EXCEPTIONAL CONDITIONS AND CONTEXTUAL INFORMATION IN NUMERICAL COMPUTING

G. JUNGMAN

ABSTRACT. Exception handling involves a tradeoff between usage of information local to the subordinate (exception raising) context, and external mechanisms for adjusting the flow of control. In some sense this is well understood; however, as one could easily guess, actual practice lags theoretical understanding by a large margin. I discuss issues related to practical implementation of exception handling, targeted to developers of numerical libraries. In the end, we seem to be left with a mechanism similar to, but more general than an old scheme first implemented in the NAG library. The generalized scheme which I suggest can provide an acceptable level of flexibility, especially from the standpoint of applications in embedded or heterogeneous environments. Two issues are clear. First, no subordinate function should be allowed to make important protocol decisions, such as whether or not to abort when an exceptional condition is detected, without the benefit of information from the calling context. Second, clients should be able to control the behaviour of subordinate functions by adjusting the contextual information which they pass to the subordinate. These basic points are not profound, but it seems that no system yet exists which meets these needs in a flexible way. I discuss a reasonable object-oriented design, with an example given in C++. I also give a rudimentary, and less powerful, C implementation.

1. EXCEPTIONAL DIFFICULTIES

Many readers will be familiar with the Ariane 5 catastrophe, a satellite launch which ended in a half-billion dollar abort due a situation directly caused by software failure [L+96]. If the only lesson learned from this story is that software engineering, and especially software safety engineering, is difficult, then the re-telling would be somewhat empty.

But W. Kahan [Kah98] (amongst others) points out what may be the most important lesson of this failure. Recall that the primary failure occurred in a subsystem which was not critical to the launch. But because of a design decision, the exceptional condition in this subsystem (a conversion overflow)

Date: June 1, 2000.
LA-UR-X.

trapped to a default exception handling mechanism which aborted the guidance subsystem. The important observation is that some exceptional conditions are critical and others are essentially harmless. If the programming environment does not provide appropriate abstractions for making such distinctions, then programmers may be forced or tricked into using default exception handling mechanisms which do not have the contextual information to make important protocol decisions, such as whether or not to call `abort()`, etc.

So this story helps identify one important feature required of exception handling mechanisms; the exception-handling mechanism must have enough contextual information that it can be programmed to make critical protocol decisions in a robust manner. As discussed below, accurate representation of context is the key to good exception-handling architectures.

This issue of protocol control was recognized in its basic form, in the implementation of the NAG library. Ford and Bentley [FB78] point out the need for a distinction between ”hard fail” and ”soft fail”, and how this distinction should be used by library implementation blocks which use other parts of the library. Later work on the NAG C library introduced a more structured interface to this distinction, including the possibility of fine-grained handler registration [Gro]. My experience with the GNU Scientific Library [Tea] was in part a rediscovery of this point, when it became clear that the initial error handling mechanisms in that project were not adequate to support library components which depended on other library components.

Requirements for good exception-handling mechanisms have quite general applicability. However, I have chosen to concentrate on numerical applications because of our interest in numerical implementations, and because such algorithms often manage fairly complicated state and display a wide variety of exceptional conditions. Numerical implementations are also simpler in some sense, because they typically do not deal with varying external state, such as an I/O intensive system or database system might.

In some circumstances, it is fair to argue that numerical applications are exploratory and non-critical in nature. After all, a scientist writing a simulation code for his desktop machine will not be too concerned if the code runs for an hour and then dumps core in an unexpected way. However, the same scientist might become quite animated if a scaled-up version of the same simulation ran for a week on a massively-parallel machine and then dumped core. So it seems that the rudimentary model does not scale. The context for the calculation has changed, but the mechanisms which implement policy have not. In the following I will consider the role of contextual information in robust exception handling.

It is worth pointing out the difference in emphasis here, compared to some of the discussion in the literature. One issue of great importance is access to floating-point implementation details in high-level languages [Hau96]. Many language environments cannot access important details of IEEE-754 [IEE85] conformant floating-point implementations, making it impossible to detect and treat some kinds of exceptional conditions with any degree of flexibility. Fixing this sort of problem requires a uniform language-independent standard for accessing such information. Furthermore, it may be necessary to provide language extensions, such as the ”enable-try-handle” block introduced by Hull et al (see Ref. [Hau96]). However, I will not address this problem (having no solution for it in any case), but instead will address the question of what goals library implementors should have for their designs, bowing to the practical constraints provided by the common languages we use.

Additionally, I am somewhat less concerned with performance issues than with the provision of good tools for developers. Some authors point out that well-designed exception-handling facilities make it easier to write efficient code [Hau96, HFT94]. However, it must be equally important that such facilities can improve maintainability and can make it easier to express exception handling constructs, thereby making them an important part of interface design. After all, the set of exceptional conditions which can occur in a given function is as valid a part of the interface specification as the set of possible results. This was stated clearly by Goodenough [Goo75], and such facility is included as part of the C++ standard, which allows methods to declare exceptions which they can throw. However, *ad-hoc* methodologies in languages without such explicit support are not generally able to support maintainence of this sort of information as part of an interface specification. As a side comment, it would be useful to have tools which allow at least some level of automatic maintenance and verification in this regard, for languages other than C++.

Finally, it should be pointed out that no observation here is likely to be original. For instance, many industrial sectors, such as aerospace and transportation, have a large investment in software safety and robustness, which depends on validation of exception-handling mechanisms as well as rigorous control of implementation algorithms. However, I think it is fair to say that very little of this engineering practice has made its way to the outside world. Furthermore, one can guess that much of the software experience along these lines probably exists in a kind of in-house limbo, practiced by local experts and not ready for wider distribution.

2. PRACTICE, COMMON AND UNCOMMON

It can be argued that common practice for exception-handling in numerical computing is very poor indeed. In some cases this is due to the lack of appropriate language support; in other cases it is due simply to a lack of concern for rigorous control of the design.

There is a broad spectrum of practice and support mechanisms. We have the following caricatures:

LEGACY: Many libraries, especially those with legacy code bases, include functions which indicate an ”error” condition has occurred, or that a less severe ”warning” condition has occurred. Typically these functions take a string argument and/or a numerical code, and print something to some output stream. The ”error” function may further distinguish itself by aborting the program.

HANDLERS: Some libraries allow registration of handler functions for various exceptional conditions, so that an exceptional condition causes the flow of control to transfer to the handler. Often the default handler is `abort()` or an equivalent. The standard C library provides such a mechanism for intercepting signals.

FLAGS: Standard C library functions typically indicate the occurrence of some exceptional condition by setting a global error variable. Clients are supposed to check the error variable after calling such a function. A slightly better version of this has the library functions returning some ”error code”, which again should be checked by the client. Of course, many clients will not check.

IEEE-754: The IEEE-754 floating-point standard recognizes the importance of ”harmless” exceptions by explicitly separating the notion of an exceptional condition from a transfer of control (trap). From the standpoint of exception-handling, I think this is the most important feature of the IEEE-754 specification. However, the notion of trap in IEEE-754 is necessarily quite limited, and in practice it does not seem possible to use the trap mechanism in any non-trivial way.

C++: C++ provides a standard exception-handling mechanism as a language feature. Fundamentally it acts as kind of combination of the Flags and Handlers methods, a combination which is a reasonable compromise representing what most users of the Flags and Handlers methods would have liked to do, given the necessary support. However, by design C++ does not make a distinction between exceptions; there is no direct way to express the notion of a ”harmless” exceptional condition, or any other variations of severity, using the C++ mechanism.

ADVANCED: Some languages support advanced notions of control flow, which could be useful in exception handling implementations. For example, Scheme

provides `call-with-current-continuation`, which is powerful enough to support many control constructs [A+]. This power stems, essentially, from the treatment of computation context (technically known as a `continuation`) as a first-class language element. One can save the state of the computation[1] and return to it later, perhaps on the occurrence of an exceptional condition. Below, I discuss how such an advanced notion might be used in exception handling.

The first three methods display the principal difficulty in exception handling, which is how to gain enough contextual information to make a well-informed decision. Global mechanisms almost always lack such information. Invoking such a mechanism typically involves a transfer of control to an execution context which cannot meaningfully deal with a general exceptional condition. Furthermore, it is difficult to implement any method for passing information to such a global context in a non-cumbersome, or even thread-safe, way. Library implementors have always been aware of this problem, but have generally ignored it because the expenditure of development resources on such issues is a hard sell in a world driven by "results".

Even in a well-reasoned discussion such as that of Ref. [Goo75], the question of how to manage the state associated to an implementation block is unclear. Consider the example

```
CALL F; [X: CALL G; [Y:... ESCAPE Z;]...]
```

given in that reference. Here `G` is a handler registered for exception `X`, which can be raised by `F`. However, there is no indication of arguments for `G`, and it is not at all clear how `F` will be able to communicate information which `G` might need in order to prepare an operation as complicated as "cleanup and resume".

Another example from the same reference appears to address the issue. It is an example of an iterator mechanism, where each successful iteration raises the condition `VALUE`, indicating that a value has been obtained. The example sums the obtained values.

```
SUM = 0;
CALL SCAN(P, V); [VALUE: SUM = SUM + V; RESUME]
```

In this example, the locality of information problem is side-stepped by passing the value back through the argument `V`. The problem is that this example does not scale. How many arguments would be required if the handler block was not summing a list of numbers but actually trying to prepare for resumption of some operation involving significant state?

[1] This is mildly inaccurate, but those readers not familiar with this feature can refer to the references for a precise definition.

There is at least one example of real-world software practice which justifies the concern over such communication. Some C math library extensions provide a handler registration mechanism which allows passage of information regarding the arguments of the function which raised the exceptional condition. For instance, consider a function such as `sin(x)`, which might raise an exception. Typically the user can define a function of the form `matherr(struct math_exception * e)`, where the argument might look something like the following [2].

```
struct math_exception {
  int     error_type;
  char *  func_name;
  double  argument;
  double  return_value;
  int     return_errno;
};
```

Upon raising the exception, the library function would transfer control to `matherr()`, passing it the argument of `sin(x)` and setting `func_name` to "`sin`". Then, in principle, the handler function could execute some code which depended not only on the error flag but on the actual value of the argument which raised the condition, and load a value into `return_value`, which would be returned to the subordinate context when `matherr()` returned control to the `sin()` function. The implementation of `sin()` would, by convention, return the value of `return_value` as its result. Unfortunately, it is not at all clear what useful function the handler could perform with this information, and what values might be appropriate for assignment to `return_value`. Furthermore, this construction obviously does not scale to other types of library functions. For instance, some attempt was made to generalize to functions of more than one variable, but one can see how this would never be satisfactory. In fact, judging by the amount of use it gets, this mechanism seems to have been stillborn; real experience has returned a verdict.

Reconciling the locality of information and the need for information in a non-trivial handler context is perhaps the fundamental problem for exception handling. In some sense such a reconciliation is impossible. The only way to preserve locality of information is to have the implementation block cleanup its own mess, since it is the only natural context which possesses the needed state information. In this case, the clean-up code does not really fall under the umbrella of an exception handling mechanism as such. It is

[2]For details about one such implementation, see the Cygnus GNU Pro Math Library manual. This mechanism seems to have had its origin in the library described by Moshier [Mos89]; see the section on the `mtherr()` function in that reference.

simply some more implementation code, which happens to handle a case which is rare in normal practice. Therefore, from this viewpoint the key problem becomes one of giving the subordinate context enough information to make a reasonable decision, when a decision is required.

One might wonder if this situation provides a tangible example for which advanced control-flow could be used. Imagine the following abstract situation. Procedure `f` calls procedure `g`, and `g` raises an exception, which transfers control to `f`. Now `f` examines the exception and decides on a course of action. One or more possible actions might require manipulating the state of `g`, for example to clean-up some data structure which is managed by `g`. With a standard throw/catch mechanism, such as provided by C++, there is no way to descend the stack after it is unwound by the throw. Therefore, `g` must act without the benefit of information from the calling context `f`, or `f` must be prescient and agree to provide `g` with a complete set of rules for its behaviour. However, one could imagine saving the continuation in effect at the time the exception is raised in `g`, and restoring it in `f`. This creates a kind of non-local communication mechanism between the calling context and the subordinate context.

Nevertheless, how such a mechanism would work in practice, and how it would scale with the size of the state in `f` and `g` is unclear. It is clear that the "prescient" `f` example, required when restoration of the `g` continuation is not supported (such as in C++), does not scale, because the maintainer of `f` and the maintainer of `g` must remain completely synchronized regarding the possible internal behaviours of `g`. The advanced mechanism of continuation restoration appears to loosen this coupling, but the appearance is deceiving because it is not clear what the two contexts have to say to each other, without a complete specification of a communication protocol, including an enumeration of possible internal states. Finally, such mechanisms cannot be expressed at the user level in a language-independent way, given the constraints on typical languages. Therefore, although such mechanisms might be useful for certain implementation details, if expressed in a language like Scheme, they will never be more than a curiosity to the general user community. I will not discuss advanced control-flow support further.

Because of this difficulty with locality of information, and given the constraints of languages in common use, I do not think it is possible to solve the problem of supporting truly nontrivial handling mechanisms as such, meaning mechanisms which make use of information from the subordinate context. However, this leaves open the question of how to support *local control of handler actions*, and this is certainly a solvable problem. By local control, I mean a mechanism which allows context sensitive control of protocol decisions, such as whether to "hard fail" or "soft fail". In some

sense, this is a fall-back to the "prescient" example from the discussion of continuation restoration above, but where the scaling problem is bypassed by encapsulating protocol control and further cataloging a small, fixed set of "actions" which exhaust the possible behaviours of the subordinate function. I will provide examples allowing such support in C and C++ below. In the following section I clarify what is meant by contextual information from the calling context and formalize a protocol for sharing this information.

3. WHO HAS THE CONTEXT?

3.1. **The Local Nature of the Problem.** We will first look at some very simple examples of machine arithmetic, to see the importance of contextual information for exceptions. The emphasis is not on machine arithmetic as such, which is capably handled by the IEEE-754 specification, but on the general structure of such examples.

Consider the following code fragment.

```
double x = foo();
double y = 1.0 + 1.0/x;
```

Suppose that the function `foo()` is guaranteed to return (a machine representation of) a real number in the range $[1, \infty]$. Therefore, it is possible for for `foo()` to overflow its output, but no other exceptional condition will occur. Then clearly the second assignment is always safe (assuming something like an IEEE-754 conformant floating-point implementation, which would set 1/Inf = +0). But if the function `foo()` can sense its own overflow condition and attempt to initiate some form of "error-handling", then the coder of the above fragment might be disappointed. This example is simple, and it illustrates how the IEEE-754 notion of a "harmless" exception can be very useful.

The next code fragment is quite a bit different.

```
double x = foo();
double y = 1.0/x;
```

Now the overflow of x translates to an underflow of y, which is probably not acceptable. Therefore the above fragment must be considered incorrect, `foo()` should be allowed to make known its exceptional result, and code should be inserted to handle that case. As a note in passing, it may interest some readers not familiar with the Ada standard to know that Ada officially ignores underflows. This decision was made to simplify the exception-handling support, under the assumption that underflows are mostly harmless. Of course, one man's underflow may be another man's overflow.

The next code fragment illustrates a somewhat more troublesome but still common problem.

```
double x = ...;
double y = sin(x);
```

The question here is manifestly one of context. What meaning is given to the numerical value of x in this fragment? Assume that `sin()` computes the trigonometric sine function of the machine representation of a real number, returning a result with fixed acceptable precision, as does any good implementation of the sine function. If x "stands for" an exactly represented number, then there is no problem in the above code. However, if x is computed from inputs in some way, then the occurrence of large values of x probably indicates that the algorithm is becoming ill-conditioned. Who will detect this condition, if its detection is desirable? We cannot seriously attempt to detect the condition outside of the `sin()` function, because that would require knowledge of its implementation, breaking its encapsulation.

The problem here is that the function `foo()` in the first examples and the function `sin()` in the last example have no way to sense the context in which they were invoked; therefore, they typically either expect the worst or do nothing at all. Notice that the standard C library implementation of `sin()` makes the latter choice. Neither choice is desirable in general [3].

When the function `foo()` is a library function, and a code fragment like the first example is part of another library function, the designer has a distinct problem. How is the function `foo()` to be implemented?

At this point, one might decide that something like the C++ mechanism solves the problem, because an exception in the function `foo()` will cause a transfer of control to the calling context. Therefore `foo()` should raise the exception, and the client should deal with it as required. However, there are serious problems with this.

Naturalness Cost: A C++ method which throws an exception cannot return a meaningful value. Therefore, even though an exception may be judged harmless by the calling context, there is no way to retrieve a value from a method with a natural prototype; pass-by-reference prototypes would be required.

Cycle Cost: The cost of throwing and catching "harmless" exceptions can be prohibitively high. Remember the design dictum: exceptions should be exceptional, they are not some perverse form of communication protocol. But whether or not a certain condition is considered exceptional is often known only to the calling context.

[3] In a footnote, Muller Ref. [Mul97, p. 179] indicates a desire for an "exactness" flag in future arithmetic standards, to indicate whether a number is to be interpreted as an exact representation or as the result of a previously rounded calculation. Such information would provide the context information to resolve this problem; of course, whether we will ever see such an arithmetic implementation is another question.

Separation: One could imagine a mechanism whereby certain exceptions could be turned off by the calling context. However, this is not the solution to the problem. The thrown exception is the only carrier of information about the possible problem. One sometimes wants to be able to turn off the behaviour ”throw exception”, without erasing the information about the exceptional condition. This is what I mean when I say that there is no such thing as a ”harmless” exception in the C++ standard. Again, the usefulness of the IEEE-754 separation of traps and exceptions becomes clear.

These are not problems with C++ exception-handling support itself. The problem exists at a higher level. The C++ mechanism is by itself not sufficient to solve the problem; however, it could certainly be part of a solution, just as any other fundamentally similar mechanism could.

3.2. **The Global Nature of the Problem.** The examples above illustrate how exception handling can create difficulty at the lowest level of implementation, due to the importance of contextual information. But contextual information can be important at all levels of system engineering.

An obvious area where context is paramount is in embedded systems, in a general sense of the term. Such a system might be anything from a microwave oven controller to a numerical library dynamically linked to a scripting language runtime. Note that the latter situation is becoming quite common in scientific computing, as people try to increase their leverage by combining different language environments and different existing tools, using a scripting language as glue. As Dekker simply states [Dek80], ”Note that an ability to suppress the printing of messages is especially important when routines are called from a program written in another language or operating within a configuration without output channels like in a process control situation.” A library which provides exception-handling according to the LEGACY model is clearly unacceptable for use in a heterogeneous environment, because it has a tendency to ”pull the rug out from under” the other players.

The scientist surprised when his expensive simulation dumps core is even more surprised when he finds the reason was that some poorly implemented canned function decided to pull the rug out from under the whole simulation, even though the exceptional condition itself was demonstrably harmless.

We must consider that one of the goals for a complex software project is to make the system as autonomous as possible, but demonstrably safe. Programs that reliably manage their own state do not have to keep bothering their creators.

3.3. **Formalization.** There are several requirements for communication with a subordinate context. The following elements allow the subordinate context to communicate exception information to the outside.

- an agreed upon set of severity levels which the subordinate context can use to quantify its distress
- an opaque method which the subordinate context can use to raise a distress call; the actual action taken will depend on the protocol encapsulated by the method

One could also consider communication of other information, implementing parametric control of the behaviour of the subordinate context, but I will not consider that here. Some ideas are given in the general discussion section below.

If one thinks of a handling protocol as a mapping from conditions to actions, then the severity levels provide the domain for all such mappings. By expectations in the following, I mean any formal output invariant which could be verified. We have the following minimal set of severity levels.

Severity Levels	Meaning
Fatal	no meaningful result can be produced
Severe	a result will be produced; it will not conform to expectations
Warning	a result will be produced; it may not conform to expectations
Information	a conforming result will be produced; some condition has occurred
None	a conforming result will be produced; normal operation

The necessity of such a prescribed set is clear, when looked at from the standpoint of abstraction. A protocol specifies that a certain action be performed when a certain condition occurs. However, the space of exceptional conditions is generally large and possibly changing (especially during development). Therefore, we must substitute a fixed space as the domain for the mapping which defines a protocol. Furthermore, severity level should not be tied to the specific exception type; it should be decided on a local basis. Therefore, although it appears that the prescription of severity levels is a compromise, it is in fact necessary.

The final ingredient we need is a way for top-level contexts to project control to subordinate contexts at any depth, possibly overriding choices made by intermediate contexts. If the implementation of a library component had final control over the context for its invocation of other library components, there would be no way to selectively trace the activity below one layer of library implementation. This is unacceptable, and a mechanism must be provided to force propagation of user-defined contexts to lower levels. For instance, if the occurrence of a “hidden” exceptional condition is causing an expensive computation to occur at an intermediate level, the

only immediate symptom will be a loss of performance; by exposing layers of exception handling, the application programmer might very quickly see what is happening, without the need to invoke a full-powered profiling tool, or worse, to tediously step through with a line-oriented debugger. Structured information about exceptional conditions should never be permanently buried in an implementation, forcing users to mine it out in unsupported ways.

One might like to have a mechanism for projecting control which was graded by the "depth of the invocation", so that layers of invocation could be peeled away. Specifically, one might like to be able to say something like "expose all subordinate exceptional conditions, but only to a stack depth of 5". However, it is not clear if this would be truly useful. Furthermore, there seems to be no easy way to implement such a scheme in any standard language. Therefore, I will consider only the simplest scheme, which allows an unyielding projection of power to subordinate levels.

Projection of power is implemented by assigning to each context an *override* property. If the current context possesses the override property, it is preferentially passed to subordinates; if not, context handling will occur as normal. The process of deciding which context to pass to subordinates will be called *arbitration*.

4. Making Good Practice Practical

If flexible exception-handling is to become part of the culture of scientific and numerical computing, it must be made easy, both for library implementors and clients. Library implementors' foremost need is for a well-defined model and a working example. Clients' foremost need is for a shield from complexity.

The following two example implementations indicate some of the possibilities. The first example is fully object-oriented, implemented in C++, and in principle provides a very wide range of control. The second example is a minimal C language implementation which allows a fixed range of control for exception-handling based on context.

As mentioned in the formalization section, I have not included secondary communication for other parametric control functions. The design point for these parametric abstractions can actually be different, due to issues like the concern over efficiency. I hope to discuss these in a separate paper discussing advanced designs for numerical libraries.

4.1. **Context Framework.** Since one clearly likes to think of contexts as objects in a framework, it is natural to express them in an explicitly object-oriented way. I choose C++ as an appropriate ”pseudo-code” for this expression, although one could easily imagine substituting other languages with object-oriented support.

```
#include <exception>
#include <string>

// Provide a layer of exception types for
// a given implementation.
class eh_exception : public std::exception
{
  // implementation specific ...
};

// Provide the notion of a context. Includes a
// convenience interface to a logging facility,
// so that developers of derived classes can
// vary this facility independently or make
// use of the default behaviour.
class eh_context
{
public:
  typedef enum { E_INFO, E_WARN, E_SEVERE, E_FATAL } severity_t;

  virtual void
  raise(severity_t s, eh_exception * e) const = 0;

  bool isDominant(void) const;
  void setDominant(bool);

  virtual void
  log(eh_exception *) const;

  // further context sensitive operations ...
};

// Implement arbitration of contexts.
// To be used by library implementation.
inline
const eh_context &
eh_context_arbitrate(const eh_context &,
```

```
                        const eh_context &)
{ /* implementation suppressed ... */ }

// Default user context.
class eh_context_user : public eh_context
{
  // override raise() to define protocol ...
  virtual void
  raise(severity_t s, eh_exception * e) const;
};

// further useful context declarations
// ...
```

The above takes care of that part of the communication with the subordinate context involving raising exceptions. It encapsulates into an object that set of methods which would, in a naive view, involve explicit non-local transfer of control. Thus contexts bundle up a specification of handler protocol. Clearly this is a very general notion, and one could imagine extending the context system to provide access to other semantically "global" information. Some aspects of this general picture are discussed in the section below.

4.2. **Minimal Implementation.** The following code fragments are the bulk of a minimal C language implementation of contextual exception-handling. Note that one could in principle implement a fully object-oriented framework such as the above in C, by suitable gymnastics. However, this explicit example may be useful to some readers. In any case, it helps to illustrate what one gains (and what one leaves behind) when moving to an abstract framework such as the above.

By its nature, the minimal example allows only a fixed set of actions. Therefore, a protocol specification reduces to a table of integers. The allowable actions are given in the following table.

Actions	Meaning
Exit	globally terminate execution
Dump	produce a core dump in an environment which supports such a mechanism
Log	log the occurrence of the exception to a stream
Null	no action

These actions are not mutually exclusive, so they can be combined into compound actions; however, severity levels are by definition mutually exclusive. The "default" map of severity levels to handler actions, which is appropriate in an end-user context, is given by

Severity		Action
Fatal	→	Log + Dump + Exit
Severe	→	Log + Dump + Exit
Warn	→	Log
Info	→	Log

A map appropriate for usage in a library implementation context could map all severity levels to the Null action.

First there is a formalization of the severity levels and handling actions which are available.

```
/* space of condition severities */
typedef enum {
  E_NONE   = 0,
  E_INFO   = 1,
  E_WARN   = 2,
  E_SEVERE = 3,
  E_FATAL  = 4
} eh_severity_t;

/* space of condition dispatch actions */
typedef enum {
  D_EXIT  = 0x01,
  D_DUMP  = 0x02,
  D_LOG   = 0x04,
  D_ABORT = D_EXIT|D_DUMP,
  D_NULL  = 0
} eh_dispatch_t;

/* override states */
typedef enum {
  E_OR_YIELD    = 0,
  E_OR_DOMINATE = 1
} eh_override_t;

/* context object */
typedef struct {
  eh_dispatch_t   _dispatch_table[4]; /* map sever-
ity space to action space */
  eh_override_t   _override;          /* dominance      */
  FILE *          _log_stream;        /* where to log   */
}
```

Next is a set of standard default contexts provided by the system. Top-level users would generally execute in the default user context, though they can fashion any context which they like.

```
/* standard implementation context */
const eh_context_t eh_context_impl =
{
  { D_NULL, D_NULL, D_NULL, D_NULL, D_NULL }, E_OR_YIELD, stderr
};

/* standard user context */
const eh_context_t eh_context_user =
{
  { D_NULL, D_NULL, D_LOG, D_LOG, D_LOG|D_ABORT }, E_OR_YIELD, st
};

/* nosey user context */
const eh_context_t eh_context_user_nosey =
{
  { D_NULL, D_LOG, D_LOG, D_LOG, D_LOG }, E_OR_DOMINATE, stderr
};

/* paranoid user context */
const eh_context_t eh_context_user_paranoid =
{
  { D_NULL, D_LOG, D_LOG|D_ABORT, D_LOG|D_ABORT, D_LOG|D_ABORT },
};
```

Also provided are several macros, mostly for the benefit of the implementation. One allows easy expression of arbitration. Another is the basic invocation to raise an exception in a given context.

```
#define EH_ARBITRATE(context_ptr, context_parent_ptr)  \
  (context_parent_ptr->_override ?                     \
   context_parent_ptr : context_ptr)
#define EH_RAISE(severity, code, message, context)     \
  ... implementation suppressed ...
```

5. The Bigger Picture

I have concentrated on the difficulties with exception handling in scientific codes, arguing that subordinate threads of execution should be able to import information from their calling contexts, in particular an encapsulation of protocol. In this way, they might have enough information to make

important decisions, such as whether or not to raise an exception, and what handling method to invoke when an exception is raised. However, the role of contextual information could be seen as even more fundamental. Simply stated, context should control the behaviour of algorithms.

As an example, consider the problem of instrumenting codes. In any compute-intensive environment, it is fundamentally important to instrument codes in order to understand the performance of algorithms in real situations. Instrumentation, if the barrier to use is not too high, can also be used during development. Ideally, the instrumentation should be as non-intrusive as possible. More precisely, the concern is that the instrumentation be non-intrusive from the code development point of view, i.e. that the possibility for instrumentation not require the explicit awareness of the developer. Profiling tools realize this goal in the simplest sense, because one need only recompile the code with profiling turned on to create an instrumented version of the code. Of course, it can be hard to create a build environment where one has any selectivity in this process.

One could imagine a different, object-oriented, view of instrumentation. In this view, an instrument is a kind of active contextual object, which interacts with subsytems simply by being the context for their execution. These contexts could be managed in real-time, by suitably enlightened debuggers or other runtime environments. If performance was an absolute essential, the instrumentation objects could be implemented at compile-time using static polymorphism (templates in C++).

As another example, consider controlling the precision of a computation. This is a simple form of general algorithm control. Fine-grained control of precision could be a useful optimization for some algorithms, if situations where less precision is required can be detected.

6. CONCLUSION

In one sense, there is no limit to the amount of contextual information that could be made available to a subordinate function, since one could imagine adding requirements and function arguments without limit. Clearly this is not desirable. Furthermore, it seems impossible to use specific information from a subordinate function in any useful way.

Instead I have advocated a simple framework for encapsulating policy from the calling context and making it available to subordinates. This suggestion is the natural evolution of the “hard fail”/“soft fail” distinction first introduced in the implementation of the NAG library, extended to encapsulate policy in a general way. This seems to be the required compromise which leads to maintainable but flexible exception handling.

Finally, it might be interesting to consider some anecdotal arguments for adopting such a system broadly in the scientific computing community. I believe that, as systems become more complex and heterogeneous, the need for more fine-grained control of exception handling will become great. Consider an explicit example of the level of complexity in some scientific computing projects. The Stanford University Center for Integrated Turbulence Studies [fITS], funded as part of the DOE ASCI alliance program, seeks to model a complete jet engine, coupling codes for reacting flow simulation, turbomachinery simulation, flow-mechanical interaction, etc., creating a front-to-back simulation of the internal flow in the engine. Code for various subsystems will be produced by separate teams, some project members devoting full-time effort to code integration. Production runs for such simulations will have no limit to their hunger for computing resources, running for months on the most powerful (and finicky) computing platforms available. Projects of this scale could face difficulties in integration without very precise control over global design issues, such as exception handling.

By concentrating on the fundamentals of numerical computing, such as exception handling, access to the full floating-point capabilities of the hardware, control of precision, frameworks for control of algorithms, and other basic issues, we might hope to serve the greater portion of the community, obtaining near-optimal return for our investment.

7. Acknowledgements

I think J. Amundson and B. Gough for many useful discussions. Finally, because no actual system yet implements the proposed mechanism, it should be considered a work in progress, and all comments regarding its details are welcome.

References

[A$^+$] H. Abelson et al., *Revised(5) report on the algorithmic language scheme*.

[Dek80] T. Dekker, *Design of languages for numerical algorithms*, Production and Assessment of Numerical Software, 1980, p. 291.

[FB78] B. Ford and J. Bentley, *A library design for all parties*, Numerical Software - Needs and Availability, 1978, p. 3.

[fITS] Stanford Center for Integrated Turbulence Studies, http://ctr-sgi1.stanford.edu/CITS/.

[Goo75] J. Goodenough, *Exception handling: Issues and a proposed notation*, Comm. ACM **18** (1975), 683.

[Gro] The Numerical Algorithms Group, *Nag c library: Mark 5*, http://www.nag.com/local/manuals/numeric/c/Manual/CL05_index.html.

[Hau96] J. Hauser, *Handling floating-point exceptions in numeric programs*, ACM Trans. Prog. Lang. Sys. **18** (1996), 139.

[HFT94] T. Hull, T. Fairgrieve, and P. Tang, *Implementing complex elementary functions using exception handling*, ACM Trans. Math. Soft. **20** (1994), 215.

[IEE85] IEEE, *Ieee standard for binary floating-point arithmetic*, Institute for Electrical and Electronics Engineers, New York, 1985.

[Kah98] W. Kahan, *How java's floating-point hurts everyone everywhere*, http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf.

[L+96] J. Lions et al., *Ariane 5 flight 501 failure: Report by the inquiry board*, http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.

[Mos89] S. Moshier, *Methods and programs for mathematical functions*, Ellis Horwood, 1989.

[Mul97] J. Muller, *Elementary functions: Algorithms and implementation*, Birkhauser, 1997.

[Tea] The GSL Team, http://sourceware.cygnus.com/gsl.

LOS ALAMOS NATIONAL LABORATORY, T-8, LOS ALAMOS, NM 87545
E-mail address: jungman@lanl.gov