Extended Exceptions for Contingencies

Thorsten van Ellen
BTC AG
Business Unit Software Solutions
26121 Oldenburg, Germany
thorsten.van.ellen@btc-ag.com

Wilhelm Hasselbring
University of Kiel
Software Engineering Group
24118 Kiel, Germany
wha@informatik.uni-kiel.de

ABSTRACT

We observed a general problem of sequential programs, which often results in design and programming errors in industrial software engineering projects, and propose a solution approach. Telephone lines may be busy, banking accounts may be overdrawn and disks may be full. These things happen in the real world, causing the disruption and non-fulfillment of an expected service. Ignoring these problems leads to violations of the postconditions of the caller that depends on the service. The conditions are exactly known and cannot always be avoided, but measures could be taken afterwards. A good program should handle them as part of the specification. As such they are not specification violations and should not be regarded as errors. Unfortunately, they usually can or shall not be handled immediately within the direct caller, e.g., for information hiding reasons. The problem is similar to the problem of error code handling and handling them with exception mechanisms seems reasonable, but the problem is even more complex. These situations must not terminate the system suddenly, because that also violates postconditions. Consequently, exceptions for these situations must be distinguished from exceptions for errors and are worth handling separately. Therefore, we introduce the new concept *contingency* for such situations. Since the conditions are defined, they are candidates for forward recovery, but conventional exception mechanisms are not appropriate for that purpose. Appropriate mechanisms are presented in this paper. A systematic inspection and handling of contingencies with these mechanisms before runtime can diagnose and avoid subsets of specification violations effectively. An evaluation approach will be presented.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures, Procedures, functions, and subroutines*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SERENE 2008, November 17-19, 2008, Newcastle, UK.
Copyright 2008 ACM 978-1-60558-275-7/08/11...$5.00

General Terms

Reliability, Languages

Keywords

contingency, exception, forward recovery, resumption, reliability

1. INTRODUCTION

Telephone lines may be busy, banking accounts may be overdrawn and disks may be full. These things happen in the real world and cause the disruption and non-fulfillment of an expected service. Ignoring them leads to violations of the postconditions of the caller, since the caller depends on the service. You know exactly what happened if you are familiar with that implementation level. The conditions are exactly known and cannot always be avoided, but measures could be taken afterwards. A good program should handle such special situations as part of its normal code, allowing for them as part of the specification. As such they are not a violation of the specification and should not be regarded as errors.

A common recommendation is not to communicate such as exceptions, since exceptions should only be used for specification violations, i.e., if the pre- or postcondition is violated [9]. Instead, it is recommended to declare them explicitly within the interface [5, 3, 13] and redeclare them within the interface of the caller if they have not been handled yet [5, 3]. It is not only recommended to redeclare them, but also to adjust their abstraction to the current interface abstraction [17]. Furthermore, when using exceptions, the termination model for handling exceptions is preferred [8, 3, 16] over the challenged resumption model [8, 3, 15]. All these recommendations are challenged in this work for contingencies.

Section 2 explains some terms and introduces the term contingency for these situations. Section 3 discusses contingencies extensively and shows that they are very difficult to handle and handling them within conventional programming languages with exceptions is reasonable. Section 4 presents the objectives and added value of this work. Section 5 clarifies important properties of contingencies. Section 6 pinpoints some significant deficiencies of conventional exception handling mechanisms for handling contingencies. Accordingly, Section 7 proposes extended exception mechanisms. Section 8 discusses work and mechanisms related to the solution approach proposed here. Finally, Section 9 describes further work.

2. TERMS

This section describes some essential terms.

Often, errors are defined as states, e.g., by [1], for example, if a traffic light shows red and green at the same time, but such a definition does not include erroneous state *transitions*, e.g., if a traffic light changes directly from green to red where no erroneous state is involved. Therefore, the term situation is used instead of state here and is defined as follows:

Definition 1. **Situation**: A situation is a sequence of n states with $n > 0$.

This definition includes states and transitions. A situation can be a sequence consisting of a single state and for the sake of simplicity situations can be illustrated as states.

Definition 2. **Specification**: A specification is a complete, consistent description of all situations that are allowed for a system.

Example 1. A specification of a function can be composed of a pre- and postcondition. The precondition for a traffic light transition function describes the valid states:

$$pre : state = green \vee state = yellow \vee state = red,$$

the postcondition describes the valid transitions, starting from the valid states:

$$\begin{aligned} post : (state = green \wedge state' = yellow) \\ \vee (state = yellow \wedge state' = red) \\ \vee (state = red \wedge state' = green). \end{aligned}$$

The length of the regarded sequences of states (situations) is arbitrarily determined by the specification, e.g., the length of some regarded situations within example 1 is 2. Only traffic light state transitions that conform to the described situations in that specification are allowed, all other state transitions are not allowed.

A specification is not only a mathematical set of states or situations, because it has a structure, conditions and names. A specification of a non-trivial system, e.g., with the specification languages B or Z, is usually modular, similar to program code. A specification usually contains modules or functions building upon or calling each other. A specification contradicts itself and is not consistent if it allows a situation as a normal result within a postcondition of a callee module, e.g., result `LineBusy` of module `sendFax` within example 3, and forbids the same simultaneous situation at another calling module, e.g., by preconditions of the following operation.

Definition 3. **Error**: An error is a situation, the conditions of which contradict the specification.

An error or specification violation occurs, for example, if the situation does not comply with the preconditions or postconditions, as [9] states, e.g., if a traffic light changes directly from green to red.

The new term contingency (coined by [13]) is defined (differing from [13]) as follows:

Definition 4. **Contingency**: A contingency is a situation that is described within the specification of a module, and represents a module result where the task or function, which calling modules depend on, was not performed.

Usually contingencies are directly perceived as if they were errors, e.g., violations of preconditions [9], but that is not a correct perception.

Example 2. A method `allocateMemory` might expect a parameter `requiredAmount` of type `integer` that should be in the range of positive values. If the invoker passes negative values, it violates the precondition. Such a situation is an error, but if the invoker passes positive values and the condition `availableMemory < requiredAmount` evaluates to true, the method returns `OutOfMemory` as specified and both the invoker and the method are not doing anything wrong, nevertheless both cannot perform their usual work. `OutOfMemory` can be formally defined, often cannot be avoided and should be specified. Such a situation is a contingency.

Example 3. If a fax should be sent via a modem controlled by software, the telephone line can be busy (`LineBusy`) or the number might be gone (`SIP_ERROR_410_NumberGone`). Usually, both are not avoidable, not even by changing the program code of the software. Regardless of how sophisticated the software is, at some implementation level the contingency may occur. Nobody can buy and control the whole telephone network and there is always the risk that the unique end device that should be reached is not available. Even if there is a possibility to guarantee a solution technically, it might be too expensive for a small application, e.g., a dedicated line and end device. The same is true for parallel access to database tables, files or other unique entities or devices.

Contingencies are results of modules indicating that the module could or should not fulfill its usual work. Such situations are not errors or specification violations, because they are unavoidable or intentional behaviors and therefore can be found within the specification. Contingencies are exactly described within the specification and therefore are specification compliant.

At the time the contingency is detected no damage occurred, yet, the system still behaves as specified. Only if the contingency is ignored and not handled specifically, will damage inevitably occur. From the perspective of the dependant module, contingencies are work refusals whose potential appearance is known in advance, independently of the abstraction level they represent. If no specific additional measures are taken, the expectations of dependant modules that do not regard the possible work refusal will not be fulfilled and they cannot reach their postconditions. As a result, a specification violation (error) will appear. As will be shown in the discussion in Section 3, there is no practicable solution to regard every contingency at every call.

Contingencies differ from normal situations in that normal situations do not represent work refusals and without additional specific measures do not necessarily run into specification violations (errors).

3. DISCUSSION

This section discusses language constructs to communicate and handle contingencies. You might think that there are only few such contingencies, but quite the contrary is true if you are looking for them.

Example 4. A simple routine like `openFile` might have up to 20 or more of them: `Drive/Dir/FileNotFound/Locked/NameInvalid, DiskNotInDrive,`

`DiskNotFormatted, DiskFull, EndOfFile, NoAvailableFileHandles, NetworkDisconnected` etc.

Usually, they are far from being systematically and completely documented or even declared within the interface, but nevertheless they are always present. Such situations are omnipresent. Their number is finite as is the implementation, but Oracle database routines and other complex systems might return thousands of them (see Section 5).

Unfortunately, mostly contingencies can or shall not be handled immediately as easily as other normal specified situations within the direct caller. The reason is that not all required information is available or not all required components for measures are accessible, because the required elements should be kept encapsulated in other call levels. For example, the graphical user interface usually is not available within lower levels to inform or ask the user. Even if immediate handling would be possible, full disks and other contingencies may occur at several places and their solution should not be implemented redundantly at several places. If the contingency is not solved immediately, it must be communicated to the dependant caller by one means or another.

3.1 Avoiding Exceptions?

How should these contingencies be communicated to the caller? Communicate contingencies to the caller not as exceptions as recommended by [9] and communicate them as special values declared in the interface or its documentation? That is reasonable if they do not disclose implementation details, but declare about 20 or more contingencies in the interface of the caller of `openFile` because the direct caller cannot handle it immediately? And additionally, do it in the interface of the caller of the caller and so on along the whole caller chain until they are handled? The contingencies of `openFile` are not the only contingencies that might occur in the call graph of the call chain. All unhandled contingencies would occur and accumulate within all interfaces of all callers of the call graph until they are handled. This accumulation of unhandled contingencies is unavoidable. At the top level routine, e.g., `main`, all unhandled contingencies of the whole system accumulate. Really manually declaring them all explicitly might be incredible much work.

Disclose all these implementation details within the interfaces on all levels? If one day, someone decides to change the implementation and the corresponding contingencies, change all intermediary interfaces? This leads to an intolerable maintainability problem.

Solve the maintainability problem by abstracting the implementation details of contingencies as usually recommended, e.g., abstracting the implementation details `FileLocked` and `TableLocked` of two different implementations by `ResourceLocked`? This might be even more work. If one day, someone decides to handle some of these contingencies at another level, still again change all intermediary interfaces? Additionally, how should `DiskFull` and other contingencies of `openFile` and other implementation details be abstracted to conform to the abstraction of the routine `commitTransaction`?

Furthermore, the primary argument against abstracting the implementation details is that indispensable implementation information gets lost along with options for forward recovery, i.e., a specific handling, repairing and finally resumption after a successful repairing. The valuable implementation information should never get lost [12]. The different contingencies cannot be distinguished and handled separately and specifically anymore if they are abstracted. Therefore, implementation specific measures cannot be taken, e.g., measures for `FileLocked` probably operate on the operating system and measures for `TableLocked` operate on the database.

Declare an abstract return value class or exception class on the interface for several contingencies and embed more distinguishable information at runtime (nested or derived objects or exceptions) like it is usually done in Java? Then you never know before runtime what really can happen. If a method may throw an `IOException`, is it a `DiskFull`, `FileReadOnly` or what else? The information is not available within the interface at development time, but at runtime, when it is too late to develop a handler. The complete information of the interface must be available at development time before runtime!

3.2 Worse than Error Code Handling?

This problem is similar to the problem of error code handling that has been found impracticable and has been replaced by exception handling. Exceptions surely are a part of the solution, but the problem is even worse than error code handling. Nobody is amazed if an error terminates the system, but everybody is annoyed if the line is busy or the disk is full and additionally the system crashes.

If an error already occurred, usually it is reasonable to terminate the system with an exception, but if the system is still in defined circumstances of the specification, sudden termination is a violation of the specification and one of the errors that should be avoided and therefore not a good option. Forward recovery is the best option. At least in these cases, all corresponding exceptions must be distinguished and handled separately from exceptions representing errors.

3.3 Problem Statement

The discussed ideas cannot be a satisfying solution. There seems to be no single approach regarding all of our discussed arguments, especially at least one of the arguments about accumulation, implementation dependence and abstraction and their consequences is overlooked.

[5] describes a concept with declaration of exceptions within the interface and compile time checking similar to Java's checked exceptions disregarding accumulation and implementation dependence. Java has checked exceptions that also disregard accumulation, but usually will be solved by abstraction. [9] only allows the use of exceptions for errors, all other situations must be declared within the interface. [17] recommends to abstract. Nobody seems to recognize the whole combination of problems or even accidentally solves them. We have never seen real code that consequently solved things like that. No one seems to do it reliably. Until now, it is not possible to determine all contingencies of all implementation levels at development time with conventional languages. How can they ever be handled reliably completely? They cannot at the first run! Usually, they simply crash at the user.

We assume, there is no practical and general approach for contingencies besides exceptions so far, caused by the described accumulation and implementation dependence. This might be one main reason for much of the confusion about the nature of exceptions, because they are not only indispensable for errors but also for contingencies.

In our perception there exist no satisfying solution to declare/document contingencies at development time, communicate them at runtime and handle them without specification violation so far. They are not even recognized separately from specification violations and normal situations. Nevertheless they are omnipresent and are worth to be regarded and handled separately. Therefore, we try to establish the new term contingency within Section 2 to separate them from errors and normal situations.

Separating contingencies, surprisingly opens a new perspective providing new understandings and a plethora of innovative topics that makes this paper a bit crowded, which is also reflected in the objectives in Section 4. Once you take this perspective, some commonly made observations can be explained, for example, the confusion about the nature of exceptions and the seeming misuse, teething problems etc.

We feel like contingencies are a systematic source of errors and might encompass much more than teething problems, e.g., diverse inconsistencies, missing requirements and potentially catastrophes. Contingencies are a general and systematic problem of sequential programs that needs to be solved. We believe, this problem can be solved better and many errors can be avoided systematically with appropriate language mechanisms. Vice versa, as long as this problem is not solved, this source of errors will continue to exist.

The space of errors is divided with the term contingency to make the situations that are treatable, accessible for forward recovery. Conventional exception handling is not able to make this distinction and therefore appropriate handling and resumption is not possible.

We intend to contribute to the discourse of errors, exceptions, handling and resumption.

4. OBJECTIVES AND ADDED VALUE

Contingencies must be handled in the sense of forward recovery if following specification violations should be avoided. The objectives of our work and this paper are to

- distinguish contingencies from normal situations and errors.
- enable a simple communication of errors and contingencies between the call levels.
- determine contingencies automatically at development time and always get a current and complete documentation of all available contingencies of all levels. (This is solved by alternative mechanisms, instead of declaring them all manually within the interface.)
- determine the source of some potential errors directly (not via symptoms) and before runtime, thereby reducing time and effort to analyse errors and determine the sources.
- enable forward recovery. This should generally be possible, i.e., rescue situations without database transaction mechanisms, e.g., within cross-system interfaces, and even for side effects like physical processes.
- keep information hiding as far as possible.
- avoid complications of conventional interfaces, cumbersome cleanups and partial repetitions.
- enable overriding any handling by outer context with broader knowledge and component access.
- reduce redundant code, the total amount of code and code complexity needed for communicating, handling, repairing and resuming.
- show deficiencies of conventional exception handling for these objectives.

Only sequential, imperative programs should be regarded, even though we assume our approach might also be reasonable for non-sequential programs and our thoughts might be translated to event based systems and other programming paradigms.

5. PROPERTIES OF CONTINGENCIES

The following section presents significant properties of contingencies and describes why handling of contingencies within conventional programming languages with exception handling is reasonable.

5.1 Contingencies are Omnipresent

As shown in the discussion, contingencies can be very numerous alone within one single function. They are littered over many functions within the whole system and all levels.

The following example from the database domain illustrates how numerous contingencies actually are within everyday life. Presumably, contingencies are not less numerous within operating systems or other complex environments like enterprise resource planning (ERP) systems, e.g., SAP, only less documented and apparent. This shows how important it is to handle these situations explicitly.

Example 5. Oracle maintains a documentation [10] of all problem messages of the Oracle database. It encompasses over 2000 pages, each with several messages. Hence, it documents several thousands of entries and mostly contains detailed and specific (not abstracted) and therefore helpful hints for each single known situation that can occur at runtime. Not all of these entries describe situations where the database is within an unknown or undefined state (error) that must be repaired and prevents the database to operate correctly. Instead, substantial amounts of them are numerous contingencies that are recognized, intercepted and communicated at runtime successfully and documented (quasi specified) at development time of the caller. However, not all of them can occur if the calling program is implemented correctly, e.g., the message ORA-01747 ("invalid column") cannot occur if all data definition language (DDL) and data manipulation language (DML) statements are consistent.

5.2 Contingencies are Unavoidable and Better Treatable than Errors

Function results that represent contingencies (refusals to work) usually will be avoided intuitively at system development time if possible. The contingencies that remain are unavoidable, but on the other hand their circumstances are exactly known and their conditions are defined. Conditions of errors contradict the specification and are not exactly described within the specification. Handling contingencies is therefore easier than handling errors.

5.3 Contingencies Disclose Implementation Details and Must Not Be Abstracted

When passed to the callers in the caller hierarchy, contingencies disclose implementation details, not immediately, but usually after a few call levels when a class or group of classes representing the same abstraction is left.

If different contingencies are mapped to one abstract contingency, e.g., `OutOfMemory` and `DiskFull` to `OutOfResource`, the different conditions of the different contingencies cannot be distinguished anymore and specific handling is no longer possible.

Example 6. A specific handling for `OutOfMemory` like swapping is neither applicable for `DiskFull` nor for the abstraction of both `OutOfResource`.

Our unconventional recommendation is *not* to abstract contingencies for information hiding reasons, because it would map the contingency of the current implementation and the potential contingencies of future implementations to one abstracted situation for which a current handler might not be appropriate. The implementation details might be indispensable [12] for repairing at runtime. Since it is mandatory to handle contingencies specifically to avoid subsequent specification violations, it must be assured that they remain unambiguous and will *not* be abstracted.

6. DEFICIENCIES OF CONVENTIONAL EXCEPTION MECHANISMS

This section outlines deficiencies of conventional exception mechanisms to easily handle contingencies successfully and continue execution afterwards.

Ascertaining contingencies before runtime.

To handle contingencies, they must be ascertainable at development time. Conventional languages have no practicable mechanisms to determine all exceptions that can occur syntactically within a code fragment at development time, although that should be not a complex problem.

Distinguishing contingencies and errors.

Unlike errors, contingencies are not allowed to terminate and therefore must be distinguished from errors. Conventional languages have no exception mechanism to distinguish them.

Repairing lower level implementation details.

Repair measures can be taken in defined circumstances, but they require access to the implementation details of the lower call levels from the higher call levels by one means or another. Conventional languages have no language mechanisms to do so. In the following Java example 7 the lower level method call of `saveEditedData` does not exist anymore after throwing an exception, but even if the stack would not be unwound, no language construct exist to easily manipulate the variable `currentPath`.

Example 7. The routine `main` with access to the graphical user interface executes several tasks and handles all occurrences of `DiskFull` by asking the user for an alternative path, but is not able to access and change the variable `currentPath` of the lower level:

```
void main(String[] args) { // GUI-access here
  try {
    doTasks(args);
  } catch (DiskFull) {
    String alterPath =
      AskUserForAlternative.execute().getAnswer();
    // repair lower level with alterPath, but how?
  }
}

// many call levels lower:
void saveEditedData(Data edited) { // No GUI here
  if (getFree(currentPath) < edited.size()) {
    throw new DiskFull(getFree(currentPath),
                       edited.size());
  }
  // ... writing data on disk
}
```

Resuming.

After a successful handling, the program should continue execution to fulfill the postconditions, but how? Conventional exception handling does not offer an easy mechanism to continue execution. The termination model is not helpful for continuing by any means, because the stack is always unwounded and then important issues must be solved to continue within defined conditions: abandon some work already done, undo or handle internal and external side effects even irreversible physical effects, determine where and how to continue, redo some work and loose performance.

Language mechanisms for resumption are the only known, generally and easily applicable option to bypass the latter hurdles.

Overriding handlings.

No known language offer a mechanism to override handlers similar to polymorphy to exploit additional context knowledge, components and access available in higher levels. Therefore, the Java example 8 cannot override the handling of the exception `PaperJam` within the method `print` by the handling in the method `printAdvanced` and access its additional hardware component `advancedPaperEmitter`.

Example 8. The routine `print` might be the implementation of a simple printer model, catches the exception `PaperJamDetected` and handles it conventionally. The routine `printAdvnc` might be the implementation of a sophisticated printer model with an additional paper emitter hardware component that reuses the routine `print` and tries to override the handling of `PaperJamDetected`, but that is not possible.

```
public void print(Object document, int fromPage) {
  try {
    // print ...
  } catch (PaperJamDetected jamDetected) {
    // Default handling: log and cancel
    logger.error("Paper jam: aborting print.");
  }
}

public void printAdvnc(Object doc, int fromPage) {
  try {
```

```
    print(document, fromPage);
  } catch (PaperJamDetected jamDetected) {
    // overriding for automatic repair impossible!
    advancedPaperEmitter.removeJam();
    printAdvanced(document, jamDetected.atPage());
  }
}
```

7. AN APPROACH FOR EXTENDED EXCEPTION MECHANISMS

A common recommendation is to use exceptions only for errors [9]. This is nearly impossible in non-trivial cases caused by contingencies as we have shown in the discussion section. Therefore, we unconventionally recommend to use exceptions also for contingencies.

Since conventional exception handling is deficient for forward recovery of contingencies and resumption, we sketch extended exception mechanisms that offer appropriate handling and resumption. Mainly concepts of the language Common Lisp are sketched for the language Java. They should also be transferable to other languages.

After giving an overview, the elements of our approach are presented in more detail. Afterwards, an example illustrating the extended mechanisms follows.

7.1 Origins and Overview

Significant ideas are coined by the language Common Lisp [11].

One important concept of Common Lisp (and our approach) is to refine the conventional twofold separation of problem detection (`throw`) and problem handling (`catch`) into a threefold separation of problem detection, problem handling and optional problem solution. The problem solution resumes the execution at any implementation level that is appropriate for the repair and will be called from the handling by name. Common Lisp calls them restart, we call them offer. The problem handling is also called decision. The resumption model of our approach is exactly carried over from Common Lisp, especially multiple, named offers including parameters on multiple, arbitrary levels.

The only important difference is the reversal of the search direction for decisions and offers that is explained in more detail in Section 7.2. Additionally to the concepts of Common Lisp, our approach distinguishes errors from contingencies by different keywords, ascertains all contingencies at development time with the help of the development environment and let the developer interactively choose the contingency to handle and insert the corresponding code into the program.

7.2 Elements of the Approach

All deficiencies of conventional exception handling that have been mentioned within Section 6 are solved by the following elements of the approach.

Distinguishing contingencies.

A new keyword is introduced into the language to distinguish contingencies from errors. Contingencies are marked and thrown with the new keyword `signal` and distinguished from errors that are thrown with the known keyword `throw`. For backward compatibility, the conventional behavior of `throw` should not be changed, the stack will be unwound by `throw` and resuming is not possible anymore, but `signal` should behave differently. It should not unwind the stack immediately. Selecting the appropriate keyword determines whether the exception is resumable or not.

As a result, all contingencies occurring syntactically within a code fragment, including all called levels, can be distinguished from errors by keyword and before runtime.

Ascertaining contingencies automatically before runtime and choosing interactively.

It should be possible to handle arbitrary contingencies at arbitrary places, but determining all possible contingencies is very difficult with conventional mechanisms. The need to ascertain all exceptions at development time should not be satisfied by declaring all of them within the interface explicitly and manually. Therefore, we suggest an alternative tool supported approach by automatically ascertaining all of them at development time and presenting them within a dialog. Due to the expected huge amount of contingencies (see Section 5) only an interactive choice of the contingencies that should be handled seems reasonable. This requires a cooperation of all corresponding parser or compiler respectively and development environments. The dialog should offer a sorting and filtering by diverse criteria, e.g., by type, class hierarchy, location, call chain, frequency, already registered occurrence, the existence of a decision for the contingency, personal settings, name pattern matching etc. After the developer has chosen the contingency that should be handled, appropriate code is inserted into the program.

Overriding by reversal of search direction.

All conventional languages and also Common Lisp search for handlers (decisions) and restarts (offers) from the lowest levels to the highest bottom-up. In contrast to the conventional search direction, our approach searches in reverse direction top-down. By reversing the search direction, it becomes possible to override existing handlers by higher levels as it was intended by the example 8. It is a dynamic binding similar to object oriented polymorphism, but it does not search along a class inheritance path, instead it searches along the call stack.

Distinguish handling with and without possibility of resumption.

If an exception is thrown and the stack is unwound, it can be catched with `catch`, but it cannot be resumed anymore. The behavior of `catch` should not be changed for backward compatibility, because existing handlers assume that the stack is already unwound, resources have already been cleaned etc. These existing correct assumptions should not be violated. Therefore, a resumable exception must not occur at existing `catch` handlers. Hence, a new keyword is required to distinguish and handle exceptions that are resumable. The new keyword `decide` should be used to handle signaled and resumable contingencies. If multiple call levels of the call chain exist that can handle the contingency, the decision that is the topmost in the call chain is executed due to the reversed search direction. In this way, any decision can be overridden quasi polymorphically along the call chain.

Repairing lower level implementation details by offers for resumption.

In different situations different measures are needed to repair the implementation details of the same level. Therefore, multiple different possibilities for resumption of contingencies can be defined at every arbitrary level with separate name and parameters. These resumption possibilities are introduced with the new keyword `offer`, which is followed by a name and formal parameter declarations with usual notation. These offers are side entrances into the interrupted methods that are still on the stack. They are like procedures (without result value), because they should resume and not return. Offers can access all implementation details of that implementation level. They are additional interfaces and can keep the information hiding principle as conventional interfaces. Offers can only be called by a decision of a contingency.

Resuming.

The new keyword `resume` is used to call offers from decisions, followed by the name and the required actual parameters with usual notation. If multiple offers with the same name and parameters on different call levels exist, the topmost offer is chosen due to the reversed search direction. For this reason, the resumption is not always fixed to the level where the contingency is signaled originally. In this way, offers can be overridden quasi polymorphically along the call chain. When the offer and its level is chosen, the stack will be unwound (lately) till there.

Repairing by cooperation of levels.

The approach presented here enables the cooperation of the involved levels, e.g., to use the context knowledge (`advancedPaperEmitter` of the decision level of example 8 or the user interface of the example 7) and also to repair the implementation (`currentPath` of example 7) on the offer level without violating its information hiding.

7.3 Illustrating Example

In the following example, that was coined by [14] and slightly extended, the concepts of Common Lisp are sketched for the language Java. The example shows all syntactic extended mechanisms in conjunction. It will be presented as source code within example 9.

Within the example, multiple log-files should be read. Within the log-files are multiple lines that should be checked for whether they are well-formed. For this purpose, two nested loops are used. The first loop iterates over the files. The second loop iterates over the entries of one file. Both loops are implemented within two separated methods `analyseLogs` and `parseFile`, of which the first calls the second. For each line of the files a third method `parseEntry` will be called that checks whether the line is well-formed. If not, it signals the contingency `MalformedLine` by the new keyword `signal`. Both loops offer an option to resume the abstraction of which corresponds to the according implementation level, i.e., the loop within `analyseLogs` offers `skipFile` and the loop within `parseFile` offers `skipEntry`, each without parameters. The offers are introduced by the new keyword `offer`. Furthermore, both loops decide what should happen in the case that a line is not well-formed (`MalformedLine`). For the choice of the contingency that should be handled, the new keyword `decide` is used. Within the handler (decision) of the chosen contingency the choice of the repair and resumption offer is done with the new keyword `resume`. Within the method `parseFile` the repair offer `returnEntry` is called and the element `defaultEntry` is passed as parameter, which only exists there. This way both involved levels can cooperate using their specific implementation details. Two possibilities exist within the example on two different call levels to decide or handle the contingency `MalformedLine`. The higher level method overrides the decision of the lower level method, therefore the decision with resumption of `skipEntry` is chosen.

Example 9.

```
void analyzeLogs(Files openFiles) {
  for (File file: openFiles) {
    try {
      use(parseFile(file));
    } offer skipFile() {
      continue; // nothing else to do
    } decide (MalformedLine x) {
      if (x.firstStackFrame().startsWith("mylib"))
        resume skipEntry();
    }
  }
}

Entries parseFile(File openFile) {
  Entry defaultEntry = new Entry();
  Entries result;
  while (!openFile.EOF()) {
    Entry entry = null;
    try {
      String logTxt = openFile.line();
      entry = parseEntry(logTxt);
    } offer skipEntry() {
      // entry = null;
    } decide (MalformedLine) {
      // entry = null;
      resume returnEntry(defaultEntry);
    }
    if (entry != null)
      result.add(entry);
  }
  return result;
}

Entry parseEntry(String logTxt) {
  if (entryIsWellFormed(logTxt)) {
    return new Entry(logTxt);
  } else {
    signal new MalformedLine(logTxt);
  }
}
```

8. RELATED WORK

This section covers an account of prior related work, languages and mechanisms, explaining why this has not solved the problem. Related languages and mechanisms are summarized within Table 1. The rows of the table list the features of the presented approach. The columns show which features are supported by the related languages and mechanisms.

Features	Common Lisp	Smalltalk	Eiffel	Callbacks [6]	(Closures)	(Continuations)	AOP
Distinction of contingencies	-	-	-	-	-	-	-
Ascertaining contingencies before runtime	-	-	-	-	-	-	-
Interactive choice	-	-	-	-	-	-	-
Reverse conventional search direction	-	-	-	-	-	-	-
Access to decision level after recognition	x	x	x	-	(x)	(x)	-
Resumption	x	x	-	x	(x)	(x)	-
Resumption at arbitrary level	x	-	-	-	-	-	-
Multiple offers at the same level	x	-	-	-	-	-	-
Repair of offer level	x	x	x	x	(x)	-	-
Parametrizing of offer/repair	x	-	-	x	(x)	-	-

Table 1: Comparison of alternative languages and mechanisms (x = possible, (...) = probably, but not proved, - = not possible)

8.1 Distinction of Errors and Contingencies

The idea for the english term contingency was coined by [13], but he defines contingencies partially different. He defines that contingencies are "expected conditions" (could be called specified) "demanding an alternative response" (but need not be work refusals) that "can be expressed in terms of the method's intended purpose" (or abstraction). Consequently contingencies that are only expressed in terms of the lower levels method's intended purpose are no contingencies anymore at higher levels. Therefore, our contingencies of lower levels that disclose implementation details are Ruzek's faults, "that cannot be described without reference to the method's internal implementation" [13].

Although [5] recommends to use exceptions not only for domain or range failures or violations of pre- or postconditions respectively but also for other kinds of situations and to support resumption, he does not distinguish contingencies from other situations.

[9] outlines situations like Overflow, OutOfMemory, special I/O cases incompletely as hardware or operating system signals or as abnormal conditions. He speaks of implicit preconditions that are impractical or impossible to formulate. This is not correct. Often they are easy to formulate and must not be excluded by preconditions because these situations often are unavoidable and the system usually should not behave arbitrarily, e.g., when `FileLocked` or `LineBusy` occurs. The system should handle them in a defined way. Therefore, they should be handled separately as part of the postconditions of the higher levels. Probably it is no accident that Meyer mentions the need to distinguish different exceptions exactly in those cases and examples where the task or function is not fulfilled, circumstances are defined and specific measures are reasonable (contingencies).

[3] divides the specification of a currently considered program level into a standard service specification and an exceptional service specification. The exceptional service specification does only contain the behavior of the lower level exceptions if they have been redeclared for the considered level. He divides the set of all input states of a program into 4 different domains determined by the relation of the program result and its correct prediction by the specification. Our contingencies fall into all domains. The unanticipated input domain is also called specification failure and implicitly includes some of our contingencies and his lower level exceptional service specifications. Since we do not recommend abstraction and manual redeclaration of those contingencies we are unlucky with the term specification failure for those.

[17] distinguishes conditions (as used in the Common Lisp community) and errors. He recommends to use Common Lisp's conditions for "unusual outcomes" and defines them as follows: "The contract of a function specifies, among other things, the possible "outcomes" of calling the function. There is always one "usual" or "straight-line" kind of outcome, and then there can be zero or more "unusual" outcomes" ([17]). He does not define them as work refusals, but work refusals might be part of unusual outcomes. He recommends to always adjust the abstraction of unusual outcomes to the abstraction of each caller interface. We do not recommend this, since it makes a specific handling and resumption afterwards impossible.

8.2 Resumption Model, Threefold Separation and Dynamic Binding

Often, safety critical systems must not terminate after an error occurred, but usually termination and failing fast is recommended. The preference for the termination model [7, 4] is reasonable for errors, but if the goal is forward recovery, there is no better alternative to the resumption model. The termination model is not sufficient as explained within Section 3.2. Callbacks or closures are recommended as substitute [7, 16], but they do not allow access to the implementation details of the signaling level and the higher level at the same time the contingency occurred, thereby obstructing cooperation of the levels, repairing and reasonable resumption.

Eiffel [9] offers a retry semantics [2], but no resumption model as proposed here.

Smalltalk offers termination, retry, resumption and notification. Resumption always immediately starts behind the signaling point. The exception determines on a case-by-case basis whether resumption is allowed. The signaler can influence this by choosing an appropriate exception. The handler can determine whether the exception is resumable and decide whether the execution should be resumed. In the case of resumption the signaling statement may even return a value that was delivered by the handler, thereby passing data from the handler to the signaler, but it is not possible to offer multiple resumption possibilities with different names and multiple, named parameters on different call or implementation levels.

Common Lisp offers termination, resumption and notifi-

cation. Common Lisp is the first known approach that practices a threefold separation (see Section 7.1). The resumption model proposed here was directly carried over from Common Lisp. Our offers correspond to restarts and our decisions correspond to handlers. An equivalent mechanism for notification is not intended in our work. The main difference of our approach and Common Lisp are some additions named within Section 7.1 and the reversal of the search direction to enable overriding handlers and restarts (decisions and offers), and thereby a kind of polymorphy. Within all mechanisms known before, the reaction of the system is prompted from the lower levels to the higher levels even in Common Lisp [14] and by [5]. As soon as a reaction takes place, all remaining upper levels no longer have a chance to react anymore even if they hold some reactions that can judge the application context better.

[6] also offers a threefold separation and multiple resumption possibilities with different parameters within Java but only within the signaling level without names by overloading, thereby preventing multiple resumption possibilities with equal parameters. Resumption is internally implemented with callback mechanisms that can not access the implementation details of the handler level exactly at the time the contingency occurred.

Continuations may enable access to the implementation details of the higher levels and resumption afterwards, but lack other features.

AOP join points for throw statements are unknown within all known AOP frameworks, therefore it cannot be avoided that the stack is unwound, hence, resumption is not possible.

8.3 Summary: Difference from Prior Work

So the differences of this work from prior work is that prior work does not distinguish and regard contingencies especially of lower levels, their implementation dependence and relevance, their accumulation and do not solve the problems arising from them, e.g., specification violations. We distinguish contingencies from errors, and thereby situations that can be handled within defined conditions from situations within undefined conditions.

We recognize and solve the problems arising from declaring contingencies explicitly within the interface, e.g., by checked exceptions. We provide explanations for the confusion about exceptions that are no errors.

We recommend forward recovery and resumption for contingencies and to enable specific handling do *not* recommend abstraction of contingencies. We provide mechanisms to support forward recovery by the developer at development time including mechanisms to override existing error handlings.

9. FUTURE WORK

This section describes the next steps in our research. The main steps are an empirical research and a proposal for the Java standard.

Formalization and implementation.

To clearly state what contingencies are, a set based formalization will be developed similar to [3]. An initial implementation of the extended exception mechanisms for Java will be build. This includes the extension of the syntax, compiler and possibly the JVM. It will be build by using the open source implementation of Sun. The implementation of the extended mechanisms itself will be open source. Comrades are welcomed.

Evaluation approach: empirical research.

Empirical research about the benefits of the proposed approach will be performed. Controlled experiments as well as industrial research might be possible. For the arrangement and analysis of the empirical research the GQM approach (Goal, Question, Metric) will be used, i.e., develop objectives, questions, corresponding hypothesis, metrics, measurements and evaluations in this order. Possible objectives are mentioned in Section 4.

Controlled experiments with conventional and extended mechanisms can be applied within a student software practical. It must be ensured that students with knowledge of the extended mechanisms must not influence students without that knowledge, e.g., by performing the corresponding experiments successively. A briefing should provide an understanding of contingencies.

In the first step, the students will be provided with the task to develop a solution for a problem including solutions for as many contingencies as possible without prior knowledge of concrete contingencies and without extended mechanisms. Additionally, a task will be assigned to adjust the implementation, e.g., from a file based variant to a database variant. We will explicitly point out that the students have to fulfill this special task exactly three times under different conditions and should try to optimize this task. Automatic tests checking the expected postconditions, but not provoking contingencies will be provided with source code. The results of the first step will be archived for the third step.

In the second step, automatic tests provoking contingencies for the first implementation, e.g., file based variant, will be provided without source code. The task of the students will be to solve the failing tests without extended mechanisms. Afterwards the implementation should be migrated the second time. Automatic tests provoking contingencies for the second implementation variant will be provided without source code and must be solved again. Additionally, a task to override existing handlers will be provided.

In the third step, the students will be provided with extended exception mechanisms and the task to repeat the second step on the basis of the archived results of the first step.

The solutions will be measured regarding the objectives. Criteria can be

- the number of failing tests (errors) found and provoked by contingencies or the number of contingencies solved without prior knowledge of them in the different steps,
- the time needed to solve contingencies or the number of contingencies solved in a limited time frame,
- the amount of code of the implementations or their complexity,
- the amount of implementation details occurring within the interfaces,
- the effort needed to adjust the implementation,
- the effort needed to analyse errors and to find the causing contingencies,

- the number of builds and iterations to solve contingencies,
- the effective solution of contingencies without runtime tests,
- answers of interviews about the subjective experience and opinion of the students about the extended exception mechanisms.

Proposal for the Java standard.

Additionally, the solution approach is proposed for the Java standard. It will be submitted by a Java Specification Request (JSR) and be implemented corresponding to the process defined by Sun, i.e., by constituting an expert group and developing a draft. Comrades for the expert group are welcomed. This might additionally require an extension of the Java specification, Java standard library and a test suite for the JSR.

10. CONCLUSIONS

Contingencies are undesired results of routines but specified as well as the desired results within the specification. In this light, contingencies are the situations that really can be handled and where forward recovery and resumption is reasonable. Hence, contingencies may unclose the hopes that initially have been associated with error and exception handling.

[8] is often cited as argument against the resumption model and has the opinion that a more complex mechanism can be justified only if the additional expressive power it provides is frequently needed. As we have shown, resumption is needed frequently for forward recovery of contingencies. Some may fear the additional complexity of resumption, but we do not see any other appropriate solution. No alternative approaches than resumption to solve contingencies are known. Additionally, our impression is that the presented concepts are simple and less complex than other already established concepts from other domains, e.g., event oriented systems.

A systematic inspection and handling of contingencies with these mechanisms before runtime can diagnose and avoid subsets of specification violations effectively. Furthermore, such errors are not detected by symptoms, but on the basis of their sources. Therefore, partially extensive analyses to draw conclusions from the symptoms to the sources are not necessary.

We assume that the systematic handling of contingencies has positive effects for the production readiness in the first iteration. Potentially fewer teething problems occur in first product versions, but even if the presented mechanisms are helpful, they will not solve thousands of contingencies within one single system immediately. Contingencies will still lead to numerous errors further on. We hope that over time increasingly more frameworks, libraries, programs and systems find reasonable algorithms for forward recovery and solve their contingencies.

Especially for safety critical applications the proposed extended exception mechanisms might be very helpful immediately.

11. REFERENCES

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

[2] P. A. Buhr and W. Y. R. Mok. Advanced exception handling mechanisms. *Software Engineering*, 26(9):820–836, 2000.

[3] F. Cristian. Exception handling and tolerance of software faults. In M. R. Lyu, editor, *Software Fault Tolerance*, pages 81–107. John Wiley & sons, 1995.

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

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

[6] A. Gruler and C. Heinlein. Exception handling with resumption: Design and implementation in Java. In *PLC*, pages 165–171, 2005.

[7] B. Liskov. A history of CLU. *SIGPLAN Not.*, 28(3):133–147, 1993.

[8] B. H. Liskov and A. Snyder. Exception handling in CLU. *IEEE Trans. Softw. Eng.*, 5(6):546–558, 1979.

[9] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[10] Oracle. Oracle9 i database error messages, release 2 (9.2) part no. a96525-01. 2002. `http://download.oracle.com/docs/cd/B10501_01/server.920/a96525.pdf`.

[11] K. M. Pitman. Exceptional situations in Lisp. In *Proceedings for the First European Conference on the Practical Application of Lisp (EUROPAL'90)*, Cambridge, UK, 1990.

[12] M. Raento. What should exceptions look like? *Mika Raento's Blog*, July 2006. `http://www.errorhandling.org/wordpress/?page_id=100`.

[13] B. Ruzek. Effective java exceptions. *dev2dev.bea.com*, January 2007. `http://www.oracle.com/technology/pub/articles/dev2arch/2006/11/effective-exceptions.html`.

[14] P. Seibel. *Practical Common Lisp*. Apress, September 2004. PDF at `http://www.apress.com/resource/freeebook/9781590592397` and HTML at `http://gigamonkeys.com/book/`.

[15] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Longman, April 1994. ISBN 0201543303.

[16] B. Stroustrup. Bjarne stroustrup's C++ style and technique FAQ. May 2008. `http://www.research.att.com/~bs/bs_faq2.html`.

[17] D. Weinreb. What conditions (exceptions) are really about. *Dan Weinreb's Weblog*, March 2008. `http://danweinreb.org/blog/what-conditions-exceptions-are-really-about`.