Stayin' Alert:
Moulding Failure and Exceptions to Your Needs

Anya Helene Bagge Valentin David Magne Haveraaen Karl Trygve Kalleberg

University of Bergen, Norway
{anya,valentin,magne,karltk}@ii.uib.no

Abstract

Dealing with failure and exceptional situations is an important but tricky part of programming, especially when reusing existing components. Traditionally, it has been up to the designer of a library to decide whether to use a language's exception mechanism, return values, or other ways to indicate exceptional circumstances. The library user has been bound by this choice, even though it may be inconvenient for a particular use. Furthermore, normal program code is often cluttered with code dealing with exceptional circumstances.

This paper introduces an alert concept which gives a uniform interface to all failure mechanisms. It separates the handling of an exceptional situation from reporting it, and allows for retro-fitting this for existing libraries. For instance, we may easily declare the error codes of the POSIX C library for file handling, and then use the library functions as if C had been extended with an exception mechanism for these functions – a *moulding* of failure handling to the user's needs, independently of the library designer's choices.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Reliability, Languages, Design

Keywords Failure, Errors, Partiality, Guarding, Aspects, Separation of Concerns, Alert Reporting and Handling, Domain-Specific Exception Language, Abstraction, Mouldable Programming

1. Introduction

Wherever there is software, there are errors and exceptional situations, and these must always be considered when writing and maintaining programs. Programming failure handling code is a tedious and error-prone task. Dealing with every possible exceptional situation leads to cluttered and hard to read code; not dealing with errors can have costly or perhaps even fatal consequences.

Some have argued that error handling should be avoided altogether. Instead, programs should be written so that errors never occur. Algorithms should be formulated so as to remove the exceptional corner cases, as this improves both the readability and maintainability of the code. This view is fundamental to the design of SPARK Ada [2], where the Ada exception mechanism has been removed in an attempt at making validation and verification easier. This ideal advocated by such a "keep errors out" approach is certainly desirable. It is generally preferable to write algorithms with as few corner cases as possible.

In many cases, however, removing the errors altogether is simply not feasible [27]. Most modern applications run in multi-user, multi-process environments where they share resources such as storage and network with other applications. In these situations, operations on files, network connections and similar operating system resources can always fail, due to interaction with other programs on the running system or external devices.

Errors and exceptional situations need not always be caused by external factors, however. Even in situations where resource requirements are known in advance and guaranteed to be available, exceptional situations may occur, as none of the mainstream languages support resource-aware type systems [31].

As an example, consider the implementation of a simple abstract data type, say, a hash table, that is intended for other developers to reuse. In the case where the user (the *caller*) tries to look up a value for a non-existent key, an exceptional situation has occurred. Some possibilities for dealing with such a situation are:

- *Undefinedness*: this situation is outside the specified behaviour of the hash table. The caller cannot have any expectations as to what will happen.
- *Termination*: the program will terminate when this situation occurs. It is up to the caller to ensure that this does not happen.
- *Alert the caller*: report that an exceptional condition occurred. Given proper language mechanisms, alerts allow the user of the hash table to implement alert handling, such as logging, recovering from or ignoring the failure.

Undefinedness requires no language support, and termination can usually be implemented by a call to an `exit` function. In languages supporting Design by Contract (discussed in Section 2.1), termination is automatic if a function fails to satisfy declared conditions either before or after invocation.

Several different alert reporting mechanisms are in common use. Goodenough [10] first introduced the exception handling mechanism[1] that is now found in most modern languages, and is

[1] The word 'exception' was coined as a way of emphasising that exceptions are not just for handling errors, but can be used for any kind of exceptional circumstance. However, it is easy to confuse the concept of handling exceptional circumstances and the exception handling language constructs found in many languages. We have therefore elected to use the word 'alert' for any reported exceptional situation, independent of the alert reporting mechanism and the alert handler, which receives the alert report and deals with it appropriately. The word 'exception' on its own will refer to the language construct.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.

currently the recommended way of handling exceptional situations. Returning a special error value, often -1 or null, or setting a global variable is another common technique, often used in older code and languages. Other than exceptions, most reporting mechanisms are ad-hoc, in that there is no way to declare which mechanism is used. Conventions do exist – for example, most POSIX [24] functions report errors by returning -1 – but they are not declared explicitly in the code, making it difficult to automate alert handling. We therefore propose that each function declares its alert reporting. For example, a hash table lookup function may declare that it returns null if the key was not found:

```
val lookup(tbl t, key k)
  alert NotFound post(value == null);
```

Handling alerts is no easy task either. Different reporting mechanisms have different default handlers – exceptions, for example, typically terminate the program if they are not caught, whereas return values are ignored if they are not explicitly tested. Furthermore, different alert reports are checked in radically different ways; exceptions are received by a **try/catch** clause somewhere in the call hierarchy, return values must be checked after each return – often tedious and inconvenient. Changing the report mechanism means changing all handlers. We propose a way of declaring handlers which is independent of the alert reporting mechanism, and which can apply at various granularities, from a single expression to the whole program. For example, the following handler ensures that NotFound errors from the lookup function is handled by substituting the string "(unknown)":

```
on NotFound in lookup() use "(unknown)";
```

Our contribution in this paper is a detailed discussion of failure handling mechanisms and the proposal of a language construct for alerts: Alert reporting may be declared for precondition and postcondition violations, exceptions, error flags and return codes, simplifying the task of staying alert for run-time problems. Alert handlers can be defined independently of the reporting mechanism, allowing a library implementor to alert its user in a way convenient for the library, and a library user to handle the alert in a convenient way at the call site. Alert handlers can be declared at a per-function and per-call-site basis, but it is also possible to declare policies common to a group of functions, such as a class or a library. In this way we can relatively easily retro-fit alert declarations for legacy code, e.g., the POSIX C library, easing the burden of checking in all kinds of strange ways for relevant I/O errors. Hence we approach *mouldable programming*, a way of moulding programming to our needs, and not being forced to program in strange ways due to arbitrary choices from language and library designers, or from perceived expectations from a user community.

This paper is organised as follows. In Section 2, we elaborate on the problem of handling failures and exceptional situations. In Section 3, we discuss separation of concerns, and granularity. In Section 4 we introduce our alert language extension, and continue by discussing its implementation in Section 5. In Section 6 we discuss related work, leading up to a concluding discussion of our language extension in Section 7.

2. Problem

The problem we are facing, is implementing an *alert protocol* between callers and callees that can transmit status information from the callee about the validity of its computed result back to the caller. Goodenough [10] points out that this is a way to extend an operation's domain (input space) or range (output space). The caller will declare an *alert handler* for the types of alerts it wants to handle, and the callee may *report an alert* during its computation, thus becoming the *(alert) reporter*.

2.1 Alert Reporting Mechanisms

Current programming paradigms and languages provide a number of ways for dealing with failure, dating back to the earliest days of programming. Hill [12] discusses possible mechanisms, anno 1969, which includes specific return values, use of gotos to parameterised labels, callbacks, global error flags, and passing pointers to variables which will receive an error status.

Return Values Designating at least one value in the domain of the return type as an error marker is perhaps the most prevalent form of alerting. This technique is frequently found in operating system APIs, such as POSIX and Win32, in many language standard libraries, and in many frameworks. Functions returning objects often use null as such a marker, functions returning numeric values for file handles or indexes often use -1. If the return type only allows for one error marker, an additional mechanism, such as a global flag, is needed to distinguish between different kinds of errors.

The IEEE floating-point arithmetic standard [9] allows a wide range of error return values. Some of these automatically propagate through an expression, like NaN – "Not a Number". NaN occurs as a result of $0/0$, $\sqrt{-1}$, $\log(-1)$, etc. A more interesting error value is $+\infty$ or $-\infty$, which is the result of e.g. M/m where a very large number M is divided by a very small number m, resulting in numerical overflow – a number too large to be represented as a floating point number. Infinities propagate through addition, subtraction and multiplication, but disappear after division. The expression $a + 4/(M/m)$ yields a, as 4 divided by infinity yields 0.

If the return value for failure can also be a valid return value, for example if division by zero returns zero, we are faced with the so-called semipredicate problem: it is not possible to know if the return value signifies a failure or a valid value.

A property of the return value mechanism is that it will only propagate the alert one level, to its immediate caller. Also, it requires no alert handler setup or teardown, and thus has no overhead.

Global Error Flags Many older APIs, such as POSIX and Win32 use global error flags, often in conjunction with special return values, to elaborate on a failed function. In Win32, the function GetLastError is used to retrieve the failure code of the previously executed system call. In POSIX, the global errno variableserves an identical purpose.

The use of a global error flag variable is not thread safe. Unless special consideration is taken, multiple threads in the same process will share the same error flag variable, making it impossible to know which of the previous threads' system calls a given error belongs to. This is alleviated by having global error function, like GetLastError, instead.

Long Distance Jumps In C, the functions setjmp and longjmp are used to transfer control directly from one stack frame to one which is arbitrarily higher up. This report mechanism is often used to propagate errors many levels, but can only send an integer value. This is a low-level C/Unix-specific technique, which is also found as RtlUnwind in Win32. Both alternatives rely on low-level machine-specific register set saving and compiler knowledge. Another drawback is the difficulty of freeing allocated resources properly before the handlers for such resources leave the variable scope.

Exceptions Today, the most common way of alerting is to use the exception mechanism introduced in [10], in languages that provide this, such as CLU [18], C++ [29], Java [11], Ada [30], ML [23] and Python [32].

Raising an exception consists of two parts: First, a function, say A, sets up an exception handler listening for a particular type of exception E, using a **try/catch** (Java), **handle** (SML) or

try/except (Python) construct. It then invokes the function B, either directly or indirectly. B raises the exception E by invoking the **raise** (Python, SML) or **throw** (Java) language construct, and the search for an appropriate exception handler starts. Each stack frame is consulted in succession, until one with a handler for E is found. If no new handler for E is declared in the functions between B and A on the stack, control is transferred from B to the handler declared in A.

For the languages above, after the handler in A finishes, execution continue in A. In other languages, it is possible to either resume after the **raise** statement in B, or restart B, see Section 6 for details.

Exceptions may be either *checked* – must be declared by all functions that may throw them, both directly and indirectly – or *unchecked* – may be thrown without being mentioned in a function's list of throwable exceptions. In Java, checked exceptions are the default, but unchecked exceptions are used for catastrophic errors, such as out-of-memory errors and disk failure.

Exception handlers need not only be declared as markers on the stack. They can also be attached to classes, statically giving each class its own handler, or to objects giving each object a specific handler. This is discussed in Section 6.

Condition System The PL/I **ON** condition system, allows the programmer to attach handler blocks for pre-defined language exceptions that may occur in expressions, such as division by zero and end of file. These handlers are installed and removed dynamically. Extensions of this idea can be found in Dylan, Smalltalk and Lisp, which have systems for detecting exceptional situations based on (a restricted description of) the state space of a program. When a given failure condition is met, control is transferred to a specified error handler which can elect to try error recovery followed by resume, or terminate the function that threw the exception.

Event Handlers Event handlers and POSIX signal handlers both provide a callback mechanism which may be used for passing error notifications from the operating system to the application, or between parts of an application. This model requires no special language support, and is usually tied to the API or framework the application was written with, e.g. POSIX (signals) or Win32 GDI (events).

Guarding A pre-condition may be declared on a function, testing beforehand whether the function will return normally with the data. Effectively, pre-conditions ensure that the input falls within a function's domain, and attempts to ascertain whether the state of the system allows the function to complete. Formulating such a pre-condition may not always be possible, e.g., during complex interaction with external resources.

Contracts A significant extension to guarding is *design by contract*, described by Meyer [22]. In this technique, explicit pre- and post-conditions are declared on every function. Whenever either fails, the program terminates immediately. A contract should never be checked by the caller; contract verification must happen during the implementation phase, not at runtime. Eiffel [21] was the first language to support and enforce contracts, but also comes with a notion of exception handling. A routine may have a rescue handler declared for it, which may either provide some default return value, retry the routine, or fail. In the latter case, the failure will be propagated to the method's caller.

Goto The use of goto as an exception handling technique has almost disappeared with the introduction of various exception handling language features. In some restricted domains, such as the kernel code of operating systems, where space and performance considerations outweigh readability, gotos are still prevalent.

2.2 Alert Handling Policies

Even using the same basic alert reporting, different usage policies lead to large differences in alert handling in the design of frameworks and libraries. The policy about retrying on failure, is one example. Unix leaves it to the user to retry failing operating system calls, for example if a long running kernel operation is preempted. Windows (and BSD Unix variants), on the other hand, retries preempted operations automatically.

In many languages, guiding principles exist about using the exception handling feature of the language. This is the case for Java, where the general recommendation is to use exceptions for alerting. Despite such principles, there are numerous examples in the library where error return values are used, among them in the implementation of the hash table.

2.3 A Game of Anticipation

The state of the art is to use design experience on a case-by-case basis to provide suitable alerts. Specifically, there exists no declarative way to select the desired error handling mechanism for part of a program. The need for design experience comes from the fact that fundamental tradeoffs between the caller and callee of an abstraction must be managed. The caller is the party which will be implementing the alert handling. As the various handling techniques have different affinities with alert reporting, and every caller is potentially different, the implementer of the callee must anticipate the handling techniques that will be used by the callers.

From the callee side, the ideal reporting mechanism may depend on the implementation of an algorithm. For instance, if we are within a deeply nested data traversal, it may be more convenient to throw an exception than to use return values.

Another consideration is who should do error checking on the input parameters. Should the callee accept erroneous input and produce garbage? Should the callee do all checking? This decision is usually coupled with significant performance tradeoffs.

The amount of anticipation required by the implementer of the callee is significant, perhaps especially in core language libraries. In Java, an example can be found in `java.util.Queue`, which provides pairs of identical functions, save for differences in alert reporting: `poll()` and `remove()` can both be used to remove the head of a queue. `poll()` returns `null` if there is no head, whereas `remove()` throws an exception in the same case. Similarly, `add()` throws an exception if a new element cannot be added to a queue, whereas `offer()` returns `false` if the insertion failed.

Determining a suitable reporting mechanism when implementing a callee is compounded by another problem: the caller is the final arbiter of what is normality and what is failure. Returning `null` from a hash table lookup may in one application be completely acceptable, and not constitute a special case in the algorithm using the hash table. In another application, it will be the sign of severe data corruption and violation of crucial invariants. In the first case, returning a *null* is neither an exceptional case, nor an error, and this situation is therefore not a candidate for alert handling. In the second, it is critical that proper alerting be used.

3. Separation of Concerns

Separation of Alert Reporting and Handling Although the mechanisms in Section 2.1 are essentially equivalent in that they all report exceptional situations (possibly with additional information), the default action taken when an error occurs differs. For return values and error flags, the default is to ignore the error. For exceptions, the default is to propagate the exception through the call hierarchy, possible leading to termination of the program. For guarding, the default is not to guard, i.e., ignore the error.

In all cases, the callee implicitly decides the default action in case of an error, by choosing a given report mechanism. This is unfortunate, since the goal of raising an alert to the caller is to let the caller decide the appropriate course of action (otherwise, the callee could simply handle everything on its own). Additionally, the choice of alert mechanism is often based on implementation pragmatics, rather than whether the default action is likely to be appropriate for the severity of the error. If the callee is changed to use a different mechanism, all call sites must be updated. Thus, we have a *tangling* of alert mechanisms (callee implementation) and alert handling (caller implementation).

Separation of Normality and Exceptionality With existing techniques for handling exceptional situations, we get a tangling of a program's normal behaviour and its exceptional behaviour. If our handling policy is to report small errors to the user and abort the program on serious errors, we have to code this into all places where errors are handled. Thus we end up with a mix of code dealing with normal circumstances, and code dealing with exceptional circumstances (alert behaviour). This leads to cluttered code and maintainability problems: if we wish to change our policy for some errors, we may need to change a lot of code in many different parts of our program.

This problem has also been observed in [16], where aspects are used to untangle the alert behaviour from the normal behaviour. The authors advocate that alert handling code be put in separate declarations – aspects – instead of being scattered around the code (see Section 6).

Granularity In some cases, mixing normal and alert behaviour may be beneficial; for example, by taking alerts into account locally, we may compensate, e.g., by substituting a different return value. In some cases, such decisions must be made for each call site; in other cases, we may be able to provide a common policy for an entire class, a module or a set of functions.

An example of this is IEEE floating-point expressions. Here a NaN is propagated through the expression and may be tested for, while an overflow ($\pm\infty$) may be propagated or consumed depending on the expression itself.

4. Alert Language Extension

While the existing body of research on exception handling addresses many of the concerns we have mentioned above, one area of the design space remains relatively unexplored: how to extract and declare separately the handling of exceptional situations. This is what we will address in the next sections.

Our mouldable abstraction of alert handling provides for separation of mechanism and handlers, separation of normal behaviour and failure behaviour, and allow decisions to be taken at the appropriate level of granularity, i.e. at the expression, statement, function, class, module or component level. Furthermore, our proposed solution allows the callee to declare what is normality and what is exceptional; allows the caller to declare the desired alert handling policies; can be applied retroactively to existing libraries; and is able to distinguish different types of errors.

A grammar for the alert extension is presented in Figures 3, 4 and 5. Although our prototype implementation (discussed in Section 5) is an extension of the C language, we will discuss the extension in terms of a C/C++/Java/C#-like language with an exception facility.

The grammars in this paper are meant for human consumption, and not for use directly in an implementation. Non-terminals written in upright font are meant as hooks into the base language. Non-terminals ending in *name*, *type* or *expr* are all names, types or expressions of the appropriate type. The notation " $\overset{*}{,}$ " and " $\overset{+}{,}$ " is used for comma-separated lists.

declaration ::= **alert {** *alert-def*$\overset{*}{,}$ **}** *super-alert** **;**
alert-def ::= *alert-name* [(parameter-list)] *super-alert**
super-alert ::= **:** *alert-name*

Figure 1. Grammar for the declaring new alerts.

EBADF	Bad file descriptor
EINTR	Interrupted system call
EIO	Input/output error
ENOENT	No such file or directory
ENOMEM	Insufficient memory available

Figure 2. A small selection of POSIX error codes, used, e.g., for `open`, `read` and `write`. The codes are set in the global `errno` variable, and should be checked whenever a function raises an error (typically by returning `-1`)

4.1 Distinguishing Different Alerts

Alerts are declared with the **`alert`** declaration, using syntax similar to the **`enum`** declaration (see grammar in Figure 1):

```
alert {MyAlert};
```

Multiple comma-separated alerts can be declared in the same declaration. For example, a selection of POSIX error codes is listed in Figure 2. These errors can occur during normal file operations, such as `open`, `read` and `write`. We can give each of them a unique alert name with the following declaration, with different (case-sensitive) names to avoid name clashes:[2]

```
alert {eBadF, eIntr, eIO, eNoEnt, eNoMem};
```

If we look at the selected error codes (and a POSIX reference), we see that they fall into roughly four categories: temporary conditions (EINTR); system problems outside the program's control (ENOMEM, EIO); problems that might be correctable with user help (ENOENT); and programming errors (EBADF). Thus, it is useful to be able to group them, so that we may, for example, automatically retry temporary failures, ask the user for a new file name on permission or missing file problems, and abort the program on system errors and programmer errors. To do this, we organise our alerts in a hierarchy, similar to an inheritance hierarchy in OO programming (which is also used for exceptions – in Java, for example):

```
alert {Retry, AskUser, FatalSys, FatalBug};
alert {eNoEnt : AskUser};
```

The colon separates a sub-alert from its super-alert in a declaration. Multiple alerts can be assigned a common super-alert:

```
alert {eIO, eNoMem} : FatalBug;
```

Additionally, an alert may have more than one super-alert:

```
alert {StupidMistake};
alert {eBadF} : FatalBug : StupidMistake;
```

To avoid cycles in the inheritance graph, alerts must be declared before they are used as super-alerts. The built-in alert `Alert` is the super-alert of all other alerts.

Finally, we note that it is sometimes useful for the callee to pass some information back to the caller. To do this, the alert must be declared with one or more arguments. For example, an `Error` alert which allows a message to be passed to the caller:

```
alert {Error(char *msg)};
```

[2] Alert names are in a separate namespace, but error codes are commonly declared with macros in C, which ignores namespace boundaries.

```
declaration ::= fun-dcltr (alert alert-rep | throws-clause)* [fun-body]
declaration ::= alertrepdef alert-rep alert-rep-name ;
declaration ::= funspace funspace (alert alert-rep)+
  alert-rep ::= [alert-name] pre [unless] ( cond-expr )
  alert-rep ::= [alert-name] post [unless] ( cond-expr )
  alert-rep ::= [alert-name] on throw exception-name
  alert-rep ::= alert-rep-name
  alert-rep ::= { alert-rep *, }
```

Figure 3. Grammar for specifying alert reporting. The *funspace* non-terminal is defined in Figure 5.

We will see below how the values are passed from callee to caller.

4.2 Specifying the Alert Reports of a Function

Alert reports are specified at the callee side with an **alert** clause in the function declaration, c.f. grammar in Figure 3. Possible reporting mechanisms include condition checks before (pre conditions, useful for guarding) and after (post conditions, for checking return values and global error flags) a call, and exceptions. For example, to declare that a hash table lookup function lookup returns 0 on failure, we write:

```
val lookup(tbl t, key k) alert post(value == 0);
```

To use a more specific alert than the default Alert, we simply add the name of the desired alert:

```
val lookup(tbl t, key k) alert NotFound post(value == 0);
```

The **post**-clause takes an expression, which is evaluated after the function call returns – if the expression is true, the function has failed. The special keyword **value** can be used to check the call's return value (the type of **value** is that of the return type of the callee). Similarly, the **pre**-clause takes a condition which is checked *before* the function is called.

The condition expressions may be arbitrarily complex, but should only use globally accessible names, arguments to the call, and **value**. Arguments are referred to by the name they are given in the function declaration, and have whatever values they have at the time of checking (before or after the call).

```
// alert if table t cannot be expanded due to memory constraints
int insert(tbl t, key k, val v)
  alert eNoMem post (value == -1 && errno == ENOMEM);
```

Specifying a condition with the **unless** keyword negates it, providing a more intuitive way of specifying invariants and pre/post conditions (which are often specified in terms of what is normal, and not in terms of what is exceptional).

```
// separate success/failure flag if no return values can be used for alerting
val lookup(tbl t, key k, bool *success)
  alert NotFound post unless(*success);
```

Exceptions (if the language supports it) can be declared with a **throws** (Java) or **throw** (C++) clause, provided that the exception name has also been declared as an alert:

```
alert {AnException};
int f() throws AnException;
```

If a mapping between exceptions and alert names is desired, an **on throw** clause may be used:

```
int f() throws AnException
  alert Error on throw AnException;
```

In all cases, information may be passed to a handler using alert parameters:

```
declaration ::= handler handler-name ( parameter-list ) statement
declaration ::= on alert-pattern statement
  statement ::= retry [( argument-list )] [max int-expr] ;
  statement ::= use expr ;
  statement ::= handler-name ( argument-list ) ;
  statement ::= statement-body on alert-pattern statement
alert-pattern ::= single-alert +; [in funspace]
alert-pattern ::= alert-pattern or alert-pattern
single-alert ::= alert-name [( parameter-list )]
       expr ::= expr <: [alert-pattern] : handler
    handler ::= expr | { statement }
    handler ::= handler-name ( argument-list )
```

Figure 4. Grammar for alert handlers. The *funspace* non-terminal is defined in Figure 5.

```
val lookup(tbl t, key k)
  alert NotFound(k) post(value == null)
```

The callee must still provide some way sending this information, either through updating of arguments, return values, global variables or exception objects – alert parameters are merely a declaration of whatever mechanism is used. For exceptions, the exception object is available for use in alert parameters:

```
int f() throws AnException
  alert Error(e.msg) on throw AnException(e);
```

4.3 Alert Handling

The *alert handler* will treat all alerts the same, whether they are reported by return value, condition check, or exception. The grammar for alert handlers is presented in Figure 4. There are two alert handling constructs: **on**, for specifying an alert handler at any scoping level, down to a single statement, and the *handler operator*, **<::**, which specifies a handler for a single expression.

4.3.1 The on Construct

The **on** declaration takes a *alert-pattern* and a statement. The declaration is lexically scoped and applies to all call sites it matches within its block. The statement form of **on** applies to a single statement. If more than one handler matches, the most specific one closest in scope applies, or a compile-time error is given if there is more than one equally suitable handler.

The *alert pattern* specifies for which combination of alerts and callees the handler applies. The handler itself is a single or compound (block) statement, which should provide a replacement value, retry the computation, refer to another handler, or terminate the caller. It is an error for a handler to complete without providing a replacement return value when one is needed – in this case, we terminate the program (though we could check statically whether this can occur, and other design choices are certainly possible).

Within a handler, the use statement may be used to provide a replacement value; use exits from the handler as if the callee had returned normally with the value provided. For example, the following defines a handler for NotFounds in the lookup function which substitutes the value "Doe, Jane" for failed lookups (e.g., when mapping ID numbers to names):

```
on NotFound in lookup() use "Doe, Jane";
```

The following does the same for all alerts in lookup:

```
on Alert in lookup() use "Doe, Jane";
```

The next two declarations both do the same for `NotFound` in all functions (the `%` matches all functions[3]):

```
on NotFound in % use "Doe, Jane";
on NotFound use "Doe, Jane";
```

The following `NotFound` handler applies to just the preceding `print` statement:

```
print(lookup(tbl, key)) on NotFound use "Doe, John";
```

The **`retry`** statement tries the failed call again (possibly with a maximum retry count, specified with "**`max`** *number*" – the default is to retry indefinitely). The **`retry`** statement also takes an optional list of arguments, which will replace the arguments in the failed call. It will exit from the handler, continuing execution at the point of the call which reported the alert – except when the maximum retry count has been reached, in which case execution continues with the next statement after **`retry`**.

For example, the following specifies that on a `NotFoundError`, we should ask the user for a new name and try again (maximum 5 times). If our recovery attempts fail, we substitute an empty string.

```
on NotFoundError in readfile(char *name) {
  warn("trying again...");
  char* name = askUser();
  if(name != NULL) retry(name) max 5;
  warn("giving up...");
  use ""; }
```

4.3.2 The Handler Operator

The handler operator provides a convenient way of handling alerts at the expression level. The left operand is an expression to be evaluated, and the right operand is a handler to be used if an alert was reported in the expression. Within the operator itself, one may specify which alerts should be handled. Alerts are specified in the same way as for **`on`**, the handler can be either an expression giving a replacement value, a call to a previously defined handler, or a statement list (enclosed in braces). In the following example, a string is substituted if a lookup fails:

```
print("result: ", lookup(t,k) <:: "Unknown");
```

An alert pattern can be specified between the colons:

```
print("result: ", lookup(t,k) <:NotFound:  "Unknown");
```

Furthermore, handler code can be provided, as for **`on`**:

```
fd = open(name, flags) <:eNoEnt:  {
    char *newname = askUser();
    if(newname != NULL) retry(newname,flags);
    else giveUp("couldn't open file"); };
```

This will try to open a file, and if the file is not found, the user will be asked for another name. If the user provides one, we try that instead, otherwise, we abort with a message.

4.4 Abstraction

Our extension provides abstractions for alert handling and reporting. The **`handler`** construct declares handlers which may be used later on by the **`on`** declaration or the handler operator. For example,

```
handler log(msg, dflt) {
  print("An error occurred:  ", msg);
  use dflt; }
```

which may be used as:

```
on NotFound in % log("Lookup failed", "");
name = askUser()
  <:eNoEnt:  log("No response from user", "--");
```

Handler abstractions look deceptively like functions in both definition and use, but are not functions, since the **`retry`** and **`use`** statement would be tricky to implement in a separate function. Instead, the definitions are expanded inline wherever they are needed. Hence, (mutually) recursive handlers are not allowed.

The **`alertrepdef`** declaration declares alert reporting mechanisms for use in a function declaration. It follows the same pattern as the C/C++ **`typedef`** construct. This is useful when several functions share the same alert behaviour. For example,

```
alertrepdef alert Error post(value == 0) ErrorOnZero;
```

`ErrorOnZero` can then be used for functions raising errors with a zero return value:

```
int f() alert ErrorOnZero;
```

4.5 Sending Information from Callee to Caller

Alerts can have associated values (alert parameters), allowing a callee to provide additional information to a caller. A similar idea is found in exception handling (e.g., in Java or C++), where exceptions are objects that may contain information relevant to the exceptions. As shown in Section 4.1, valued alerts should be declared with arguments:

```
alert {Error(char *msg)};
```

At the callee side, we provide a suitable value in the **`alert`** clause:

```
int read(int fd, void *buf, size_t count)
  alert Error("read error") post(value == -1);
```

The value can be obtained at the handler side from the alert pattern:

```
on Error(msg) in read() { print(msg); exit(1); };
```

In this example, `msg` is declared as a string, and gets the value `"read error"` from the failed `read()`. If more than one alert is given in the alert pattern, all of them must have the exact same argument list. It is not necessary to mention the arguments if they are not needed by the handler.

If the return value of the callee is to be available to a handler, it must be passed as a parameter, as return values are not always available (e.g., for exceptions and `pre` conditions), and there is no way for the handler to distinguish between different alert reporting mechanisms.

Note that, unlike exceptions, we need not construct an alert object as an aggregate of values. Instead, code is generated in the handler which obtains the information directly (which is why only arguments, return values and global variables can be passed from the reporter). Thus, as for alert conditions, we are restricted to expressions which have the same meaning for both the callee and the caller (i.e., global names and operators, constants and arguments, either before or after the call).

In the case of functions which change their arguments (or modify global data structures), it is possible that the state of these variables is inconsistent when the handler is invoked. In this case, it is up to the handler to put things in a consistent state or terminate execution. Ideally, functions would ensure that the program state is rolled back to a safe point before an alert is reported, or at the very least, declare that this may not happen for some or all alerts. This problem is also found with the common exception handling mechanisms. We have not dealt with this problem yet.

4.6 Granularity and Funspaces

By granularity, we refer to the coarseness of a declaration in the hierarchy from expression through statement, function declaration, and optionally class, module and subsystem level, all the way to the system level. Our language extension provides additional granularity alternatives, among them groups of functions, which

[3] The `%` was chosen to avoid confusion with pointers (*) in C/C++, and is used in a similar fashion in AspectC++ [28].

funspace ::= [return-type | **%**](function-name | **%**)
[**(** parameter-pattern-list **)**]
funspace ::= **funspace** *funspace-name*
funspace ::= **{** *funspace****,** **}**
declaration ::= **funspacedef** *funspace funspace-name* **;**

Figure 5. Grammar for declaration of function spaces.

we call funspaces. Funspaces can be applied to both alert handling and reporting.

4.6.1 Granularity of Handlers

In most languages, exception handlers are specified at the statement level, and the exception declaration (e.g., `throws` in Java) occurs along with the function declaration.

In our system, the declaration of both alert handlers and alert reporters can occur at various levels of granularity. As we have seen, handlers are declared with **on** declarations. These can occur at any level in the scope hierarchy (from global, through namespace/package and class, down to blocks and single statements within a function) and apply to the scope in which they are declared. Additionally, handlers can be declared at the expression level using the handler operator (**<::**). If multiple handlers are in conflict, the most specific handler takes precedence, i.e. the one with the most specific alert pattern at the finest level of granularity.

The concept of a scoping level can be refined using *funspaces*. A **funspace** declares a set of functions, i.e. a subspace of the namespace for function names. The grammar for funspaces and funspace declarations is presented in Figure 5.

A funspace is basically a list of function patterns. For example, (a non-exhaustive list of) the file operations of POSIX can be declared as follows:

```
funspacedef { open(), close(), creat() } posix_io;
```

Each entry in the funspace list conforms to a pattern. For C and languages without overloading, giving a function's name is sufficient. In languages with overloading, the full signature must be given;

```
funspacedef {
    int open(const char *pathname, int flags),
    int open(const char *pathname, int flags, mode_t m),
    int close(int fd),
    int creat(const char *pathname, mode_t m) }
  posix_io;
```

The pattern can also contain wildcards, with `%` matching any single item, and `..` matching any argument list. For example, The pattern `% %(const char*, mode_t)` would match any function with any return value, that takes two parameters: a `const char*` followed by a `mode_t`, e.g. `creat`:

```
int creat(const char *pathname, mode_t mode);
```

Funspaces, being sets of functions, can be merged, allowing us to construct the `posix` funspace from smaller, task-specific funspaces.

```
funspacedef {
    funspace posix_io,
    funspace posix_memory,
    funspace posix_process }
  posix;
```

```
alert {PrecondFailure, Whoops};
on PrecondFailure in * {fatal("Precond failed");}

int f(int x) alert PrecondFailure pre unless(x > 0)
             alert Whoops post(value > 10);
int ff(int a, int b)
{ on Whoops in f() {print("whoops!"); use 0;}
  return f(f(a));
}
```

```
int f(int x);

int ff(int a)
{ int r;
  if(a > 0)
  { r = f(a);
    if(r > 10) { print("whoops!"); r = 0; }
    if(r > 0)
    { r = f(r);
      if(r > 10) { print("whoops!"); r = 0; }
    } else fatal("Precond failed");
  } else fatal("Precond failed");
  return r;
}
```

Figure 6. Comparison of the Alert extension to C (top), and the corresponding normal C code (bottom).

This is not merely a syntactic convenience. Different subsets of a given API often use different sets of errors, each specific to that subset. Sometimes, the same numerical error value is reused with different meaning across different subsets. `ENOMEM` when returned from `mmap` has a different meaning than `ENOMEM` returned from `stat`. Using funspaces, these differences can be captured at the granularity of function groups, rather than having to be specified on per-function basis.

4.6.2 Granularity of Alert Reporting

In the previous sections we saw how alert reporting is declared on individual functions. Using funspaces, reporting mechanisms may conveniently be declared on groups of functions. The following declares that the *eNoMem* alert will be reported on any function in the POSIX funspace if it returns -1 and the global variable `errno` is set to `ENOMEM`.

```
funspace posix alert eNoMem post(value == -1
        && errno == ENOMEM);
```

Both handlers and alerts can be declared at any scoping level and on funspaces, but the declarations are completely independent. For example, the alert *eNoMem* may be specified for all POSIX functions, as above, while a handler for this alert could be declared for only one expression inside a particular function in a given program, e.g.,

```
f() { open("foo",O_RDONLY) <:eNoMem: exit(EXIT_FAILURE);}
```

Or, it could be declared for all POSIX functions:

```
on eNoMem in funspace posix { exit(EXIT_FAILURE); }
```

Multiple, overlapping funspaces may be declared, and both alerts and handlers may be specified independently for each funspace.

4.7 Interfacing with Legacy Code

Introducing new failure handling disciplines typically means that legacy code must be rewritten if it is to take advantage of it. This is the case with exceptions, for instance: if you want exceptions in an existing library which reports errors with return values, you will have to either rewrite the library or write a wrapper for it.

Funspaces, together with handling and reporting abstractions can be used to specify alert reporting mechanisms and handling for a large number of existing functions in a few lines of code. This makes reuse of existing libraries simpler, which is especially im-

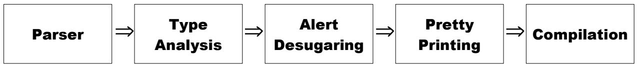

Figure 7. The compiler pipeline for our extended C language.

portant since many older libraries use exotic alert reporting mechanisms which may be inconsistent with newer code.

If the library comes with structured documentation, it may be possible to automatically extract alert specifications from the documentation. For example, the POSIX standard [24] comes with structured manual pages in HTML format, available online. A tool could be written to extract from the manual pages function names, return value style ("returns -1 on error") and which error codes are applicable to the function, and generate the necessary declarations. We are currently exploring this option.

5. Implementation

We have made a prototype implementation [1] of the language extension for C that implements the compiler pipeline shown in Figure 7. The prototype is implemented using the Stratego/XT [5] program transformation language, and the C-Transformers [3] framework for C99 transformation.

The prototype consists of an extension to the C99 grammar [14] (written in the grammar formalism SDF2 [33]) and a set of transformations translating the extended C code to standard C. As seen in Figure 7, the parser will recognise the extended C language and produce an abstract syntax tree (AST). Minimal type analysis is then performed to check that the handlers are type consistent with the functions they will be applied to. Next, the alert extension is "peeled off" in a desugaring step before the the AST is pretty-printed into text and fed to a normal C compiler.

5.1 Translation scheme

The translation algorithm works roughly as follows (for C99, following traversals can be combined into one pass, since the language requires declaration before use):

First, traverse the AST and look at all function declarations and definitions. For each declaration or definition, extract the signature (i.e., name, return type, argument types) and the alert mechanism (i.e. pre/post conditions – exceptions are not available in C), and store this for later.

Next, traverse the AST and look for scopes, function calls, `on` handler declarations or handler operators. When seeing a *scope*, create a new, nested scope for subsequent `on` handler declarations. When later existing this scope, drop all handlers registered in this scope. When seeing an *`on` handler declaration*, register its entire definition in the current scope. When seeing a *handler operator* (`<::`, expand the pattern in Figure 8. When seeing a *function call*, check if the signature of the callee is matched by any of the registered handlers. Check the textually closest handlers first and proceed to parent scopes. If a matching handler is found, expand the pattern in Figure 8.

Keep in mind that a function call is only rewritten if at least one relevant handler is found for it. Let us call the closest (and thus active) handler *current-handler*. Given a function call $f(e_1, ...)$ to function t_0 $f(t_1, ...)$ where t_0 is the return type, f the name, t_1, ... the types of the formal arguments, and e_0, ... the expressions for the actual arguments, the instantiation of the template in Figure 8 occurs as follows: First, a local variable r of type t_0 is declared, and will hold the eventual return value. Then, each of the actual arguments is evaluated and stored, each e_x into a local variable v_x. Then, the expression for the `pre` condition of f is evaluated on the variables v_x (the expression <*precond(v0, ...)*> means that the pre condition code is expanded in-place – care is taken to avoid accidental variable capture), and if it succeeds, we must invoke the alert handler. The code for *current-handler* is expanded in place, and `use` statements in the body are translated into assignments to r. The process is similar for the `post` condition and the `post` condition handler. An instantiation of this template was given in a cleaned up, human-readable form in Figure 6.

```
t0 r;
{ t1 v1 = e1; ...
 if(<precond-f(v1, ...)>) {
  <current-handler>;
 } else {
  r = f(v1, ...);
  if(<postcond-f(r)>) {<current-handler>};
} }
```

Figure 8. Template for desugaring function calls.

5.2 Implementation issues

In C, which lacks function overloading, matching a function call to the function's declaration can be done easily just by comparing names. In other languages, such as C++ and Java, overload analysis is needed to distinguish functions sharing a common name.

Although overload analysis is unnecessary for C, we still need to be able to determine the types of arbitrary expressions, in order to declare temporary variables and type-check substituted values. Support for this is lacking in Transformers, but was easily added.

A special problem occurs with function pointers, since it is in general impossible to determine statically which function will be called at runtime. In the case of dynamic loading, the function may not even be written yet. A similar problem occurs with exceptions in object oriented languages, where the preferred solution, in for example Java, is to require the exceptions of a function in a subclass to be an (improper) subset of the exceptions of the corresponding function in the superclass. This technique translates to our alerts as well: We can add a declaration about alerting to the type declaration of the function pointer, i.e. the function pointer declaration now also declares the alert mechanism for the function it will eventually point to, and type checking on the function signature, with alert declarations, must be performed when function pointers are assigned to. Arguably, this will make function pointers even more difficult to read, but these syntactical issues can be remedied by judicious use of `typedef`s. Our current implementation does not yet support this.

5.3 Compiling to Aspects

The application of alert handlers to function call sites is a separate, cross-cutting concern, and can certainly be considered an aspect in the sense used by Kiczales et al [15]. If we targeted AspectJ rather than C, the template in Figure 8 could be realized as an around advice, where the invocation of f is replaced with a call to `proceed`. The `pre` and `post` condition expressions would be placed before and after the call to `proceed`, respectively, in the same fashion as now. `Use` statements in the handler body would translate into `return`s.

The full granularity of handler declarations from expression level to arbitrary function groups would be harder to capture faithfully, however. While funspaces can be captured by normal point-cuts, by listing all function names in the point-cut, we do not see any easy and robust way of encoding point-cuts that exactly match expression level handlers.

6. Related Work

Language support for exception handling has been introduced gradually since the 1960's. Research-wise, PL/I's condition system and later CLU [18, 17] and Ada's structured exception handling have perhaps been the most influential.

Hill [12] documents about ten different idioms for raising exceptions in languages such as Algol and Fortran, which at the time did not have any exception handling facility. He advocates disciplined control transfer over special return values.

Randell [26], introduces a failure recovery system inspired by "stand-by sparing", as found in hardware designs. Their technique assumes a nested, block-structured language, and provides transaction-like error recovery. Each block is a transaction, and may have one or several recovery blocks associated with it. A block has an acceptance test, which acts as a postcondition check. If a block fails, by failing its postcondition check, the program state is unrolled to the state before the block was entered, and the list of associated recovery blocks is tried in order, each time preceded by an unroll, until one of the recovery blocks passes the acceptance test. If the list is exhausted before recovery occurs, the error is passed to the enclosing block. This technique does not support raising of errors to handlers further up the call chain. MacLaren [20] critiques the design for being too complex, and encouraging bad idioms, like a global **ON** handler for file exceptions which set global error flags that must be checked after by the caller of any file operation, thus effectively degenerating to global error values.

Borgida [4] discusses language features for exception handling with a focus on the interplay between exception handling and transactions found in database and information systems. He advocates the support for resumption, user-defined exception types, classification of exception types, and preventing the handler from modifying the context of the alerter. The language presented supports transactional unrolling in the case of unhandled exceptions and the capture of accountability in transactions using exceptions.

In a sufficiently reflective language, such as Oberon, exception handling may be entirely implemented by the user without extending the language, as is shown in [13]. Oberon allows reflection over stack frames using "riders". Exceptions are thrown by invoking a rider that locates the appropriate handler in an enclosing procedure on the stack. If the found handler returns, this is taken as termination, and the stack below the handler is cleaned. If the handler invokes **Resume**, execution resumes at the point of exception.

Romanovsky and Sandén [27] discuss good and bad practices in exception handling, dividing the problem into bad language design and misuse due to insufficient training. They argue that languages should support two kinds of exceptions with respect to their program units (modules or packages): internal and external. External exceptions must be declared and checked, i.e. a propagation discipline must be declared and the compiler must enforce it. They argue further, based on experience with Ada, that exceptions in OO-languages must be classes, and user-definable, so that information may be passed from the exception raiser to the handler with the exception, and so that the exceptions may be classified based on type. They also argue that exceptions can aid in, rather than complicate program validation and verification. The critique of Romanovsky and Sandén about a propagation discipline was addressed by Luckham and Polak [19]. They describe a language extension to Ada for specifying the propagation of exceptions. This extension has not been included in later versions Ada, however.

Cui and Gannon [7] describe an alternative exception handling system for Ada than the school of Goodenough [10]. Instead of being declared as part of the control structure, as markers on the stack, exception handlers are dynamically attached to objects. When an object raises an exception, its associated handler is invoked. The authors refer to this as data-oriented exception handling. Our alert system provides no easy way to support this form of exception handling.

An argument against exception handling for embedded systems – the difficulty of predicting timing constraints in the face of exceptions – is addressed by Chapman et al [6], where the authors presents a model for static timing analysis of exceptions in a subset of Ada83.

The Lisp condition system [25] is similar to PL/I's **ON**, but supports more of the features first described by Goodenough [10], such as resumption.

Other functional languages, such as SML and Haskell, also support exceptions. Wadler [34] shows how exceptions may be realized in purely functional programming languages, using monads.

Dony [8] describes an object-oriented exception handling for Smalltalk, where users can define new exceptions, where exception objects contain information passed from the alerter, and where different exceptions can be distinguished and organised in a hierarchy based on their type. Like Goodenough's approach, the possible action of the handler are resumption, termination, and retry, but the choice is determined by the type of the exception object, rather than the alerting primitive. Smalltalk-80 allows the declaration of class-handlers: per-class exception handlers. These do not take handlers in the dynamic call context into account, as proposed by Goodenough; this is provided by Dony's extension.

Lippert and Lopes [16] describe how to untangle error handling code from algorithmic code using aspect-oriented programming. The solution is applied to a framework constructed using design by contract. Aspects are used to extract the contract checking code from the rest of the framework. A drawback of the proposed solution is that the contract is declared in documentation comments along with the framework (algorithmic) code, whereas the contract checking code is maintained elsewhere, thus opening up for deviations between the declared contract and the actual contract checking code.

7. Conclusion

The state of the art in alert handling provides reporting mechanisms that are both efficient and expressive. However, with the exception of the aspect-oriented approach to separating out failure behaviour [16], failure handling code is largely rigid. The alert reporting and handling are tangled, and the implementer must always choose a mechanism when implementing a function, but in doing so, also makes an implicit choice about the handling policy.

We have presented a flexible alert language extension that supports decoupling of the reporting and handling mechanisms for exceptional behaviour. The extension user-defined alert handling and reporting at a wide range of granularities. It allows the caller to declare what is normal and what is exceptional, and to declare separately the desired handling policies. This improves reuse of existing libraries and components, as policies can now be specified retroactively by the library user. We have sketched its implementation [1] based on the Transformers program transformation framework. The extension functions as a compiler extension in the form of a pre-processing step to the C compiler. The design space where the alerter and handler are decoupled is largely unexplored. This is unfortunate, since even the modest extensions we have shown to a simple, imperative language with exceptions could improve both the reuse of existing code bases and the clarity of failure handling code. Some work on this topic has been done in the context of aspect-orientation, but we believe that our alert declaration language is more precise and concise than generic join-points and advice. One could consider our language extension as a domain-specific aspect language for error handling.

While our proposed extension has only been realized for two rather simple languages, in the imperative style, we expect it to be transportable to other languages and paradigms. The syntax should be modified to fit the conventions of the language; in this paper, we have based the syntax on C-like languages; this would be out of place in Python, for instance. A few obvious extensions may be necessary. In general, funspace patterns may need to be more like AspectJ [15] or AspectC++ [28] point-cuts, to deal with function overloading and namespaces. In our subject language, C with exceptions, this was not needed, since we only have one global namespace and no overloading. It is important that funspaces should continue to be function (or method) groups, so that funspaces can cross-cut namespaces. This will keep the flexibility of specifying alert mechanisms and handlers across namespaces. More research is necessary to determine the best interaction between funspaces and method visibility, however.

Acknowledgments

This investigation has been carried out with the support of the Research Council of Norway. We thank the anonymous reviewers and May-Lill Sande for their helpful comments.

References

[1] Alert extension for C. URL `http://www.codeboost.org/alert/`.

[2] J. Barnes. *High Integrity Ada. The SPARK Approach*. Addison-Wesley, 1997.

[3] A. Borghi, V. David, and A. Demaille. C-transformers: A framework to write C program transformations. *ACM Crossroads*, 12(3), April 2006.

[4] A. Borgida. Language features for flexible handling of exceptions in information systems. *ACM Trans. Database Syst.*, 10(4):565–603, 1985. ISSN 0362-5915.

[5] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: Components for transformation systems. In F. Tip and J. Hatcliff, editors, *PEPM'06: Workshop on Partial Evaluation and Program Manipulation*. ACM Press, January 2006.

[6] R. Chapman, A. Wellings, and A. Burns. Worst-case timing analysis of exception handling in Ada. In L. Collingbourne, editor, *Ada: Towards Maturity. Proceedings of the 1993 AdaUK conference*, pages 148–164. IOS Press: London, 1993.

[7] Q. Cui and J. Gannon. Data-oriented exception handling. *IEEE Trans. Softw. Eng.*, 18(5):393–401, 1992. ISSN 0098-5589.

[8] C. Dony. Exception handling and object-oriented programming: Towards a synthesis. *SIGPLAN Not.*, 25(10):322–330, 1990. ISSN 0362-1340.

[9] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991. ISSN 0360-0300.

[10] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975. ISSN 0001-0782.

[11] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, USA, 1996. ISBN 0-201-63451-1. xxv + 825 pp. LCCN QA76.73.J38G68 1996.

[12] I. Hill. Faults in function, in Algol and Fortran. *Comp.J.*, 14(3), 1970.

[13] M. Hof, H. Mössenböck, and P. Pirkelbauer. Zero-overhead exeption handling using metaprogramming. In *SOFSEM '97: Proceedings of the 24th Seminar on Current Trends in Theory and Practice of Informatics*, pages 423–431. Springer-Verlag, London, UK, 1997. ISBN 3-540-63774-5.

[14] ISO/IEC JTC1/SC22/WG14. *ISO/IEC 9899: Programming languages – C*, 1999. URL `http://www.open-std.org/JTC1/SC22/WG14/www/standards`.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001: Object-Oriented Programming: 15th European Conference*, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, June 2001. ISBN 3-540-42206-4.

[16] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. *ICSE*, 00:418, 2000. ISBN 1-58113-206-9.

[17] B. Liskov. A history of CLU. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 133–147. ACM Press, New York, NY, USA, 1993. ISBN 0-89791-570-4.

[18] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, 1977. ISSN 0001-0782.

[19] D. C. Luckham and W. Polak. Ada exception handling: an axiomatic approach. *ACM Trans. Program. Lang. Syst.*, 2(2):225–233, 1980. ISSN 0164-0925.

[20] M. D. MacLaren. Exception handling in PL/I. In *Proceedings of an ACM conference on Language design for reliable software*, pages 101–104. ACM Press, 1977.

[21] B. Meyer. *Eiffel: The language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 0-13-247925-7.

[22] B. Meyer. *Object-Oriented Software Construction, 2nd ed.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997. ISBN 0136291554.

[23] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML – Revised*. MIT Press, 1997. ISBN 0262631814.

[24] The Open Group. *The Single UNIX® Specification, Version 3 / IEEE Std 1003.1-2001*, 2004. URL `http://www.unix.org/version3/online.html`.

[25] K. M. Pitman. Condition handling in the Lisp language family. In *Advances in Exception Handling Techniques*, pages 39–59. Springer-Verlag, 2000.

[26] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449. ACM Press, New York, NY, USA, 1975.

[27] A. Romanovsky and B. Sandén. Except for exception handling ... *Ada Lett.*, XXI(3):19–25, 2001. ISSN 1094-3641.

[28] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*. Sydney, Australia, February 2002.

[29] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, third edition, 1997.

[30] S. T. Taft, R. A. Duff, R. Brukardt, and E. Plödereder, editors. *Consolidated Ada Reference Manual. Language and Standard Libraries, International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1*, volume 2219 of *Lecture Notes in Computer Science*. Springer, 2001. ISBN 3-540-43038-5.

[31] H. Truong. Guaranteeing resource bounds for component software. In *FMOODS'05*, volume 3535 of *LNCS*, pages 179–194. Springer, 2005. ISBN 3-540-26181-8.

[32] G. van Rossum and F. L. Drake, Jr. *Python Language Reference Manual*. Network Theory Ltd, Bristol, UK, 2003. ISBN 0-9541617-8-5.

[33] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[34] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. Springer, 1995.