# Discovering Faults in Idiom-Based Exception Handling

Magiel Bruntink
Centrum voor Wiskunde en Informatica
P.O. Box 94079
1090 GB Amsterdam, The Netherlands
Magiel.Bruntink@cwi.nl

Arie van Deursen
Software Evolution Research Laboratory
EEMCS
Delft University
Mekelweg 4, 2628 CD Delft, The Netherlands
Arie.van.Deursen@cwi.nl

Tom Tourwé
Centrum voor Wiskunde en Informatica
P.O. Box 94079
1090 GB Amsterdam, The Netherlands
Tom.Tourwe@cwi.nl

## ABSTRACT

In this paper, we analyse the exception handling mechanism of a state-of-the-art industrial embedded software system. Like many systems implemented in classic programming languages, our subject system uses the popular return-code idiom for dealing with exceptions. Our goal is to evaluate the fault-proneness of this idiom, and we therefore present a characterisation of the idiom, a fault model accompanied by an analysis tool, and empirical data. Our findings show that the idiom is indeed fault prone, but that a simple solution can lead to significant improvements.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*

## General Terms

Design, Reliability, Verification

## Keywords

Exception handling, fault model, program analysis

## 1. INTRODUCTION

A key component of any reliable software system is its exception handling. This allows the system to detect errors, and react to them correspondingly, for example by recovering the error or by signalling an appropriate error message. As such, exception handling is not an optional add-on, but a sine qua non: a system without proper exception handling is likely to crash continuously, which renders it useless for practical purposes.

Despite its importance, several studies have shown that exception handling is often the least well understood, documented and tested part of a system. For example, [30] states that more than 50% of all system failures in a telephone switching application are due to faults in exception handling algorithms, and [20] explains that the Ariane 5 launch vehicle was lost due to an unhandled exception.

Various explanations for this phenomenon have been given.

First of all, since exception handling is not the primary *concern* to be implemented, it does not receive as much attention in requirements, design and testing. [26] explains that exception handling design degrades (in part) because less attention is paid to it, while [9] explains that testing is often most thorough for the ordinary application functionality, and least thorough for the exception handling functionality. Granted, exception handling behaviour is hard to test, as the root causes that invoke the exception handling mechanism are often difficult to generate, and a combinatorial explosion of test cases is to be expected. Moreover, it is very hard to prepare a system for all possible errors that might occur at runtime. The environment in which the system will run is often unpredictable, and errors may thus occur for which a system did not prepare.

Second, exception handling functionality is crosscutting in the meanest sense of the word. [21] shows that even the simplest exception handling strategy takes up 11% of an application's implementation, that it is scattered over many different files and functions and that it is tangled with the application's main functionality. This has a severe impact on understandability and maintainability of the code in general and the exception handling code in particular, and makes it hard to ensure correctness and consistency of the latter code.

Last, older programming languages, such as C or Cobol, that do not explicitly support exception handling, are still widely used to develop new software systems, or to maintain existing ones. Such explicit support makes exception handling design easier, by providing language constructs and accompanying static compiler checks. In the absence of such support, systems typically resort to systematic coding idioms for implementing exception handling, as advocated by the well-known *return code* technique, used in many C programs and operating systems. As shown in [4], such idioms are not scalable and compromise correctness.

In this paper, we focus on the exception handling mechanism of a 15 year-old, real-time embedded system, developed by ASML, a Dutch company. The system consists of approximately 10 million lines of C code, and is developed and maintained using a state-of-the-art development process. It applies (a variant of) the return code idiom consistently throughout the implementation. The central question we seek to address is the following: "how can we reduce the number of implementation faults related to exception handling implemented by means of the return code idiom?". In order to answer this general question, a number of more specific questions needs to be answered.

1. What kinds of faults can occur? Answering this question requires an in-depth analysis of the return code idiom, and a fault model that covers the possible faults to which the idiom can lead;

2. Which of these faults do actually occur in the code? A fault model only predicts which faults can occur, but does not say which faults actually occur in the code. By carefully analysing the subject system (automatically) an estimate of the probability of a particular fault can be given;

3. What are the primary causes of these faults? The fault model explains *when* a fault occurs, but does not explicitly state *why* it occurs. Because we need to analyse the source code in detail for detecting faults, we can also study the causes of these faults, as we will see;

4. Can we eliminate these causes, and if so, how? Once we know why these faults occur and how often, we can come up with alternative solutions for implementing exception handling that help developers in avoiding such faults. An alternative solution is only a first step, (automated) migration can then follow.

We believe that answers to these questions are of interest to a broader audience than the original developers of our subject system. Any software system that is developed in a language without exception handling support will suffer the same problems, and guidelines for avoiding such problems are more than welcome. In this paper we offer experience, an analysis approach, tool support, empirical data, and alternative solutions to such projects.

## 2. RELATED WORK

**Fault (Bug) Finding** Recently a lot of techniques and tools have been developed that aim at either static fault finding or program verification. However, we are not aware of fault finding approaches specifically targeting exception handling faults.

Fault finding and program verification have different goals. On the one hand, fault finding's main concern is finding as many (potentially) harmful faults as possible. Therefore fault finding techniques usually sacrifice formal soundness in order to gain performance and thus the ability to analyse larger systems. Specifically, Metal [12], PREfix [7], and ESC [14] follow this approach. We were inspired by many of the ideas underlying Metal for the implementation of our tool (see Section 5).

Model checking is also used as a basis for fault finding techniques. CMC [23] is a model checker that does not require the construction of a separate model for the system to be checked. Instead, the implementation (code) itself is checked directly, allowing for effective fault finding.

On the other hand, program verification is focused on proving specified properties of a system. For instance, MOPS [8] is capable of proving the absence of certain security vulnerabilities. More general approaches are SLAM [2] and ESP [10]. While ESP is burdened by the formal soundness requirement, it has nevertheless been used to analyse programs of up to 140 KLOC.

**Idiom Checkers** A number of general-purpose tools have been developed that can find basic coding errors [17, 24]. These tools are however incapable of verifying domain-specific coding idioms, such as the return code idiom. More advanced tools [31, 29] are restricted to detecting higher-level design flaws but are not applicable at the implementation level.

In [4], we present an idiom checker for the parameter checking idiom, also in use at ASML. This idiom, although much simpler, resembles the exception handling idiom, and the verifier is based on similar techniques as presented in this paper.

**Exception Handling** Several proposals exist for extending the C language with an exception handling mechanism. [19, 25] and [32] all define exception handling macro's that mimic a Java/C++ exception-handling mechanism. Although slightly varying in syntax and semantics, these proposals are all based around an idiom using the C `setjmp/longjmp` facility.

Exceptional C [16] is a more drastic, and as such more powerful, extension of C with exception handling constructs as present in modern programming languages.

All these proposals differ from our proposal (Section 7) in that our proposal still uses the return-code idiom, but makes it more robust by hiding (some of) the implementation details. This makes migration of the old mechanism to the new one easier, an important concern considering ASML's 10 MLoC code base.

Several papers describe exception handling analyses for Java-like exception handling mechanisms. Robillard and Murphy [27] show how exception structure can degrade and present a technique based on software compartmenting to counter this phenomenon. Their work differs from ours in that they reason about the application-specific design of exception handling, whereas we focus on the (implementation of) the exception handling mechanism itself. Fu *et al* present an exception-flow analysis that enables improving the test coverage of Java applications [15]. Similarly, Sinha and Harrold present a class of adequacy criteria that can be used for testing exception handling behaviour [28]. Although this work could lead to better tests, and hence well-tested exception handling code, a test-based approach remains necessarily partial. Hence, such techniques should be considered complementary to our technique. Additionally, both techniques are targeted toward Java-like exception handling mechanisms, and it is not clear how they would work for systems using the return-code idiom.

## 3. CHARACTERISING THE RETURN CODE IDIOM

The central question we seek to answer is how we can reduce the number of faults related to exception handling implemented by means of the return code idiom. To arrive at the answer, we first of all need a clear description of the workings of (the particular variant of) the return code idiom at ASML. We use an existing model for exception handling mechanisms (EHM) [18] to distinguish the different components of the idiom. This allows us to identify and focus on the most error-prone components in the next sections. Furthermore, expressing our problem in terms of this general EHM model makes it easier to apply our results to other systems using similar approaches.

### 3.1 Terminology

An exception at ASML is "any abnormal situation found by the equipment that hampers or could hamper the production". Exceptions are logged in an *event log*, that provides information on the machine history to different stakeholders (such as service engineers, quality assurance department, etc).

The EHM itself is based on two requirements:

1. a function that detects an error should log that error in the event log, and recover it or pass it on to its caller;

2. a function that receives an error from a called function must provide useful context information (if possible) by *linking* an error to the received error, and recover the error or pass it on to the calling function.

```
1  int f(int a, int* b) {
2     int r = OK;
3     bool allocated = FALSE;
4     r = mem_alloc(10, (int *)b);
5     allocated = (r == OK);
6     if((r == OK) && ((a < 0) || (a > 10))) {
7        r = PARAM_ERROR;
8        LOG(r,OK);
9     }
10    if(r == OK) {
11       r = g(a);
12       if(r != OK) {
13          LOG(LINKED_ERROR,r);
14          r = LINKED_ERROR;
15       }
16    }
17    if(r == OK)
18       r = h(b);
19    if((r != OK) && allocated)
20       mem_free(b);
21    return r;
22 }
```

**Figure 1: Exception handling idiom at ASML.**

An error that is detected by a function is called a *root* error, while an error that is linked to an error received from a function is called a *linked* error.

If correctly implemented, the EHM produces a chain of related consecutive errors in the event log. This chain is commonly referred to as the *error link tree*, and resembles a stack trace as output by the Java virtual machine, for example.

Because ASML uses the C programming language, and C does not have explicit support for exception handling, each function in the ASML source code follows the *return code* idiom. Figure 1 shows an example of such a function. We will now discuss this approach in more detail.

## 3.2 Exception Representation

An exception representation *defines what an exception is and how it is represented*. At ASML, a *singular* representation is used, in the form of an *error variable* of type `int`. Line 2 in Figure 1 shows a typical example of such an error variable, that is initialised to the `OK` constant. This variable is used throughout the function to hold an *error value*, i.e., either `OK` or any other constant to signal an error. The variable can be assigned a constant, as in lines 7 and 14, or can be assigned the result of a function call, as in lines 4, 11 and 18. If the function does not recover from an error itself, the value of the error should be propagated through the caller by the `return` statement (line 21).

## 3.3 Exception Raising

Exception raising is *the notification of an exception occurrence*. Different mechanisms exist, of which ASML uses the *explicit control-flow transfer* variant: if a root error is encountered, the error variable is assigned a constant (see lines $6 - 9$), the function logs the error, stops executing its normal behaviour, and notifies its caller of the error.

Logging occurs by means of the `LOG` function (line 8), where the first argument is the new error encountered, which is linked to the second argument, that represents the previous error value. The function treats root errors as a special case of linked errors, and therefore the root error detected at line 8 is linked to the previous error value, `OK` in this case.

Explicit guards are used to skip the normal behaviour of the function, as in lines 10 and 17. These guards check if the error variable still contains the `OK` value, and if so, execute the behaviour, otherwise skip it. Note that such guards are also needed in loops containing function calls.

If the error variable contains an error value, this value propagates to the `return` statement, which notifies the callers of the function.

## 3.4 Handler Determination

Handler determination is *the process of receiving the notification, identifying the exception and determining the associated handler*. The notification of an exception occurs through the use of the `return` statement and catching the returned value in the error variable when invoking a function (lines 4, 11 and 18). This approach is referred to as *explicit stack unwinding*.

The particular exception that occurs is not identified explicitly most of the time, rather a *catch-all* handler is provided. Such handlers are mere guards, that check whether the error value is not equal to `OK`. Typically, such handlers are used to link extra context information to the encountered error (lines $12 - 15$), or to clean up allocated resources (lines $19 - 20$).

## 3.5 Resource Cleanup

Resource cleanup is *a mechanism to clean up resources, to keep the integrity, correctness and consistency of the program*.

ASML has no automatic cleanup facilities, although specific handlers typically occur at the end of a function if cleaning up of allocated resources is necessary (lines $19 - 20$).

## 3.6 Exception Interface & Reliability Checks

The exception interface *represents the part in a module interface that explicitly specifies the exceptions that might be raised by the module*. ASML developers sometimes specify (parts of) this interface in informal comments, but this is not strictly required.

Consequently, reliability checks that *test for possible faults introduced by the EHM itself* are currently not possible. The focus of this paper is to analyse which faults can be introduced and to show how they can be detected and prevented.

## 3.7 Other Components

An EHM consists of several other components than the ones mentioned above. Although these are less important for our purposes, we shortly describe them here for completeness.

**Handler scope** is *the entity to which an exception handler is attached*. At ASML, handlers have *local* scope: handlers are associated to function calls (lines $12 - 15$), where they log extra information, or can be found at the end of a function (lines $19 - 20$), where they clean up allocated resources.

**Handler binding** *attaches handlers to certain exceptions to catch their occurrences in the whole program or part of the program*. ASML uses *semi-dynamic* binding, which means that the same exception can be handled differently in different locations in the source code, by associating a different handler with each exception occurrence.

**Information passing** is defined as *transfer of information useful to the treatment of an exception from its raising context to its handler*. At ASML there is no information passing except for the integer value that is passed to a caller. Although an error log is constructed, the entries are used only for offline analysis.

**Criticality management** represents *the ability to dynamically change the priority of an exception handler, so that it an be* *changed based on the importance of the exception, or the importance of the process in which the error occurred*. This is not considered at ASML.

## 4. A FAULT MODEL FOR EXCEPTION HANDLING

Based on the characterisation presented in the previous section, we establish a fault model for exception handling by means of the return code idiom in this section. The fault model defines when a fault occurs, and includes failure scenarios which explain what happens when a fault occurs.

### 4.1 General Overview

Our fault model specifies possible faults occurring in a function's implementation of the *exception raising* and *handler determination* components. Those components are clearly the most prone to errors, because their implementation requires a lot of programming work, a good understandability of the idiom, and strict discipline. Although this also holds for the *resource cleanup* component, at ASML this component primarily deals with memory (de)allocation, and we therefore consider it to belong to a *memory handling* concern, for which a different fault model should be established.

The return code idiom at ASML relies on the fact that when an error is received, the corresponding error value should be logged and should propagate to the `return` statement. The programming language constructs that are used to implement this behaviour are function calls, return statements and log calls. The fault model includes a predicate for each of these constructs, and consists of three formulas that specify a faulty combination of these constructs. If one of the formulas is valid for the execution of a function, the EH implementation of the function necessarily contains a fault.

A function is regarded as a black box, i.e., only its input–output behaviour is considered. Any error values received from called functions (*receive* predicate) are regarded as input. Outputs are comprised of the error value that is returned by a function (*return*), and values written to the error log (*LOG*). This perspective allows easy mapping of faults to failure scenarios, at the cost of losing some details due to simplification. We map the input and outputs to logical predicates as follows.

First, *receive* is a unary predicate that is true for an error value that is received by the function during its execution. If a function receives an error `PARAM_ERROR` somewhere during its execution, then *receive*(`PARAM_ERROR`) holds true. If a function does not receive an error value (either because it does not call any functions, or no exception is raised by a called function), then *receive*(OK) holds. Second, *return* is a unary predicate that holds true for the error value returned by a function. Finally, *LOG* is a binary predicate that is true for those two error values (if any) that are written to the error log. The first position of the *LOG* predicate signifies the new error value, while the second position signifies the (old) error to which a link should be established. If and only if nothing is written to the error log during execution, *LOG*(void, void) holds.

The fault model makes two simplifications. First, it assumes a function receives and logs at most one error during its execution. This is reasonable, because if implemented correctly, no other function should be called once an error value is received. Second, if only one error value can be received, it makes little sense to link more than one other error value to it.

### 4.2 Fault Categories

The fault model consists of three categories, each including a failure scenario, which are explained next. The predicates capturing the faults in each category are displayed in Figure 2. Example code fragments corresponding to Categories 1–3 are displayed in Figures 3–5, respectively.

*Category 1.* The first category captures those faults where a function raises a new error ($y$), but fails to perform appropriate logging. There are two cases to distinguish. First, $y$ is considered a root error, i.e., no error has been received from any called function, and therefore $receive(OK)$ holds. The function is thus expected to perform $LOG(y,OK)$. However, a category 1 fault causes the function to perform $LOG(y,z)$ with $z \neq OK$.

Second, $y$ is considered a linked error, i.e., it must be linked to a previously received error $x$. So, $receive(x)$ holds with $x \neq OK$, and the function is expected to perform $LOG(y,x)$. A category 1 fault in its implementation results in the function performing $LOG(y,z)$ with $x \neq z$.

Category 1 faults have the potential to break error link trees in the error log. The first case causes an error link tree to be improperly initiated, i.e., it does not have the *OK* value at its root. The second case will break (a branch of) an existing link tree, by failing to properly link to the received error value. Furthermore, the faulty *LOG* call will start a new error link tree which has again been improperly rooted. Especially in the latter case it will be hard to recover the chain of errors that occurred, making it impossible to find the root cause of an error.

*Category 2.* Here the function properly links a new error value $y$ to the received error value $x$, but then fails to return the new error value (and instead returns $z$). The calling function will therefore be unaware of the actual exceptional condition, and could therefore have problems determining the appropriate handler. In the special case of $receive(OK)$, the function properly logs a root error $y$ by performing $LOG(y,OK)$, but subsequently returns an error $z$ different from the logged root error $y$.

Possible problems include corruption of the error log, due to linking to the erroneously returned error value $z$. Calling functions have no way of knowing the actual value to link to in the error log, because they receive a different error value. Even more seriously, calling functions have no knowledge of the actual error condition and might therefore invoke functionality that may compromise further operation of software or hardware. This problem is most apparent if *OK* is returned while an error has been detected (and logged).

*Category 3.* The last category consists of function executions that receive an error value $x$, do not link a new error value to $x$ in the log, but return an error value $y$ that is different from $x$. The failure scenario is identical to category 2.

## 5. SMELL: STATICALLY DETECTING EXCEPTION HANDLING FAULTS

Based on the fault model we developed SMELL, the *State Machine for Error Linking and Logging*, which is capable of statically detecting violations to the return code idiom in the source code, and is implemented as a CodeSurfer[1] plugin. We want to detect faults statically, instead of through testing as is usual for fault models, because early detection and prevention of faults is less costly [3, 6], and because testing exception handling is inherently difficult.

[1]www.grammatech.com

| **Category 1** | **Category 2** | **Category 3** |
|---|---|---|
| $\text{receive}(x) \quad \wedge$ | $\text{receive}(x) \quad \wedge$ | $\text{receive}(x) \quad \wedge$ |
| $\text{LOG}(y,z) \quad \wedge$ | $\text{LOG}(y,x) \quad \wedge$ | $\text{LOG}(\mathit{void},\mathit{void}) \quad \wedge$ |
| $x \neq z$ | $\text{return}(z) \quad \wedge$ | $\text{return}(y) \quad \wedge$ |
| | $y \neq z$ | $x \neq y$ |

**Figure 2: Predicates for the three fault categories.**

## 5.1 Implementation

SMELL statically analyses executions of a function in order to prove the truth of any one of the logic formulas of our fault model. The analysis is static in the sense that no assumptions are made about the inputs of a function. Inputs consist of formal or global variables, or values returned by called functions.

We represent an execution of a function by a finite path through its control-flow graph. Possibly infinite paths due to recursion or iteration statements are dealt with as follows. First, SMELL performs an intra-procedural analysis only, i.e., calls to other functions are not followed, and therefore recursion is no problem during analysis. Intra-procedural analysis only does not impact SMELL's usefulness, as it closely resembles the way ASML developers work with the code: they should not make specific assumptions about possible return values, but should instead write appropriate checks after the call. Second, loops created by iteration statements are dealt with by caching analysis results at each node of the control-flow graph. We discuss this mechanism later.

The analysis performed by SMELL is based on the evaluation of a deterministic (finite) state machine (SM) during the traversal of a path through the control-flow graph. The SM inspects the properties of each node it reaches, and then changes state accordingly. A fault is detected if the SM reaches the *reject* state; conversely, a path is free of faults if the SM reaches the *accept* state.

The error variable is a central notion in the current implementation of SMELL. An error variable, such as the `r` variable in Figure 1, is used by a programmer to keep track of previously raised errors. SMELL attempts to identify such variables automatically based on a number of properties. Unfortunately, the idiom used for exception handling does not specify a naming convention for error variables. Hence, each programmer picks his or her favourite variable name, ruling out a simple lexical identification of these variables. Instead, a variable qualifies as an error variable in case it satisfies the following properties:

- it is a local variable of type `int`,
- it is assigned only constant (integer) values or function call results,
- it is not passed to a function as an actual, unless in a log call,
- no arithmetic is performed using the variable.

Note that this characterisation does not include the fact that an error variable should be returned or that it should be logged. We deliberately do not want the error variable identification to depend on the correct use of the idiom, as this would create a paradox: in order to verify adherence to the idiom, the error variable needs to be identified, which would need strict adherence to the idiom to start with.

Most functions in the ASML source base use at most one error variable, but in case multiple are used, SMELL considers each control-flow path separately for each error variable. Functions for which no error variable can be identified are not considered for further analysis. We discuss the limitations of this approach at the end of this section.

The definition of the SM was established manually, by translating the informal rules in the manuals to appropriate states and transitions. Describing the complete SM would require too much space. Therefore we limit our description to the states defined in the SM, and show a subset of the transitions by means of example runs.

The following states are defined in the SM:

**Accept** and **Reject** represent the absence and presence of a fault on the current control-flow path, respectively.

**Entry** is the start state, i.e., the state of the SM before the evaluation of the first node. A transition from this state only occurs when an initialisation of the considered error variable is encountered.

**OK** reflects that the current value of the error variable is the OK constant. Conceptually this state represents the absence of an exceptional condition.

**Not-OK** is the converse, i.e., the error variable is known to be anything but OK, though the exact value is not known. This state can be reached when a path has taken the true branch of a guard like `if(r != OK)`.

**Unknown** is the state reached if the result of a function call is assigned to the error variable. Due to our limitation to intra-procedural analysis, we conservatively assume function call results to be unknown.

**Constant** is a parametrised state that contains the constant value assigned to the error variable. This state can be reached after the assignment of a literal constant value to the error variable.

All states also track the error value that was last written to the log file. This information is needed to detect faults in the logging of errors. Since we only perform intra-procedural analysis, we set the last logged v value to *unknown* in the case of an Unknown state (i.e. when a function was called). We thus assume that the called function adheres to the idiom, which allows us to verify each function in isolation. Faults in these called functions will still be detected when they are being checked.

While traversing paths of the control-flow graph of a function, the analysis caches results in order to prevent infinite traversals of loops and to improve efficiency by eliminating redundant computations. In particular, the state (including associated values of parameters) in which the SM reaches each node is stored. The analysis then makes sure that each node is visited at most once given a particular state. The same technique is used by Engler *et al.* in [12].

## 5.2 Example Faults

The following three examples show how the SM detects faults from each of the categories in the fault model. States reached by the SM are included in the examples as comments, and where appropriate the last logged error value is mentioned in parentheses. First, consider the code snippet in Figure 3.

```
1 int calibrate(int a) {      // Entry
2   int r = OK;               // OK
3   r = initialise();         // Unknown
4   if(a == 1)
5     LOG(RANGE_ERROR, OK);   // Reject
6   ...
7 }
```

**Figure 3: Example of fault category 1.**

A fault of category 1 possibly occurs on the path that takes the true branch of the `if` statement on line 4. If the function call at line 3 returns with an error value, say `INIT_ERROR` then receive(`INIT_ERROR`) holds. The call to the LOG function on line 5 makes LOG(`RANGE_ERROR`, `OK`) true, and since OK is different from `INIT_ERROR`, all clauses of the predicate for category 1 are true, resulting in a fault of category 1.

SMELL detects this fault as follows, starting in the Entry state on line 1. The initialisation of `r`, which has been identified as an error variable, causes a transition to the OK state on line 2. The assignment to `r` of the function call result on line 3 results in the Unknown state. On the true branch of the `if` statement on line 4, a (new) root error is raised. The cause of the fault lies here. SMELL reaches the Reject state at line 5 because if an error value (say `INIT_ERROR`) would have been returned from the call to the `initialise` function, it is required to link the `RANGE_ERROR` to the `INIT_ERROR`, instead of linking to OK.

```
1 int align() {             // Entry
2   int r = OK;             // OK
3   r = initialise();       // Unknown
4   if(r != OK)             // Not-OK
5     LOG(ALIGN_ERROR, r);  // Not-OK (ALIGN_ERROR)
6   return r;               // Reject
7 }
```

**Figure 4: Example of fault category 2.**

The function in Figure 4 exhibits a fault of category 2 on the path that takes the true branch of the `if` statement. Again, suppose receive(`INIT_ERROR`) holds, then the function correctly performs LOG(`ALIGN_ERROR`, `INIT_ERROR`). The fault consists of the function not returning `ALIGN_ERROR`, but `INIT_ERROR`, because after linking to the received error, the new error value is not assigned to the error variable.

Again SMELL starts in the Entry state, and subsequently reaches the OK state after the initialisation of the error variable `r`. The `initialise` function is called at line 3, and causes SMELL to enter the Unknown state. Taking the true branch at line 4 implies that the value of `r` must be different from `OK`, and SMELL records this by changing to the Not-OK state. At line 5 an `ALIGN_ERROR` is linked to the error value currently stored in the `r` variable. SMELL then reaches the return statement, which causes the error value to be returned that was returned from the `initialise` function call at line 3. Since the returned value differs from the logged value at this point, SMELL transits to the Reject state.

Category 3 faults are similar to category 2, but without any logging taking place. Suppose again that for the function in Figure 5 receive(`INIT_ERROR`) holds. For the path taking the true branch

```
1 int process(int a) {   // Entry
2   int r = OK;          // OK
3   r = initialise();    // Unknown
4   if(a == 2) {
5     r = PROCESS_ERROR; // Reject
6   }
7   ...
8   return r;
9 }
```

**Figure 5: Example of fault category 3.**

of the `if` statement a value different from `INIT_ERROR` will be returned, i.e., `PROCESS_ERROR`.

Until the assignment at line 5 the SM traverses through the same sequence of states as for the previous examples. However, the assignment at line 5 puts SMELL in the Reject state, because the previously received error value has been overwritten. A category 3 fault is therefore manifest.

## 5.3 Fault Reporting

SMELL reports the presence of faults using a more fine grained view of the source code than the fault model. While the fault model takes a black box perspective, i.e., regarding behaviour only at the interface level, SMELL reports detected faults using a white box perspective, i.e., considering the implementation level details of a function. The white box perspective is considered to be more useful when interpreting actual fault reports, which developers may have to process.

In the following we present a list of "low-level faults", or programmer mistakes, that SMELL reports to its users. For each programmer mistake we mention here the associated fault categories from the fault model. SMELL itself does not report these categories to the user. To help users interpreting the reported faults, SMELL prints the control-flow path leading up to the fault, and the associated state transitions of the SM.

**function does not return** occurs when a function declares and uses an error variable (i.e., assigns a value to it), but does not return its value. If present, SMELL detects this fault at the return statement of the function under consideration. This can cause category 2 or 3 faults.

**wrong error variable returned** occurs when a function declares and uses an error variable but returns another variable, or when it defines multiple error variables, but only returns one of them and does not link the others to the returned one in the appropriate way. This can cause category 2 or 3 faults.

**assigned and logged value mismatch** occurs when the error value that is returned by a function is not equal to the value last logged by that function. This can cause category 2 faults.

**not linked to previous value** occurs when a `LOG` call is used to link an error value to a previous value, but this latter value was not the one that was previously logged. If present, SMELL detects this fault at the call site of the log function. This causes category 1 faults.

**unsafe assignment** occurs when an assignment to an error variable overwrites a previously received error value, while the previous error value has not yet been logged. Clearly, if present SMELL detects this fault at the assignment that overwrites the previous error value.

## 5.4 Limitations

Our approach is both formally unsound and incomplete, which is to say that our analysis proves neither the absence nor the presence of 'true' faults. In other words, both false negatives (missed faults) or false positives (false alarms) are possible. False negatives for example occur when SMELL detects a fault on a particular control-flow path, and stops traversing that path. Consequently, faults occurring later in the path will go unnoticed. The unsoundness property and incompleteness properties do not necessarily harm the usefulness of our tool, given that the tool still allows us to detect a large number of faults that may cause much machine down-time, and that the number of false positives remains manageable. The experimental results (see Section 6) show that we are currently within acceptable margins.

SMELL also exhibits a number of other limitations:

**Meta assignments** Meta assignments are assignments involving two different error variables, such as `r = r2;`. SMELL does not know how to deal with such statements, since it traverses the control-flow paths for each error variable separately. As a result, when considering the `r` variable, SMELL does not know what the current value of `r2` is, and vice versa.

For the moment, SMELL recognises such statements and simply stops traversing the current control-flow path.

**Variableless log calls** Variableless log calls are calls to the `LOG` function that do not use an error variable as one of their actual arguments, but instead only use constants, such as in the following example:

```
r = PARAM_ERROR;
LOG(PARAM_ERROR,OK);
```

The problem appears when a function defines more than one error variable. Although a developer is able to tell which error variable is considered from the context of the call, SMELL has trouble associating the call to a specific error variable.

Whenever possible, SMELL tries to recover from such calls intelligently. In the above case, SMELL is able to infer that the log call belongs to the `r` variable, because it logs the constant that is assigned to that variable. However, the problem reappears when a second error variable is considered. When checking that variable and encountering the `LOG` call, SMELL will report an error if the error value contained in the second error variable differs from the logged value, because it does not know the `LOG` call belongs to a different error variable.

**Infeasible Paths** Infeasible paths are paths through the control-flow graph that can never occur at runtime, but that are considered as valid paths by SMELL. SMELL only considers the values for error variables, and smartly handles guards involving those variables. But it does not consider any other variables, and as such cannot infer, for example, that certain conditions using other variables are in fact mutually exclusive.

**Wrong Error Variable Identification** The heuristic SMELL uses to identify error variables is not perfect. False positives occur when integer values are used to catch return values from library functions, for example, such as `puts` or `printf`. Additionally, false negatives occur when developers pass the error variable as an actual parameter or perform some arithmetic operations on it. This is not allowed by the ASML coding standard, however. Currently, false positives are easily identified manually, since SMELL's output reports which error variable was considered. If this error variable is meaningless, inspection of the fault can safely be skipped.

In order to overcome these limitations, we are currently reimplementing SMELL, using *path-identification* and *constant propagation* algorithms involving all (local) variables of a function. This will solve the problems of variableless log calls and meta assignments, and reduce the problem of infeasible paths.

| | reported | false positives | limitations | validated |
|---|---|---|---|---|
| **CC1** (3 kLoC) | 32 | 2 | 4 | 26 (13) |
| **CC2** (19 kLoC) | 72 | 20 | 22 | 30 |
| **CC3** (15 kLoC) | 16 | 0 | 3 | 13 |
| **CC4** (14.5 kLoC) | 107 | 14 | 13 | 80 |
| **CC5** (15 kLoC) | 9 | 1 | 3 | 5 |
| total (66.5 kLoC) | 236 | 37 | 45 | 154 (141) |

**Table 1: Reported number of faults by SMELL for five components.**

## 6. EXPERIMENTAL RESULTS

## 6.1 General Remarks

Table 1 presents the results of applying SMELL on 5 relatively small ASML components. The first column lists the component that was considered together with its size, column 2 lists the number of faults reported by SMELL, column 3 contains the number of false positives we manually identified among the reported faults, column 4 shows the number of SMELL limitations (as discussed in the previous section) that are encountered and automatically recognised, and finally column 5 contains the number of validated faults, or 'true' faults.

Four of the five components are approximately of the same size, but there is a striking difference between the numbers of *reported* faults. The number of reported faults for the CC3 and CC5 components is much smaller than those reported for the CC2 and CC4 components. When comparing the number of *validated* faults, the CC4 component clearly stands out, whereas the number for the other three components is approximately within the same range.

Although the CC1 component is the smallest one, its number of validated faults is large compared to the larger components. This is due to the fact that a heavily-used macro in the CC1 component contains a fault. Since SMELL is run after macro expansion, a fault in a single macro is reported at every location where that macro is used. In this case, only 13 faults need to be corrected (as indicated between parenthesis), since the macro with the fault is used in 14 places.

The number of validated faults reported for the CC5 component is also interestingly low. This component is developed by the same people responsible for the EHM implementation. As it turns out, even these people violate the idiom from time to time, which shows that the idiom approach is difficult to adhere to. However, it is clear that the CC5 code is of better quality than the other code.

Overall, we get 236 reported faults, of which 45 (19 %) are reported by SMELL as a limitation. The remaining 191 faults were inspected manually, and we identified 37 false positives (16 % of reported faults). Of the remaining 154 faults, 141 are unique, and so in other words, we found 2.1 true faults per thousand lines of code.

## 6.2 Fault Distribution

A closer look at the 141 validated faults shows that 13 faults are due to a function not returning, 28 due to the wrong error variable being returned, 54 due to unsafe assignments, 10 due to incorrect logging, and 36 due to an assigned and logged value mismatch.

The *unsafe assignment* fault occurs when the error variable contains an error value that is subsequently overwritten. This kind of fault occurs the most (54 out of 141 = 38%), followed by the *assigned and logged value mismatch* (36 out of 141 = 26%). If we want to minimise the exception handling faults, we should develop an alternative solution that deals with these two kinds of faults.

Accidental overwriting of the error value typically occurs because the control flow transfer when the exception is raised is not implemented correctly. This is mostly due to a forgotten guard that involves the error variable ensuring that normal operation only continues when no exception has been reported previously. An example of such a fault is found in Figure 5.

The second kind of fault occurs in two different situations. First, as exemplified in Figure 4, when a function is called and an exception is received, a developer might link an exception to the received one, but forgets to assign the linked exception to the error variable. Second, when a root error is detected and a developer assigns the appropriate error value to the error variable, he might forget to log that value.

## 6.3 False positives

The number of false positives is sufficiently low to make SMELL useful in practice. A detailed look at these false positives reveals the reasons why they occur and allows us to identify where we can improve SMELL.

Of the 37 false positives identified, 23 are due to an incorrect identification of the error variable, 7 are due to SMELL getting confused when multiple error variables are used, 4 occur because an infeasible path has been followed, and 3 false positives occur due to some other (mostly domain-specific) reason.

These numbers indicate that the largest gain can be obtained by improving the error variable identification algorithm, for example by trying to distinguish ASML error variables from "ordinary" error variables. Additionally, they show that the issue of infeasible paths is not really a large problem in practice.

## 7. AN ALTERNATIVE EXCEPTION HANDLING APPROACH

In order to reduce the number of faults in exception handling code, alternative approaches to exception handling should be studied. A solution which introduces a number of simple macros has been proposed by ASML, and we will discuss it here. We thereby keep in mind that we know that the two most frequently occurring faults are overwriting of the error value and the mismatch between the value assigned to the error variable and the value actually logged.

The solution is based on two observations.

First, it encourages developers to no longer write assignments to the error variable explicitly, and it manages them automatically inside the macros. Such assignments can either be constant assignments, when declaring a root error, or function-call assignments, when calling a function. By embedding such assignments inside specific macros and surrounding them with appropriate guards, we can prevent accidental overriding of error values.

Second, the macros ensure that assignments are accompanied by the appropriate `LOG` calls, in order to avoid a mismatch between logged and assigned values. As explained in the previous section, such a mismatch occurs when declaring a root error or when linking to a received error. Consequently, we introduce a `ROOT_LOG` and `LINK_LOG` macro that should be used in those situations and that take care of all the work.

The proposed macro's are defined in Figure 6. The `ROOT_LOG` macro should be used whenever a root error is detected, while the `LINK_LOG` macro is used when calling a function and additional information can be provided when an error is detected. Additionally, a `NO_LOG` macro is introduced that should be used when calling a function and not linking extra information if something goes wrong.

```
#define ROOT_LOG(error_value, error_var)\
      error_var = error_value;\
      LOG(error_value, OK);\

#define LINK_LOG(function_call, error_value, error_var)\
   if(error_var == OK) {\
      int _internal_error_var = function_call;\
      if(_internal_error_var != OK) {\
         LOG(error_value, _internal_error_var);\
         error_var = error_value;\
      }\
   }

#define NO_LOG(function_call, error_var)\
   if(error_var == OK) \
      error_var = function_call;
```

**Figure 6: Definitions of proposed exception handling macro's.**

Using these macros, the example code from Section 3 is changed into the code that can be seen in Figure 7.

It is interesting to observe that using these macros drastically reduces the number of (programmer visible) control-flow branches. This not only improves the function's understandability and maintainability, but also causes a significant drop in code size, if we consider that the return code idiom is omnipresent in the ASML code base. Moreover, the exception handling code is separated from the ordinary code, which allows the two to evolve separately. More research is needed to study these advantages in detail.

The solution still exhibits a number of drawbacks.

First of all, the code that cleans up memory resources remains as is. This is partly due to the fact that we did not focus on such code, since we postulate that it belongs to a different concern. However, such code also differs significantly between different functions and source components, which makes it harder to capture it into a set of appropriate macros.

Second, reliability checks are still not available. It remains the developer's responsibility to use the macros and use them correctly. Developers can still explicitly assign something to the error variable, without using the correct macro, for example. Or, they can still raise exceptions that should not be raised by a particular function.

Last, the macros do not tackle faults that concern the returning of the appropriate error value. Since this was a deliberate choice, because such errors are rather scarce and can be easily found, this comes as no surprise.

```
int f(int a, int* b) {
   int r = OK;
   bool allocated = FALSE;
   r = mem_alloc(10, (int *)b);
   allocated = (r == OK);
   if((a < 0) || (a > 10))
       ROOT_LOG(PARAM_ERROR,r);
   LINK_LOG(g(a),LINKED_ERROR,r);
   NO_LOG(h(b), r);
   if((r != OK) && allocated)
      mem_free(b);
   return r;
}
```

**Figure 7: Function `f` implemented by means of the alternative macros.**

## 8. DISCUSSION

In our examples, we found 2.1 deviations from the return code idiom per 1000 lines of code. In this section, we discuss some of the implications of this figure, looking at questions such as the following: How does the figure relate to reported defect densities in other systems? What, if anything, does the figure imply for system reliability? What does the figure teach us on idiom and coding standard design?

### 8.1 Representativeness

A first question to be asked is to what extent our findings are representative for other systems.

The software under study has the following characteristics:

- It is part of an embedded system in which proper exception handling is essential;
- Exception handling is implemented using the return code idiom, which is common for C applications;
- Before release, the software components in question are subjected to a thorough code review;
- The software is subjected to rigorous unit, integration, and system tests.

In other words, we believe our findings hold for software that is the result of a state-of-the-art development process and that uses an exception handling mechanism similar to the one we considered.

The reason so many exception handling faults occur is that current ways of working are not effective in finding such faults: tool support is inadequate, regular reviews tend to be focused on "good weather behaviour" — and even if they are aimed at exception handling faults these are too hard to find, and testing exception handling is notoriously hard.

### 8.2 Defect Density

What meaning should we assign to the value of 2.1 exception handling faults per 1000 lines of code (kLoC) we detected?

It is tempting to compare the figure to reported defect densities. For example, an often cited paper reports a defect density between 5 and 10 per kLoC for software developed in the USA and Europe [11]. More recently, in his ICSE 2005 state-of-the-art report, Littlewood states that studies show around 30 faults per kLoC for commercial systems [22].

There are, however, several reasons why making such comparisons is questionable, as argued, for example, by [13]. First, there is neither consensus on what constitutes a defect, nor on the best way to measure software size in a consistent and comparable way. In addition to that, defect density is a product measure that is derived from the process of finding defects. Thus, "defect density may tell us more about the quality of the defect finding and reporting process than about the quality of the product itself" [13, p.346]. This particularly applies to our setting, in which we have adopted a new way to search for faults.

The consequence of this is that no conclusive statement on the relative defect density of the system under study can be made. We cannot even say that our system is of poorer quality than another with a lower reported density, as long as we do not know whether the search for defects included a hunt for idiom errors similar to our approach.

What we can say, however, is that a serious attempt to determine defect densities should include an analysis of the faults that may arise from idioms used for dealing with crosscutting concerns. Such an analysis may also help when attempting to explain observed defect densities for particular systems.

### 8.3 Reliability

We presently do not know what the likelihood is that an exception handling fault actually leads to a failure, such as an unnecessary halt, an erroneously logged error value, or the activation of the wrong exception handler. As already observed by Adams in 1984, more faults need not lead to more failures [1]. We are presently investigating historical system data to clarify the relation between exception handling faults and their corresponding failures. This, however, is a time consuming analysis requiring substantial domain knowledge in order to understand a problem report, the fault identified for it (which may have to be derived from the fix applied) and to see their relation to the exception handling idiom.

### 8.4 Idiom design

The research we are presenting is part of a larger, ongoing effort in which we are investigating the impact of crosscutting concerns on embedded C code [5, 4]. The traditional way of dealing with such concerns is by devising an appropriate coding idiom. What implications do our findings have on the way we actually design such coding idioms?

One finding is that an idiom making it too easy to make small mistakes can lead to many faults spread all over the system. For that reason, idiom design should include the step of constructing an explicit fault model, describing what can go wrong when using the idiom. This will not only help in avoiding such errors, but may also lead to a revised design in which the likelihood of certain types of errors is reduced.

A second lesson to be drawn is that the possibility to check idiom usage automatically should be taken into account: static checking should be designed into the idiom. As we have seen, this may require complex analysis at the level of the program dependence graph as opposed to the (elementary) abstract syntax tree.

## 9. CONCLUDING REMARKS

### *Contributions*

Our contributions are summarised as follows. First, we provided empirical data about the use of an exception handling mechanism based on the return code idiom in an industrial setting. This data shows that the idiom is particularly error prone, due to the fact that it is omnipresent as well as highly tangled, and requires focused and well-thought programming. Second, we defined a series of steps to regain control over this situation, and answer the specific questions we raised in the introduction. These steps consist of the characterisation of the return code idiom in terms of an existing model for exception handling mechanisms, the construction of a fault model which explains when a fault occurs in the most error prone components of the characterisation, the implementation of a static checker tool which detects faults as predicted by the fault model, and the introduction of an alternative solution, based on experimental findings, which is believed to remove the faults most occurring.

We feel these contributions are not only a first step toward a reliability check component for the return code idiom, but also provide a good basis for (re)considering exception handling approaches when working with programming languages without proper exception handling support. We showed that when designing such idiom-based solutions, a corresponding fault model is a necessity to assess the fault-proneness, and the possibility of static checking should be seriously considered.

### *Future work*

There are several ways in which our work can be continued:

- apply SMELL to more ASML components, in order to perform more extensive validation. Additionally, some components already use the macros presented in Section 7, which allows us to compare the general approach to the alternative approach, and assess benefits and possible pitfalls in more detail. We initiated such efforts, and are currently analysing approximately two million lines of C code for this.

- apply SMELL to non-ASML systems, such as open-source systems, in order to generalise it and to present the results openly.

- apply SMELL to other exception handling mechanisms for C, such as those based on the `setjmp/longjmp` idiom, to analyse which approach is most suited.

- investigate aspect-oriented opportunities for exception handling, since benefits in terms of code quality can be expected if exception handling behaviour is completely separated from ordinary behaviour [21].

### *Acknowledgements*

This work has been carried out as part of the Ideals project under the auspices of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter program. The authors would like to thank Remco van Engelen, Christian Bakker, Michiel Kamps and Pieter ten Pierick for their help with and comments on SMELL.

## 10. REFERENCES

[1] E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[2] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3. ACM, January 2002.

[3] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[4] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating Idiomatic Crosscutting Concerns. In *Proceedings of the International Conference on Software Maintenance*, pages 37– 46. IEEE Computer Society, 2005.

[5] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 200–209. IEEE Computer Society, 2004.

[6] M. Bush. Improving software quality: the use of formal inspections at the jpl. In *Proceedings of the International Conference on Software Engineering*, pages 196–199. IEEE Computer Society, 1990.

[7] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for fi nding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.

[8] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, pages 235–244. ACM, November 2002.

[9] F. Christian. *Exception handling and tolerance of software faults*, chapter 4, pages 81–107. John Wiley & Sons, 1995.

[10] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verifi cation in polynomial time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–68. ACM, May 2002.

[11] M. Dyer. The cleanroom approach to quality software development. In *Proceedings of the 18th International Computer Measurement Group Conference*, pages 1201–1212. Computer Measurement Group, 1992.

[12] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specifi c, programmer-written compiler extensions. In *4th Symposium on Operating System Design and Implementation*, pages 1–16. USENIX Association, 2000.

[13] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A rigorous and Practical Approach*. PWS Publishing Company, second edition, 1997.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[15] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott. Robustness testing of java server applications. *IEEE Transactions on Software Engineering*, 31(4):292 – 311, 2005.

[16] N. H. Gehani. Exceptional C or C with exceptions. *Software Practice and Experience*, 22(10):827–848, 1992.

[17] S. Johnson. Lint, a C Program Checker. Technical Report 65, Bell Laboratories, Dec. 1977.

[18] J. Lang and D. B. Stewart. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Transactions on Programming Languages and Systems*, 20(2):274 – 301, 1998.

[19] P. A. Lee. Exception handling in C programs. *Software Practice and Experience*, 13(5):389–405, 1983.

[20] J.-L. Lions. Ariane 5 flight 501 failure. Technical report, ESA/CNES, 1996.

[21] M. Lippert and C. V. Lopes. A study on exception detecton and handling using aspect-oriented programming. In *Proceedings of the International Conference on Software Engineering*, pages 418 – 427. IEEE Computer Society, 2000.

[22] B. Littlewood. Dependability assessment of software-based systems: state of the art. In *Proceedings of the International Conference on Software Engineering*, pages 6–7. ACM Press, 2005.

[23] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *5th Symposium on Operating System Design and Implementation*. USENIX Association, 2002.

[24] S. Paul and A. Prakash. A Framework for Source Code Search using Program Patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[25] E. S. Roberts. Implementing exceptions in C. Technical Report 40, Digital Systems Research Center, 1989.

[26] M. Robillard and G. C. Murphy. Regaining control of exception handling. Technical Report TR-99-14, Department of Computer Science, University of British Columbia, 1999.

[27] M. P. Robillard and G. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, 12(2):191–221, 2003.

[28] S. Sinha and M. J. Harrold. Criteria for testing exception-handling constructs in java programs. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 1999.

[29] T. Tourwé and T. Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 91 – 100. IEEE Computer Society, 2003.

[30] W. N. Toy. Fault-tolerant design of local ess processors. In *Proceedings of IEEE*, pages 1126–1145. IEEE Computer Society, 1982.

[31] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Working Conference on Reverse Engineering*, pages 97–106. IEEE Computer Society, 2002.

[32] H. Winroth. Exception handling in ANSI C. Technical Report ISRN KTH NA/P–93/15–SE, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden, 1993.