EXCEPTION HANDLING MECHANISMS
IN FEATURE LANGUAGE EXTENSIONS

BY

YIMENG LI

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering in the Graduate College of the Illinois Institute of Technology

Approved ______________________________
Advisor

Approved ______________________________
Co-Advisor

Chicago, Illinois
May 2006
ACKNOWLEDGMENT

This thesis could not have been written without Dr Francis Leung who not only supervised my work but also encouraged and challenged me throughout my M.S. program. Dr Leung and other ECE faculty members, Dr Jafar Saniie and Dr Thomas T. Y. Wong guided me through the dissertation process. I would like to extend my heartfelt thanks to all of them.

I sincerely thank Dr Francis Leung, Research Professor, CS Department, Illinois Institute of Technology for his support. Dr Leung gave me this opportunity to work on state-of-art technologies with an excellent research atmosphere.

From the bottom of my heart, I thank my parents who support and encourage me, and above all love me all the time. Every piece of my accomplishment today is fulfilled with their effort and love.

Finally, I express my thanks to Dr Leung’s research group especially my team mate Mr Manohar Dudda, Ms Lu Zhao and Mr Kathik Ramachandran.
# TABLE OF CONTENTS

<p>| ACKNOWLEDGEMENT                               | iii |
| LIST OF FIGURES                               | v  |
| ABSTRACT                                      | vi |
| CHAPTER                                       |    |
| 1. INTRODUCTION                               | 1  |
| 2. PROBLEM DESCRIPTION                        | 4  |
| 2.1 Exception Definition                      | 4  |
| 2.2 Exception Handler Specification           | 5  |
| 2.3 Exception Reporting and Propagation       | 5  |
| 2.4 Exception Return Models                   | 6  |
| 2.5 Objectives                                | 7  |
| 3. BASIC CONSTRUCTS OF FLE                    | 8  |
| 4. FLE EHM                                     | 13 |
| 4.1 FLE Exceptions                            | 13 |
| 4.2 Exception Handler in FLE EHM              | 15 |
| 4.3 Exceptions Reporting and Propagation      | 17 |
| 4.4 Exception Handling Return Models          | 20 |
| 5. RELATED WORK                               | 24 |
| 6. CONCLUSION                                 | 27 |
| BIBLIOGRAPHY                                  | 27 |</p>
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Resumption Model</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Simulate Resumption Model with Termination model</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>The Domain Statement for Call Processing</td>
<td>8</td>
</tr>
<tr>
<td>3.2</td>
<td>A portion of the FLE POTS code</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>Feature DoNotDisturb</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>QuietPhone feature package</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>BasicTelephony enhanced with exception declarations</td>
<td>14</td>
</tr>
<tr>
<td>4.2</td>
<td>Exception Feature Syntax</td>
<td>15</td>
</tr>
<tr>
<td>4.3</td>
<td>The CatchAll exception feature</td>
<td>17</td>
</tr>
<tr>
<td>4.4</td>
<td>The DamageControl exception feature</td>
<td>18</td>
</tr>
<tr>
<td>4.5</td>
<td>The RobustSelectiveForwarding feature package</td>
<td>18</td>
</tr>
<tr>
<td>4.6</td>
<td>Throwing Exceptions</td>
<td>19</td>
</tr>
<tr>
<td>4.7</td>
<td>Mixed Precedence</td>
<td>20</td>
</tr>
<tr>
<td>4.8</td>
<td>Retry Model Example</td>
<td>22</td>
</tr>
<tr>
<td>4.9</td>
<td>Resumption Model Example</td>
<td>22</td>
</tr>
</tbody>
</table>
ABSTRACT

Exceptions are unusual events that occur during program execution. In many applications, significant portion of the software written is devoted to handling exceptions. Despite modern programming language support, exception handling programs are difficult to develop: they contain more bugs than normal processing code and are typically not reusable. Exception handling programs entangle with and scatter among normal processing programs. Propagation of an exception event to its handler by stack unwinding increases the complexity of the programs making them difficult to develop and debug, especially with nested code. Support for return models after exception handling is incomplete resulting in awkwardly written programs.

In this investigation, we introduced the exception handling mechanism (EHM) of the Feature Language Extensions (FLE) that is designed to enable the programmer to write reusable feature programs without entanglement. FLE EHM allows the programmers to develop normal processing code and exception handling code in separate program modules. This clean line of separation enhances the reusability and portability of exception handling programs and normal processing programs. Exception propagation is treated similar to other FLE program events. As a result, the programmer does not need to master another programming paradigm. In addition, we provide support to all three basic exception handling return models to meet the needs of the applications. An implementation of the FLE EHM exists and it has been applied to some fairly complex FLE applications.
CHAPTER 1
INTRODUCTION

Exceptions are events which disrupt the normal flow of control during the execution of a program. An exception can occur synchronously at specific points in the execution of a program. For example, an exception will occur when a program trying to open a file that does not exist. An exception can also occur asynchronously or randomly at any point during the execution of a program such as when a child process spawned by the program is killed. In practice, most exceptions are used to report error conditions.

Exception handling mechanisms (EHM) are programming language facilities that enable the programmer to specify program behavior after exceptions have been detected. EHM is an important research topic since exception handling programs often occupy as much as or even more than the programs for normal processing, especially for applications that require fault tolerance.

Before the idea of EHM, return value and status flags are widely used to report exceptions. Using this method, the programs of normal processing and exception handling entangles severely. The return values and status flags must be tested frequently. In this investigation, we say the programs of different concerns (features) entangle with each other if they intertwine in the same reusable program unit (e.g. method) of the programming language.

Modern programming languages such as C++ [19] and Java [6] separate the normal processing code in a try block and exception handling code in a catch block. They free programmers from tedious testing for return values and status flags and free the computer from mostly wasting CPU cycles; exceptions are thrown when error condition occurs.
Despite the improvements, exception handling programs remain difficult to develop. Researchers have observed that exception handling code is more likely to contain bugs than any other parts of the program [5] [23]. Exception handling programs written in existing EHMs are usually not reusable [15] [10]. When adding or deleting exceptions or when their handling policies have changed, any program that call or transitively call the programs from which the exceptions are thrown may have to be changed. Exception handling is often a main reason that makes it difficult to port normal processing program from one platform to the other. For example, a networked application can run on either sockets or remote procedure call. But the two middleware generate large amount of very different exceptions and as the exception handling code are entangled with the application code, porting the application will usually require large amount of modification to existing code.

A solution to the above problems must allow the programmer to develop exception handling programs and normal processing programs as separate and reusable program modules. When normal processing and exception handling programs can be separated, the complexity of the programs is reduced; changes in exceptions and exception handling policies, even when they are due to platform changes, affect only the exception handling programs, leaving normal processing programs untouched.

To enable the programmer to develop normal processing and exception handling as separate and reusable modules is the design objective of the EHM for programs written in the Feature Language Extensions (FLE) [14]. FLE is a set of programming language constructs that enables nonprocedural programming in which the programmer does not specify the execution flows of program units. FLE is designed to allow the programmer to write separately reusable program modules that would have been entangled if they are written in an existing general purpose programming language.
Some of the results reported here were first proposed by Mr. Kathik Ramachandran [20], specifically the notion of exception feature and the interaction resolution among exception features. Many of the concepts described in [20] has been extended and clarified. For examples, original proposal requires all exceptions that appear in a domain must be declared in domain statement. Practice shows that due to large among of exceptions, it is not practical and unnecessary. Domain specific exception concept and semantics are redefined in the new proposal. More new contributions are included in this investigation, such as clarification on the propagation mechanism and three basic exception handling models, and all the implementations.

The layout of the rest of the thesis is as follows: Section 2 gives a more detailed discussion of the problems facing the design of EHM in general and our approach to those problems in particular. Section 3 gives a brief introduction to FLE. The materials in this section mainly came from [14] Details of FLE EHM are covered in section 4 and an example is presented at last. Section 5 makes a comparison between our approach and other related work in this field. Most of the FLE EHM features that are described in this thesis have been implemented and incorporated into the FLE compiler.
CHAPTER 2
PROBLEM DESCRIPTION

Modern EHM mainly consists of the following components: (1) language constructs to define exceptions, (2) language constructs to specify the exception handling programs, commonly called the exception handlers, (3) mechanisms to report and propagate an exception to the program that handles it, and (4) a return model to indicate where the execution flow goes to after the exception is handled. We will discuss the existing art in these components.

2.1 Exception Definition

An exception usually contains diagnostic information of what the event is and sometimes also why it is raised and how it should be handled. The information is known as exception parameters. In object oriented languages, such as C++ and Java, exceptions are objects. When an exception instance is created, its parameters are initialized. An exception can be derived from another exception using inheritance. For instance, RemoteException and SocketException are both derived from IOException. The programmer can choose to handle IOException or more specifically to handle the derived exceptions. In general, the programmer can handle the exceptions at any desired degree of specificity along the hierarchy. This makes exception handling much more flexible.

Java EHM applies type checking to a subset of the exceptions that can occur in a Java program. Type checking is helpful because it makes sure that the type checked exceptions are handled. Java names the other exceptions as runtime exceptions. Runtime exceptions can occur anywhere in a program and are numerous in most cases. Java does not apply type checking to them as a compromise, as forcing the programmers to deal with all of them in every method will be tedious and
overwhelming.

2.2 Exception Handler Specification

The program that deals with the exceptions is called the exception handler. It analyzes the exceptions according to the exception parameters and performs the corresponding operation such as logging the error, recovering from the error if possible or re-throwing the exception.

In Java, C++ and many modern programming languages, normal processing code is put into try blocks while their corresponding exception handlers resides in catch blocks. In a Java or C++ program, we often find a few lines of try block code followed by many lines of catch block code that followed by another try-catch block. The normal processing code and exception handling code entangle, and the program is hard to read and maintain. The situation becomes much worse when the try-catch blocks are nested. Changes to exception handling code often affect normal processing code and vice versa.

Code entanglement directly results in poor reusability and portability of programs. Since the exception handling code is embedded with normal processing code in the same reusable program unit, the normal processing code cannot be reused if its corresponding exception handling code does not meet the requirement of the new environment (such as platform and application).

2.3 Exception Reporting and Propagation

Exception reporting and propagation mechanisms are responsible for raising the exception and searching for the proper handler. In object oriented languages, an exception is thrown and a stack unwinding mechanism is used to propagate the exception to its hander. The exception can be handled anywhere along the propagation path. Stack unwinding is backtracking of the original execution flow. There can be
many different executions flows that lead to the same exception being thrown. As a result, once an exception or exception handling policy has changed, the programmer has to examine all the programs along every exception propagation path to find where to make the changes.

The above discussion points to two problems of existing EHM. First, the lack of scoping to constraint the amount of programs the programmer must consider when designing, debugging and changing exception handling programs. Second, exception handling programs are often scattered along the many execution paths leading to where the exception is thrown.

In addition, when the stack is unwound, the data on the stack between the exception raise point and the handler are destroyed. The lost of data often makes it difficult for the programmer to debug. The stack unwinding procedure is a departure of the normal execution flow. It is often difficult for the programmer to master.

2.4 Exception Return Models

After an exception is handled, the control flow can be transferred to the end of the try-catch block, the beginning of the block or the instruction immediately after the raising point. The three different control flows are called termination, retry and resumption return models. An EHM can provide support for one or more of these return models. Java supports only the termination model; C++ provides termination and retry while SmallTalk [3] supports all three of them.

It is the nature of the application that determines which returning model should apply. Not providing a particular return model to match the application requirement often results in awkward code. Figure 2.1 gives the code segment for an application that requires resumption. Figure 2.2 shows the code segment that using termination model. The original normal processing codes are now completely
intertwined with the exception handling code. [11] also shows that, in general, using the termination model to simulate retry model will require modification of the normal processing code to change the control flow.

```java
try{
    getResource(FILE_1);
    getResource(FILE_2);
} resume (ResourceException ex){
    // handling code
}
```

Figure 2.1. Resumption Model

```java
try{
    getResource(FILE_1);
} catch (ResourceException ex){
    // handling code
}
try{
    getResource(FILE_2);
} catch (ResourceException ex){
    // handling code
}
```

Figure 2.2. Simulate Resumption Model with Termination model

2.5 Objectives

Existing EHMs incur these problems: (1) Type-checking is beneficial but it is tedious to handle if it is universally applied; (2) Entanglement dramatically reduced the reusability and portability of programs; (3) Lack of scoping and stacking unwinding scatters exception handling code and make them difficult to locate and debug; (4) Inadequate support in exception return models produce awkward code.

The FLE EHM is developed to address these problems. Our key contribution is to enable separation of concern. Normal processing code is untangled from exception handling code. In addition, the programmer can separate different exception handling features such as overload control and recovery into different reusable modules.
A FLE program unit consists of a condition part and a program body part. The program body gets executed when its corresponding condition part becomes true. FLE is event driven: the evaluation of program unit condition parts is triggered by events, as the primary input of many embedded system applications are random and short-lived events such as in telecommunication systems and sensor networks.

A FLE feature contains a set of program units that perform the functionality of a feature. A feature is developed according to a model, which defines the condition space and the basic functionality of the application. The condition space is specified in a domain statement. Figure 3.1 shows a domain statement for a telephony system. The basic functionality is specified in a feature called an anchor feature. A model is built with a domain statement and an anchor feature. Features designed according to an anchor feature can be considered as an extension or enhancement of the anchor feature.

```plaintext
domain BasicTelephony {
variables:
DTEnum State {DIALING, OUTPULSING,
    BUSY, AUDIBLE,TALKING,RINGING,
    DISCONNECT,IDLE};
    State state= State.IDLE; //initial value
events:
    TerminationRequest;
    Busy;
    Ringing;
    Answer;
    Disconnect;
    Onhook;
    Offhook;
    Digits;
}
```

Figure 3.1. The Domain Statement for Call Processing
Features are integrated in a feature package without requiring modification. Features and feature packages are reusable. The programmer may package different combinations of them, or he may change the way the integration works in different feature packages to meet different user needs. For example, he may choose to use different Retry features on platforms equipped with different redundancy.

We are going to demonstrate FLE by examples on telephony system throughout this thesis. Each phone object in our telephony system is associated with two feature packages: one for digit collection and analysis (allowing for features like speed calling), and the other for call processing (allowing for features like call forwarding). Phone objects can have different sets of features in their feature packages.

The domain statement of the call processing feature package declares and initializes the domain variable state and a set of events that will be used in the condition part of the program units. A domain variable is of a domain data type which must contain public predicate methods and/or Boolean members. It is extended from a Java class with the addition of a combination function, needed to support a fast first order SAT algorithm that will be described later. State is extended from the Java enum class with values like IDLE, RINGING, TALKING and so on. In the digit analysis feature packages, we use condition variables extended from Java Integer which is unbounded. FLE is not limited to defining finite state machines.

The anchor feature POTS is given in Figure 3.2 showing only two of its program units: MakeCall applies dial tone when the user picks up the phone; ReceiveCall responds to a TerminationRequest event by updating the state of the call to RINGING and telling the calling party of that fact.

The condition part of a program unit is composed of a condition statement and an event statement. The condition statement is a first order formula of domain
variables and their predicate methods. FLE does not explicitly support the existential and universal quantifiers. When the programmer has the need to say something like “there exists some elements”, we ask him to write a predicate method non-empty () instead. The event statement specifies a list of events. Each event may be attached with a qualification which is a first order formula on data carried in the event.

The feature DoNotDisturb is shown in Figure 3.3. Its program unit Say-Busy returns a Busy event whenever the phone receives a TerminationRequest event.

It can be shown that if the conjunction of the condition parts of two program units is satisfiable, the two program units interact. When the satisfiable condition, called the interaction condition, becomes true, either program units may get executed. The programmer is required to remove, or resolve, the ambiguity. When a feature is compiled, interaction among its program units will have been resolved. It can also
be shown that two features interact if some of their program units interact. FLE provides a tool that detects the interaction condition.

Figure 3.4 shows the code of the feature package QuietPhone integrating the features POTS and DoNotDisturb. The two features interact in all their program units triggered by the TerminationRequest message. The interaction is resolved by the priorityPrecedence statement with the following semantics: when an interaction condition becomes true, only the program unit belonging to the feature with the highest precedence in the list will get executed.

When the phone that uses QuietPhone receives the TerminationRequest message, only the program unit SayBusy of DoNotDisturb will be executed. But when the phone receives an OffHook event and the phone is IDLE, then the MakeCall program unit of POTS gets invoked and the user can make phone calls.
This simple example shows that the two interacting features can be integrated together without changing each other’s code. The feature resolution facilities provided by FLE are quite general. Besides using precedence lists, the programmer can use program units to resolve interaction for any specific condition. More complex examples are given in [14].
FLE exception handling mechanism is outstanding in two aspects: (1) Exception handling code is reusable; (2) Changes in exception handling policy do not require modification in normal processing code. In this section, we will discuss in detail about FLE EHM.

4.1 FLE Exceptions

4.1.1 Exception Definition. Exceptions in FLE-EHM are special events called exception events. In FLE, exceptions are declared as objects in the same way as Java does [6]. Java exception objects can be directly used in the feature language extension. The data members of the class are used to carry information from the raise point to the handler. The information usually includes but not limited to, cause of the exception, stack trace during the exception occurrence, etc.

By accepting JAVA exception hierarchy [1], FLE exceptions can also be extended, grouped or categorized. In a telephony system, a phone can throw AreaCodeException which is the descendant of InvalidPhoneNumberException. A handler that catches the super class exception (InvalidPhoneNumberException) can handle its descendant exception classes (such as AreaCodeException). Of course, programmer can specify a handler for AreaCodeException particularly. The exception hierarchy is useful to organize exceptions and offers a more flexible exception handling style.

4.1.2 Domain Specific Exception. FLE has the notion of domain specific exceptions. In the domain statement of a FLE program, programmers can declare exceptions to be domain specific exceptions. These exceptions are the ones associated with the domain. They typically are essential recoverable exceptions of current
domain. Figure 4.1 shows an enhanced **BasicTelephony** domain statement with common exception events that will be needed regardless of the execution environment.

```plaintext
domain BasicTelephonyWithExc {
    variables:
        // Same as in Figure 3.1
    events:
        // Same as in Figure 3.1
    exceptions:
        RingCktBrokeException;
        ConfCktBrokeException;
        // and others
    resources:
        // Same as in Figure 3.1
}
```

Figure 4.1. BasicTelephony enhanced with exception declarations

FLE EHM has the following specifications for domain specific exceptions: (1) Domain specific exceptions are checked exceptions. Details about the checked exception policy of FLE EHM are discussed in next section. (2) Domain specific exceptions can only be thrown from a FLE program unit or thrown from a Java object defined as *resources*. This implies domain specific exceptions are normally user defined exceptions particularly for current domain. If a domain is extended from an existing domain, all the domain specific exceptions will be inherited.

### 4.1.3 Checked Exception

FLE EHM supports both checked and unchecked exceptions. One of the most significant advantages of checked exception is that it can prevent programmers’ mistake of leaving some essential recoverable exceptions unhandled. Note that the policy of treating checked exceptions in FLE EHM is different from that in Java EHM. The latter forces the programmer to either handle the checked exception or throw it to its caller program explicitly. Otherwise, the compilation will fail. Taking separation of concern as our main design objective,
FLE EHM does not require programmer to do anything about the checked exception while writing normal processing code. Programmers can compile their code and test it without worrying the exceptions temporarily. However, the compiler does provide warnings for the unhandled checked exceptions, which should be taken care of later. Thus, we can remind programmers to handle the essential exceptions while giving them the flexibility to focus on one task at a time.

4.2 Exception Handler in FLE EHM

The exception handlers in FLE EHM are exception program units. The syntax of an exception program unit is shown in Figure 4.2.

```
exception_prog_name1 {
  context: {
    condition:  condition;
    event:     trigger;
  }
  exception: trigger_exception;
  {
    // exception handing code
  }
}
exception_prog_name2 {
  context: {feature.prog_name}
  exception: trigger_exception;
  {
    // exception handing code
  }
}
```

Figure 4.2. Exception Feature Syntax

Similar to normal program units, exception program units are composed of the context part, the exception part and exception program unit body. The exception program unit will be invoked to handle the exceptions listed in the exception part that are thrown from the program units that matches the designated context. FLE EHM provides support to two types of syntax for the context. The programmer can
use a *condition-event* pair as the context to create a general handler or put down a particular program unit name to make a specific exception handler. Since the general handler is applied to more than one program units, no local variable can be referenced there. However, the local variable reference is supported in the specific exception handler in FLE.

Exception program unit can only exist in exception features. However, exception features are also allowed to contain normal program units mentioned in section 3. The purpose of this is that, because under certain conditions some events are considered to be unexpected or unwanted, the program unit that corresponding to these situations should be included in exception features for better separation of concern. Like normal features, exception features can be put in a feature package. If multiple exception features in a feature package interact, programmer can use the precedence list to solve the conflict.

We show the reusable exception features *CatchAll* and *DamageControl* in Figure 4.3 and Figure 4.4 respectively. They are combined with the feature package *SelectiveForwarding* into a *RobustSelectiveForwarding* (RSF) feature package shown in Figure 4.5 *CatchAll* is placed at the bottom of the precedence list in RSF.

The short hands, such as *all* in context field, *any* in event and exception field, are provided for the convenience of programmers. If the execution of RSF leads to a condition not specified or unanticipated by other features, diagnostic message will be printed by the program units in *CatchAll*. If the execution leads to an illegal condition specified by *IllegalOnhook*, the program unit in *DamageControl* will be invoked to prevent further damage. Both exception features are reusable; they can be includes in other feature packages to perform the same function. They are difficult to implement in procedural languages, such as Java can C++ where the programmer must specify execution flows. For example, to implement *CatchAll*, the programmer
exception feature CatchAll {
    domain: BasicTelephonyWithExc;
    anchor: Pots;
    catch_AllExceptions {
        context: {all};
        exception: any;
        {
            System.out.println("CatchAll: Exception Caught");
            This.dump (domain, event);
        }
    }
    catch_AllEvent {
        condition: all;
        event: any;
        {
            System.out.println("CatchAll: Unexpected Condition and Event");
            This.dump (domain, event);
        }
    }
}

Figure 4.3. The CatchAll exception feature

have to repeat the same handler in every functions.

4.3 Exceptions Reporting and Propagation

4.3.1 Throwing Exceptions. Exceptions in FLE can be raised by two means. (1) They can be explicitly thrown from a program unit by throw statement; (2) If a program unit calls a method which throws exceptions to its caller, then the exceptions will be passed on to the calling program unit. Figure 4.6 show two program units in DoNotDisturb feature that throw AreaCodeException and PhoneBookException respectively.

4.3.2 Exception Propagation. Exception propagation mechanism is responsible for delivering exceptions to correct handlers. Exception handlers in FLE EHM are
exception feature DamageControl {
  domain: BasicTelephonyWithExc;
  anchor: Pots;
  IllegalOnhook {
    condition: state.equalsTo(State.IDLE);
    event: Onhook;
    {
      System.out.println("Illegal Onhook");
      fone.disable();
    }
  }
  BrokenRingCKT {
    context: {all};
    exception: RingCktBrokeException;
    {
      System.out.println("Ring CKT broken");
    }
  }
  // Other program units not shown
}

Figure 4.4. The DamageControl exception feature

feature package RobustSelectiveForwarding {
  domain: BasicTelephonyWithExc;
  features: SelectiveForwarding, DamageControl, CatchAll;
  priorityPrecedence: DamageControl, SelectiveForwarding, CatchAll;
}

Figure 4.5. The RobustSelectiveForwarding feature package

exception program units. An exception is delivered to the correct exception program unit by the event propagation mechanism of the FLE platform. Since FLE is an event driven language, it has a propagation mechanism to propagate all its events including the exception events. The exceptions are of higher priority than other events by default. The propagation mechanism uses the exception type and the context part associated with the exception program unit to decide where to deliver the exception.

Once an exception is thrown out of a program unit in a feature package or
feature DoNotDisturb {
    domain: BasicTelephony;
    anchor: POTS;
    SayBusy {
        condition: state.equalsTo(State.IDLE);
        event: TerminationRequest evnt; {
            Busy b = new Busy();
            if(evnt.fromPhoneId != 312 ){
                throw new AreaCodeException();
            } 
            else
                router.sendEvent (evnt.fromPhoneId, b);
        }
    }
    OnHold{
        condition:state.equalsTo(State.BUSY);
        event: TerminationRequest evnt;{
            Hold h = new Hold();
            // method phoneBook.getHoldMsg() throws
            // PhoneBookException to its caller
            msg_id = phoneBook.getHoldMsg( evnt.fromPhoneId );
            h.applyMsg(msg_id);
            router.sendEvent(evnt.fromPhoneId, h)
        }
    }
}

Figure 4.6. Throwing Exceptions

any enclosed feature, the exception program unit that has the matching context and
exception type is invoked to handle it. If no such a handler is found in any of the
exception features that are included in current feature package, the exception will be
propagated outside of the feature package.

When there is more than one matching handler, the propagation mechanism
uses the precedence list provided by the developer to solve the interaction. There
are two kinds of precedence list in FLE: priority precedence and straight precedence.
In the former case, only the first matching handler will be executed, while in the
latter, all the matching handlers will be executed in the given order of priority. For
more complex scenarios, mixed precedence is used. Mixed precedence will be able to address all the required precedence combinations [20]. In the mixed precedence list showed in Figure 4.7, the exception event will be delivered to the first matching feature among feature1 and feature2. Then it will also be delivered to feature3 and feature4. But feature3 and/or feature4 will process the exception only if there is a match.

```
straightPrecedence ( priorityPecedence(feature1, feature2),
                    feature3,feature4);
```

Figure 4.7. Mixed Precedence

Exception propagation mechanism in FLE reduces the complexity of stack unwinding that troubles programmers in modern EHMs. When interaction is aroused among exception features, precedence list is used to provide various solutions elegantly.

4.4 Exception Handling Return Models

There are three basic return models for exception handling: termination, retrying and resumption. These models are, indeed, requirement of applications instead of programming style. Lack of complete support directly results in ugly code (ref. section 2). Therefore, FLE EHM has provisions for all of them. Let’s look at them one at a time.

4.4.1 Termination Model. In this model, control flow transfers from the point where exception is raised to the handler, terminating intervene blocks. After the handler does its job and exit, the control flow continues as if the incomplete operation in the guarded block terminated without encountering the exception.

Our previous examples demonstrate the termination model. When an exception in the functional program unit occurs, the execution of the rest of the program
unit is terminated; the control flow is transferred to the appropriate handler, a matching exception program unit in exception feature, where the exception is handled. After that, the FLE application will process the next event in queue. All FLE exception program units are of this model by default.

4.4.2 Retry model. Different from the termination model, retry model defines a restart point for the operation. Once the handler returns, control flow is transferred to the restart point, allowing the failed operation to be executed for another time. The restart point is often set to the beginning of the guarded block and there is hardly any other better choice.

FLE EHM implements the retry model by restoring the context, the condition-event pair, of the program unit where the exception is raised. Since the execution of program units in FLE is invoked by the matching context solely, restoring the context results in the particular program unit to be executed again. This implies that each program unit in FLE is regarded as a guarded block where operation can be restarted.

Figure 4.8 shows a sample exception feature that adopts the retry model. Retry is the keyword. Recall that the program unit MakeCall in POTS (ref. Figure 3.2) will be invoked when the state is idle and the arriving event is Termination-Request. Consequently, after the handling process of the retryHandler unit, the state will be reset to idle and an identical TerminationRequest event as last one will be generated and sent to POTS to trigger MakeCall again.

4.4.3 Resumption model. In the resumption model, control flow transfers from the raise point to a handler to process the exception and then continue from the very next instruction after the raise point.

The keyword resume indicates that an exception program unit uses the resumption model. Suppose an exception is thrown from program unit A which is
associated with a resumption handler. Same as using other handling model, the corresponding exception program unit will be invoked immediately to correct the operation. Then the next instruction, if any, in program unit A just after the raise point will get executed. If there is no instruction left in program unit A, the event will continue propagating to the next matching program unit if necessary. And if program unit A is the last program unit in the event propagation path, the next event in queue will be processed. In practice, developer may decide to log the information and continue execution on some exceptions. One of such exception handlers in FLE EHM is illustrated in Figure 4.9.

As we mentioned previously, the three basic return models are indispensable to graceful programming. Simulating one another often results in awkward code. FLE EHM supports all these models to assist the programmer to survive in all kinds of
situations.
CHAPTER 5

RELATED WORK

In this section, we’ll introduce the history of EHM briefly and discuss the related work of modern EHM in architecture level, operation system level and the language level respectively.

Exception handling has been investigated for several decades. In 1975, Goodenough [12] proposed to add explicit programming language constructs for exception handling. In 1977, combination of recovery blocks and exceptions are introduced in by Melliar-Smith and Randell [16]. When programming stepped into the Object-oriented realm, Dony [7] shows how the object-oriented formalism can improve the expressive power of an exception handling system and how it can simplify its implementation and its utilization. Recently, separation of concerns, the separation of functional code and exception handling code, becomes one of the major issues addressed by researchers. For example, [18] proposes to combine exception handling and reflection to increase this division. Aspect Oriented Programming (AOP) is studied to promote both the separation of concern and reusability in exception handling [15] [10].

Several approaches have been proposed to improve exception handling from different aspects. In architectural level, Robert Miller and Anand Tripathi [17] present an abstraction called guardian for exception handling in distributed and concurrent systems that use coordinated exception handling. [2] introduces several strategies to write quality exception handling code.

Operating system can also provide facilities for exception handling [13]. Unix Signal Mechanism is one of such examples. When an exception occur, a signal (interrupt) is generated, which preempts execution and calls a handler routine, suspending prior execution; when the handler routine returns, prior execution continues (some
systems choose to terminate). Programmer isn’t required to test a variable (such as `errno`) to see if a signal has occurred, instead the programmer has to write a handler for the signal which tells the kernel *if and when the signal occurs, do the following.* This EHM model alleviates testing for the occurrence of rare conditions throughout the program, and from explicitly changing the control flow of the program. Besides, it provides a mechanism to prevent an incomplete operation from continuing. However, the extensibility is quite limited, as most signals are predefined and unavailable to programmers.

Among many of the aspects in which exception handling is investigated, researches on language support EHM is by far the most influential ones. We’ll compare the related work with FLE EHM in this field from the following perspective: exception definition, separation of exception handling code and normal processing code, and the exception return model.

In OOP, exceptions are defined as objects. The class hierarchy offers the programmer a flexible programming style. Additionally, Java introduced the idea of checked exceptions. On one hand, it increases the robustness of software by preventing the manifestation of unanticipated checked exceptions at run-time. On the other hand, they decrease the adaptability of software because they must be propagated explicitly, and must often be handled even if they cannot be signaled. [22] introduces in a new language construct, anchored exception declarations, intended to solve this issue. FLE EHM treats checked exception differently. Compilation warnings, instead of errors, are provided on unhandled essential exceptions to remain the robustness of the software while offering more flexibility and better separation of concern to the programmers.

Java and C++, the most popular OOP, use *try-catch* pattern to separate the exception handling code and normal processing code. However, it only alleviates
the problem by making line-by-line entanglement into block-by-block entanglement. Recently, aspect oriented languages are actively studies to reduce the amount of code related to exception handling. [15] [10] both give quantitative results on handling exceptions using AspectJ in different scenarios. They show that, in general, AOP can promote the reusability and separation of concern between normal processing programs and exception handling programs. However, it requires careful design and planning. Otherwise, the behavior of the programs may unintentionally alter when they are extracted to aspects. Moreover, it is, in many cases, necessary to refactor the existing application code to expose the joint points that AspectJ can capture. This results in impact in the overall cohesion of the system. Our approach can completely separates the functional code and the exception handling code with regular feature and exception feature. All the work related to exception handling can be completed in exception features without any refactoring process.

Surprisingly, SmallTalk [3] is the only that supports three basic exception return models. C++ [19] supports retry and termination model and Java [6] only provides support to termination model. Inadequate support to exception return models directly results in awkward programs and often aggravates the code entanglement. FLE EHM supports all three basic models to adapt various scenarios.
In this investigation, we introduced several novel mechanisms for exception handling for the FLE programming language extensions.

We follow the existing object oriented programming aproach of defining exceptions as objects but provides compile time warnings on unhandled checked exceptions so that the programmer can develop normal processing programs separated from exception handling programs.

We provide language constructs to enable development of exception handling code and normal programming code separately in exception features and regular features, thus, enhancing the reusability and portability of both.

The propagation of exceptions follows the same design paradigm of FLE. The context dependent propagation frees the programmer from the complexity brought in by stack-unwinding, and, hence, easier to understand.

We support all three basic exception return models, termination, retry and resume, in the language level to help the programmer to develop programs to meet different application requirements.

We consider this to be our main contribution: FLE EHM separates normal processing code and exception handling code into separate reusable features, as a result, changes due to exception handling policy are isolated, making other software that interacts with exception handling more portable.

A prototype of FLE EHM is implemented and available for downloading on the website (www.openflx.org). It has been applied to the development of a telephony system implemented in FLE.
BIBLIOGRAPHY


