path information to the exception call graph the following algorithm is applied. Starting from the exception nodes the directed edges are traced back and the exception path information is annotated to the edges. Reaching a node, the exception path information contained in all the node's outgoing edges is subsumed and exceptions handled by a corresponding handler are removed building a node's exception path information set (epis). Consequently, every node is annotated with a set of potentially raised exceptions containing the exception source and propagation path. All the node's incoming edges are traced back again and the edges are annotated with the exception path information subsumed in the node's exception path information set. This algorithm is repeated until the system's entry point is reached. The exception propagation information set at the root node (system entry point) exactly shows which exceptions are potentially propagated beyond the scope of the system and exactly show the exceptions' origins and propagation paths. A pseudo code illustration of this simplified algorithm is depicted in listing 1.

The algorithm described in listing 1 is simple and straight forward and terminates for acyclic graphs. But general calling hierarchies as represented by the ECG might contain cyclic references – i.e. programs containing recursive or cyclic method calls. If a graph contains cycles the algorithm shown above will not terminate. An improved annotation algorithm has to mark already visited nodes to detect cyclic relationships and stop to follow such paths again. The exception call graph (ECG) and exception propagation graph(EPG) respectively can be regarded as a deterministic finite state machine or automata (DFA) represented by the 5-tupel $A = (Q, \Sigma, \delta, q_0, F)$ [11], where

- Q is the finite set of states represented by methods
- Σ is the finite set of symbols represented by method calls

```
1  Start at every exception node
2  Annotate the exception path information(-
       sets) to all incoming edges
3  Trace back all incoming edges
4  Reaching a node, adapt and extend exception
       path information according to:
5  switch (node.type) {
6          case failable expression:
7                  add expression information
                       to the exception path
                       information
8          case method:
9                  add method information to
                       the exception path
                       information
10         case handler:
11                 negate exception path
                       information
12 }
13 Build a set from all incoming exception
       path information(-sets) excluding
       negated exception path information (
       considering sub-typing!)
14 Continue with step in line 2
15 The algorithm stops when reaching the entry
       point
```

Listing 1. Exception Propagation Information – Annotation Algorithm

- δ is the transition function $\delta : Q \times \Sigma \rightarrow Q$
- q_0 is the start state
- F is a set of states of Q (i.e. $F \subseteq Q$)

Finite state machines are closely related and correspond to regular expressions. Regular expressions can be seen as algebraic descriptions of languages. It can be proved that if $L = L(A)$ for an arbitrary DEA A there is a corresponding regular expression R with $L = L(R)$ where L is a language over Σ [11]. Consequently, the propagation path expressions can easily be expressed by using regular expressions representing the potentially cyclic subgraph.

Basically, in this subsection a novel graph-based approach to exception flow analysis was presented. A graph representation (exception call graph) of a program's calling hierarchy, related exceptions and exception handlers are derived from the abstract syntax tree. To support reasoning about exception propagation this graph structure is annotated with exception path information resulting in the exception propagation graph. This interrelation is depicted schematically in figure 4.

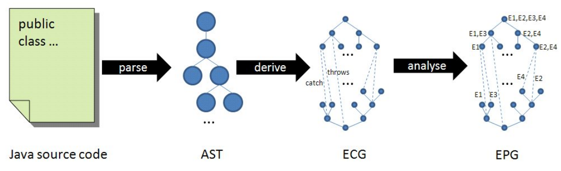


Figure 4. Exception Flow Analysis

```
class MyDemo{
 public static void main(String[] args )
     throws E2 {
  try {
   myMethod1( );
  } catch (E1 e) { ; }
  . . .
  myMethod3( );
 }
 void myMethod1( ) throws E1{
  myMethod2( );
 }
 void myMethod2( ) throws E1{
  throw new E1();
 }
 void myMethod3( ) throws E2 {
  if (...) throw new E2();
  if (...) myMethod3( );
 }
}
```

Listing 2. Java Code Example

To ease the understanding of the static code analysis proposed in this paper a small Java code fragment is given in listing 2.

The annotated exception propagation graph corresponding to the Java code fragment shown in listing 2 is illustrated in figure 5.

3.2.2 Runtime Issues – Dynamic Fault Containment

As already illustrated there is a strong need for default exception handling mechanisms located on a software layer below the application layer itself. These default mechanisms are not meant not replace customized application specific exception handlers but complement them. When an exception is raised during runtime the possibly affected data structures have to be detected directly. This means the identification has to be performed before the exception objects are created and especially before the exception objects are propagated along the stack trace. It is not sufficient to only identify object types but specific object instances. During runtime when the exception is raised object references to all affected data structures have to be collected by a system mechanism. This feature has to be supported by the program language's runtime environment. The information on the affected data structures are added to the information in the exception objects. Today exception objects mainly include stack trace information used for debugging purposes when an exception leads to a system crash. Extending exception objects with references to the affected data structures will help in developing reasonable exception handling mechanisms. Often, context information to understand the exception on a higher abstraction level is missing anyway. Besides the information that something went wrong signaled by an exception, information on affected data structures is essential.

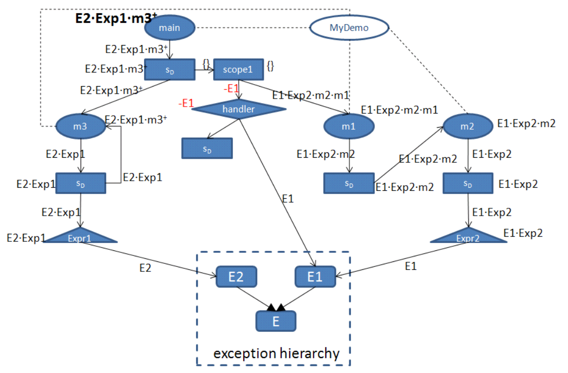


Figure 5. Exception Propagation Path Example

After the data structures have been identified, client access has to be prevented or restricted automatically. Protective wrappers guarding the access have to be built around the quarantine areas dynamically during runtime. In a first prototype implementation the wrapping of data structures can be achieved by providing or enhancing objects with constraints. These constraints implemented as preconditions[10, 16] are true by default and can be adapted during runtime. If the access to an object is to be prevented, the precondition is simply changed to false. Between these two extremes – full access and no access – arbitrary access restrictions embodied by formal boolean expressions are possible. If an exception can be assigned to client's wrong input parameters for example, further calls with the same parameters can be blocked by including the parameters into the precondition. The constraints defined for each object must be maintained during execution unless they are explicitly waived by the application. If a client application's access to a guarded object is declined a default exception has to be raised. Every client commits to catch and handle this default exception.

For every exception raised during runtime an analysis can be performed to check if a corresponding exception handler exists on the exception's way up the stack trace.

This runtime analysis is based on the exception propagation graph (EPG) resulting from the static code analysis introduced in the section on static code analysis. Basically, the exception identifier (of the exception currently raised) has to be concatenated with the stack trace information representing the call path to the system point where the exception has been raised. Combining these two pieces of information results in an expression type similar to the exception path information introduced above ($epi = exception \times propagation\ path$). As described earlier every node in the exception propagation path is annotated with a regular expression representing the exception path. To analyze if an exception currently raised will be caught on it's way up the stack trace, this exception's actual exception path simply has to be matched to the set of exception path regular expressions (epis) at the root node of the system or component respectively. The exception path information set contains all exceptions with the corresponding exception sources and propagation paths propagated beyond the analyzed node in the calling hierarchy. Consequently, if no match can be found the exception is definitely caught by a corresponding handler on the propagation path and vice versa. The matching can be easily achieved by a generated finite state machine representing the regular expression. The exception path to be matched can be regarded as a word in the sense of formal languages that has to be accepted by the finite state machine describing the language[11]. If there is no match at the root node of the system it is assured that the exception is handled by an application-level exception handler. In this case the exception is propagated up the stack trace as usually done in today‘s exception handling mechanisms. After handling the exception in the application-level exception handler the automatic fault containment has to be revoked by removing the guarding wrappers. If a match can be found it is assured that the exception will not be caught on it's way up the stack trace. In this case two steps have to be taken – graceful degradation by restricting access to affected data structures and transforming the uncaught incompatible exception into a compatible default exception. The former step has already been initialized directly after the exception was raised. Transforming the exception into a default exception and adding references to affected data structures is straight forward. Again a client has to commit to catch the default exception.

During the life cycle of a system fault containment data accumulates. This data has to be cleaned up by a kind of garbage collection. As already explained wrappers related to exceptions already handled can be removed directly. Furthermore a mechanism to store fault containment data is required to persist this information. Housekeeping and persistence are in the scope of further investigation.

4 Conclusion

Exception handling is often not considered high priority when developing software systems. Systematic exception handling strategies and tool support for reasoning about exception flow and exception compatibility between components are rare. Accidents such as the Ariane 5 crash illustrate the possible impact of inappropriate exception handling on the overall system dependability. In programming languages such as Java compilers help in reasoning about exception flow. Nevertheless many problems such as unchecked exceptions or long exception propagation paths have to be detected and resolved by the developers manually during the development process and even still persist undetected in deployed software systems.

In this paper a novel approach to systematic exception handling based on a sound exception model is presented. Reasoning about exception propagation within applications using a graph-based static code analysis approach is emphasized. The overall objective of the graph-based analysis approach is the identification of exception related problems such as uncaught exceptions and long exception propagation paths. Based on a program's abstract syntax tree the exception call graph (ECG) is derived – a graph representation of the calling hierarchy, raised exceptions, the exception hierarchy, exception handlers and their interrelationship. The exception call graph is extended with annotations representing exception propagation paths resulting in the exception propagation graph (EPG). The exception propagation graph directly reveals problems such as uncaught exceptions. Furthermore, the EPG supports dynamic fault containment during runtime detecting, collecting and isolating "infected" data structures. The concepts proposed in this work are applicable to any object-oriented programming language that defines exceptions as objects. The following listing briefly summarizes the contribution of this work:

- a novel graph-based exception analysis approach based on static code analysis
- a novel concept improving current exception handling mechanisms by extending programming language's exception handling constructs
- a novel generic fault containment approach proposed as a system-level default exception handling mechanism based on the concept of graceful degradation

Future investigations will focus on:

- the development of a graph-based static code analysis tool,
- the implementation of the generic fault containment mechanism and

- the evaluation of the concepts introduced in this paper by applying them to sample applications tested under various injected fault conditions.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure comput. *IEEE Trans. Dependable Sec. Computing*, 1(1):11–33, 2004.

[3] P. A. Buhr and W. Y. R. Mok. Advanced exception handling mechanisms. *IEEE Trans. Softw. Eng.*, 26(9):820–836, 2000.

[4] B.-M. Chang, J.-W. Jo, and S. H. Her. Visualization of exception propagation for java using static analysis. page 173, 2002.

[5] 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.

[6] J. B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[8] N. Heintze and J. Jaffar. Set constraints and set-based analysis. pages 281–298, 1994.

[9] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[11] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

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

[13] J.-W. Jo, B.-M. Chang, K. Yi, and K.-M. Choe. An uncaught exception analysis for java. *J. Syst. Softw.*, 72(1):59–69, 2004.

[14] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[15] J. Lions. Ariane 501: Report by the inquiry board. 1996.

[16] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[17] R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. *Lecture Notes in Computer Science*, 1241:85–103, 1997.

[18] M. P. Robillard and G. C. Murphy. Analyzing exception flow in java programs. *SIGSOFT Softw. Eng. Notes*, 24(6):322–337, 1999.

[19] 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.

[20] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah. A static study of java exceptions using jesp. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 67–81, London, UK, 2000. Springer-Verlag.

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

[22] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Softw. Eng.*, 26(9):849–871, 2000.

[23] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.