Exception Handling in Object-Oriented Systems

Report on ECOOP '91 Workshop W4

Christophe Dony[‡] **Jan Purchase**[†] **Russel Winder**[†]

[‡] Rank-Xerox France and LITP
Universite Paris VI
4 place Jussieu, 75005 Paris
France

[†] Department of Computer Science
University College London
Gower Street, London WC1E 6BT
United Kingdom

1 Introduction

The 1970s saw the development of exception handling systems dedicated to procedural programming (Goodenough, 1975; Ichbiah, 1979; Liskov & Snyder, 1979; Mitchell, Maybury & Sweet, 1979). Such tools and the associated mechanisms were designed to enhance software reliability, reusability, readability and debugging. The 1980s saw the emergence of object-oriented technology which brought to programmers a new way of thinking about and designing their programs, as well as some new tools to make the programs more modular and reusable.

Today, object-oriented language designers and users are showing a renewed interest in exception handling. Exception handling systems have recently been, or are being, integrated into many object-oriented languages (C++ (Koenig & Stroustrup, 1990), CommonLisp(+CLOS) (Pitman, 1988), Eiffel (Meyer, 1988), Smalltalk(ParcPlace Systems, 1989) etc.). There are two main reasons for this interest. Firstly, as object-oriented languages become more sophisticated, the problem of coping with exceptional situations occurring at run-time becomes more complex and the need for appropriate tools and language mechanisms to detect, handle and correct errors more crucial (Purchase & Winder, 1991). Secondly, the infeasibility of achieving information hiding and modularity at a large scale without exception handling systems.

The goal of the workshop, organized by the authors, was to compare existing systems developed for object-oriented languages and to address various issues connected with the semantics and the implementation of these systems. The call for participation for the workshop stated many of these issues, the main ones being:

- The relationship between exception handling, software quality, modularity and reusability.
- The problem of object consistency in the presence of exceptional events.
- The level of modularity used to associate handlers with code.
- The role of formal specifications.
- The use of reflection.
- The problems raised by concurrency.

Some other issues were raised during the workshop:

- Should the exception handling system be implemented using the host language or should the exception handling system be an integral primitive of the language? To paraphrase: should exception handling be an integral part of the language or should it be implemented on top of the language?
- Should exception handling be separate from, or be in-line with, the main code?

A number of different exception handling systems were presented during the morning, followed by discussion in the afternoon. The presented systems were sufficiently different to form an interesting overview of various techniques associated with various languages. The Smalltalk exception handling system (ParcPlace Systems, 1989) is dedicated to a dynamically typed language with strong requirements for rapid prototyping. The Eiffel mechanism has been designed for a strongly typed language with strong requirements for reliability and robustness. The system designed for BETA is also dedicated to a strongly typed language, but it is built using an original technique initially applied to block-structured systems. The other systems presented, those for Solve, P-Sather, the University of Washington distributed system and ANSA-DPL, put the emphasis on numerous issues connected with concurrency.

In the first part of this paper we give an overview of these presentations. In the second part, we summarize the main results of the discussion. It is important to realize that many issues were not fully discussed and others were left unresolved. Nonetheless we feel it important to introduce these issues, as well as the more fully discussed ones, as it was agreed they will become more important in the future when the more immediate problems are resolved. To start the discussion however, we present a short state of the art summary to ensure that the reader appreciates the historical background of the field.

2 Exceptions: The State of the Art

2.1 Systems for Procedural Languages

The specifications of existing exception handling systems for procedural languages have all been influenced by Goodenough's seminal paper (Goodenough, 1975). Well known implementations include MESA (Mitchell, Maybury & Sweet, 1979), CLU (Liskov & Snyder, 1979) and ADA (Ichbiah, 1979). These systems have introduced the following ideas:

- An exception handling system should provide infrastructure allowing programmers to establish communication between a function or procedure which detects an exceptional condition while performing an operation, and those functions or procedures that are clients of this operation and wish to handle the error. Exceptions thus allow users to associate handlers with statements or expressions, the association taking effect when the statement is evaluated. Handlers are functions or procedures able to access the environment in which they are defined.
- It is possible to have a stack of handlers, a handler associated with an inner statement can hide a handler associated with an outer one. When an exception is signaled, the execution stack is searched from innermost to outermost stack frame for a handler.
- When invoked, handlers are executed in the lexical environment of their creation, they also are able to receive arguments from the signaller.
- Different functionalities are offered by various systems to restore an errant system to a coherent state. Handlers can either:
  1. transfer control to the statement following the signalling one (resumption), or
  2. discard the context between the signaling statement and the one to which the handler is attached (termination), or
  3. signal a new exception.
- An important feature for readability and information hiding is the ability to declare potential exceptional results in the public interface of procedures and the opportunity to trap all possible exceptions with a single handler.

2.2 Systems for Object-Oriented Languages

Most exception handling systems developed for object-oriented languages (Dony, 1988; Knudsen, 1987; Koenig & Stroustrup, 1990; Lacourte, 1991; Purchase & Winder, 1990) except that developed for Smalltalk, inherit the characteristics described above. In Smalltalk, handlers can only be associated with classes, and, when an exception is raised, handlers are searched in the class where the method is defined. Advantages and drawbacks of this approach have been discussed in (Dony, 1990). The main drawback being that exceptions are not propagated to callers of the signaling method, making it difficult to ensure information hiding. Several systems (Dony, 1988; Purchase & Winder, 1990) combine the classical scheme and the Smalltalk association scheme. That is, they allow handlers to be defined at the class level but they take the invocation history into account while looking for handlers after an exception occurs.

The data structure design of many systems (Dony, 1990; Koenig & Stroustrup, 1990; ParcPlace Systems, 1989) has been influenced, to a greater or lesser extent, by the Flavors system (Moon & Weinreb, 1983) in which exceptions are classes. An instance of one of these classes is created when an exception is raised. All handlers for that exception receive this instance as a single argument, it holds in its slots the information about the exceptional event. The CLOS exception handling system (Pitman, 1988) is a modern system, directly inspired by that of Flavors. The advantages of this representation are:

- The ability to organize exceptions into an inheritance hierarchy reflecting the possible sharing of structures and behavior.
- The ability to trap any subset of exceptions with a single handler.
- The ability to define handling primitives on exception classes and to invoke them via message sending to the instance argument, making it impossible to perform an inappropriate action for a given exception.
- The simplification of the signaling process (as shown in (Dony, 1990)), due to the distribution of handling primitives on various abstract exception classes which restricts the number of signaling primitives.

3 The Presentations

The first part of the workshop was devoted to a number of presentations. Two of these (Deutsch and Nerson) were solicited by the organizers, the others were selected by the organizers from the position papers received. The criterion used was essentially: "Would a presentation of the attendee's position and work act as a basis for useful discussion." This was clearly a subjective decision by the organizers, but we could see no sensible alternative. We believe that no attendee suffered, in the final analysis, from not being able to put forward their views and that the selection worked to generate constructive debate.

We offer here our (subjective) summary of the presentations in the order given at the workshop. The later ones are somewhat less full since many of the issues had been raised in the earlier presentations. We have not had the chance to circulate this to all the attendees prior to publication, thus, there may be factual errors (hopefully not). If there are, then we are confident they will come to light very quickly.

3.1 Jan Purchase & Russel Winder — Exceptions in Solve

The Solve programming language (Roberts, Wei & Winder, 1990) is an object-oriented language designed to harness parallelism. It was developed as part of an ESPRIT project (1588 SPAN) to act as a framework for integrating heterogeneous subsystems on a parallel machine. Although the language was designed to allow separate object specification and implementation, it initially had no support for exceptions and was deficient in its support for debugging. To correct these faults, work was undertaken to introduce exceptions both to handle exceptional events and to integrate computation monitoring and debugging.

In Solve, each object is an instance of a type, with a communications process responsible for receiving messages, binding them to methods and initiating method processes; Solve supports multiple method processes for each object, running concurrently. Communication is achieved via a mail-drop transport mechanism supporting object migration. The reader is referred to the papers cited above for details, the crucial point here is that, despite the significant concurrency available, each object has a single process responsible for message receipt and, hence, for method binding; message reception and semantic interpretation is thus serialized.

Solve now supports preconditions and postconditions associated with each method of a type. This is similar to and inspired by Eiffel (Meyer, 1988). Solve, however, goes further: Domain conditions and event trace conditions can be associated with each type and, hence, each object in the system. Assertion inheritance is supported with a semantics somewhat different from that of Eiffel. Unlike many languages, exceptions are not raised explicitly in code; exceptions are not a general-purpose programming tool to handle asynchronous events but are only available to handle exceptional events. Exceptions and their handling are declared and assertions made regarding the event sequences, states and methods. Exceptions may or may not be a run-time consequence of this declaration. These features permit a very flexible specification of object behaviour, any violation of which is an exception.

Assertion checking and exception handling are controlled by the binding process (dispatcher). On message reception the following takes place: the current object state is temporarily saved then the domain assertions, trace assertions and preconditions for the pending method checked. If these assertions are satisfied, the method is executed. The postcondition and domain assertions are then checked. If these are satisfied, the state is updated (by updating only the amended values) and any result returned. If at any stage an assertion fails or there is a exception created in any message send during the execution of the method, a named exception is raised. Exception handlers are associated with the possible exceptions in the implementation of the object. Possible actions are:

1. Repair the state and retry.
2. Terminate the program.
3. Signal the message sender (raise a new, named exception in the parent),
4. Spawn a debugging process.

As befits the object-oriented approach, only local state repairing is permitted within handlers.

The specification of the type of an object gives a list of the potential exceptions that may be signalled by an object of that type. Users of an object are expected to be able to handle the full set of exceptions, either to actively handle them or to pass them on to their client, with a transformation of name to make it more meaningful at that level of abstraction.

Below are fragments of the specification and implementation for a generic stack. The example is explicitly over-asserted in order to demonstrate most of Solve's features.

```
Signature Stack(SomeType)
SuperTypes (Object)
InstanceOperations
    Monitor Stk
   push : (<SomeType>) -> <selfType>
       postcondition (pushFailed)
   pop : () -> <SomeType>
   top : () -> <SomeType>
       precondition (stackEmpty)
   isempty : () -> <Boolean>
    EndMonitor
DomainConditions
    infeasibleSize
TemporalProtocol
    nonEmptyStack
   satisfies tr
   inwhich $a
   iff $a!send(pop) <= $a!send(push)
End
```

In the implementations of the pre- and post-conditions, primed variables refer to the after state, the unprimed variables to the before state. Note: The exception is signalled only if the assertion evaluates false. For brevity, we show only the implementation details concerning the method **push**.

```
Implementation Stack(SomeType)
Includes (Object)
InstanceSection
    Local storage <List(SomeType)>

    Export Const push <Method((SomeType), selfType)> :=
    [ <SomeType> item |
   storage<--atInsert(1, item) ;
   => self
    ]
   postcondition (
       badAddition [ self'<--top()<--eq(item) ]
       wrongSize [ storage'<--length()<--subtract(1)<--eq(storage<--length()) ]
   )

    .
    .
    .

LinkSection
    Handles precondition::badAddition With pushError
    Handles postcondition::wrongSize With internalError

HandlerSection
    Local Const pushError <Method((), Void)> :=
    [
   self<--dispatcher~retry(2) ;
   self<--dispatcher~toclient(pushFailed)
    ]
    Local Const internalError <Method((), Void)> :=
    [
   self<--dispatcher~debug()
    ]

DomainSection
    infeasibleSize [ storage<--length()<--ge(0) ]
End
```

This model is based on the work reported in (Purchase & Winder, 1990) to which the reader is referred for further detail about the syntax and semantics of behavioural specification. We should note that, as work progresses, changes in Solve syntax have had to be made for consistency reasons. A publication is forthcoming detailing the final syntax (Winder et al., 1991).

The emphasis of the work is to integrate in a single mechanism all types of exception (software and hardware); to provide a more formal basis for specifying object behaviours; and to provide for the harnessing of parallelism. Consequently, this leads to easier debugging and better robustness of software.

3.2 Peter Deutsch — Exceptions in Smalltalk

Peter Deutsch was invited to present an overview of the exception handling system of ParcPlace Smalltalk, introduced in version 2.4 of the language and slightly modified in versions 2.5 and 4. This new system replaces the previous Smalltalk system by introducing a stack-oriented discipline. Handlers can be associated with any expression, they cannot be defined at the class level, and after an exception the execution stack is searched for a valid handler. Handlers can resume, terminate or signal a new exception but it is also possible to specify at signaling time that resumption will be impossible.

Two classes comprise the system. The class `Signal` represents exceptional conditions. Exceptions are signaled and handlers are defined by sending messages to instances of this class. An example of signaling:

```
aSignal := Signal new.
aSignal raiseWith: anArgument
```

It is possible to determine at the point where a `Signal` is defined whether the exception will be resumable or not. Several primitives are defined to signal exceptions. It is an error to raise an exception that requests resumption when the associated signal as been defined as not resumable.

The following is an example of defining a handler:

```
aSignal handle: [:arg | handler-body] do: [protected-expression]
```

Instances of class signal can be linked together with a specific link named `parent`, this allows the trapping, with a single handler, of a signal and all its children. A handler is a block (a lexical closure) accepting one argument passed at handler invocation time. The argument will be an instance of the class `Exception`, which represents an occurrence of an exceptional condition and holds information in its slots, for example the argument passed to the method `raiseWith:`, to enable execution. Handling is performed by sending messages to this argument. Thus, methods performing resumption, termination and so on are defined on the class `Exception`.

The following is an example of a handler response:

```
FileDirectory class methodsFor: 'utilities'

isDirectory: fileName
    "Answer true if the file is a directory."

    ^Filename errorReporter inaccessableSignal
   handle: [:ex | ex returnWith: false]
   do: [fileName asFilename isDirectory]
```

This data structure organization, with two classes, has been compared with the Flavors system. Deutsch admitted that disconnecting the signal hierarchy, composed of "parent" links, from the class inheritance hierarchy and putting handling protocols in a separate class `Exception` leads to a less expressive system. For example, there is no place to store a method dedicated to the handler response for a particular signal, e.g., a redefinition of the method `returnWith` providing additional handling before the call of the primitive one. The reason for doing so was to avoid the creation of too many new classes which are computationally expensive objects in Smalltalk (have global names, are managed by the environment, etc.). The system is open and there is nothing to prevent a user from changing the default handling behaviour of exception objects or the signal objects of which they are arguments.

Experiences with using this mechanism were offered. Exceptions are used in all the parts of the implementation of Smalltalk, e.g., for controlling the correct termination of processes, for controlling underlying operating systems errors (the underlying system exceptions are trapped and converted in Smalltalk exceptions), for controlling the user actions in the debugger, for ensuring the integrity of managers like the emergency evaluator that run externally provided code, for implementing catch-throw behavior in various places, etc. An interesting conclusion is that, according to Deutsch, allowing resumable signals was not a good idea. There are only three places in the system where they are used and all of them could have been avoided by defining different handlers and by using the "retry" instead of the "resume" option. Furthermore, resumption is expensive since it requires the creation of a lexical closure each time a handler is defined.

Another major point he made concerns "unwind protection". In systems in which both resumption and termination are allowed, cleanup actions cannot be defined within handlers since the handler does not know, when propagating an exception, whether the original method will be restarted or not. A specific primitive, externally disconnected from the exception handling system, allows users to specify cleanup actions to be executed at the end of an operation. This primitive is mainly used in Smalltalk to ensure that external resources (e.g., files and semaphores) are released after execution. The problem is that the execution stack needs to be protected before any resource allocation is done, otherwise a window of vulnerability exists between allocating resources at the beginning of a new operation and protecting the stack, during which an exception could leave resources un-released. Although this window can be made very small, the problem cannot be totally overcome without altering Smalltalk significantly.

The last point concerned the implementation. The first was done entirely in Smalltalk and used the Smalltalk ability to represent, when needed, its stack frames as first class objects, directly accessible from the language. However this turned out to be quite expensive since each time an exception was raised it was necessary to reify stack frames. In the new implementation, modifications have been made so that frames in which handlers could be defined are recognized by the virtual machine.

3.3 Jean-Marc Nerson — Exceptions in Eiffel

The Eiffel exception handling system was introduced into the language approximately three years ago. Exception handling was seen as essential to Eiffel since without it, a program had to terminate if a contract could not be satisfied. This was deemed to be unsatisfactory. The following quote from Nerson's talk introduces the Eiffel philosophy:

> *Exceptions in Eiffel are treated not as a control structure but as a technique for dealing with abnormal situations when some unexpected event prevents a routine from fulfilling its specification. The mechanism is based on the Eiffel constructs for assertions; the fundamental idea is "programming by contract", and exceptions are needed when some contract is broken. Exceptions in Eiffel are raised when one of the following situations is detected:*
>
> - *Violation of an assertion, when monitored.*
> - *Failure of a called routine.*
> - *Access to a non-existent object, as in* `x.f` *where* `x` *is a void reference.*
> - *A hardware or operating system exception occurs, indicating some abnormal event (numerical overflow, user interrupt, I/O error etc.) during the execution of the routine.*

The notion of contract is defined by the pre- and post-conditions and by the invariants of the routine and of the class in which the routine is defined. The originality of the system lies in the connections between exceptions and the assertions controlling the contract; exceptions are raised when:

1. A pre- or post-condition is violated.
2. It is impossible for a routine to do anything other than propagate an exception to its caller.

When an exception occurs during the execution of a routine, the rescue clause of the routine (if it exists) is executed, otherwise the routine is said to fail and the exception is propagated to the caller. Rescue clauses are the Eiffel version of handlers, they can be associated only with the whole routine body. They can also be defined at the class level. Within the rescue clause, two techniques are available to the programmer:

1. Retry the execution of the whole routine body after some modifications to the state has been made. Retrying without any modification would clearly cause an infinite loop, but the compiler does not check for this. Retrying is achieved by explicit use of the `retry` keyword within the rescue clause. We must at this point highlight a problem with the terminology: Nerson used the word resumption to describe the retry operation, whereas resumption is more usually used to mean that the handler does not discard the execution stack enabling execution to restart immediately after the operation that raised the original exception. As we will see later, some kind of resumption (with the usual meaning) is possible in Eiffel.
2. "Organized panic", bring all affected objects to a coherent state without using the retry option. This will automatically trigger an exception in the caller, which will recursively have to decide what to do in response to this exception, having the same two possible choices.

Here is an example taken from Nerson's presentation.

```
quasi_inverse (x: REAL): REAL is
    local
   already_tried: BOOLEAN
    do
   if not already_tried then Result := 1/x else Result := 0
   end
    rescue
   already_tried := true;
   retry
    end -- quasi_inverse
```

Markku Sakkinen suggested that it would be simpler to write `Result = 1/x` in the do clause and `Result := 0` in the rescue clause. That is, why not allow rescue clauses to end the routine by returning a result, provided that the relevant assertions are re-checked after the rescue clause? Nerson responded that the Eiffel philosophy was to enforce a programming style in which the semantics of the operation implemented by the routine can be found in its body, rescue clauses being kept as small as possible. Hence when Eiffel supports algorithmic redundancy, all alternatives are located in the same method body.

The Eiffel system as described above seems simple to understand and to use, but its expressive power is quite limited compared to systems in which handlers receive an exception object as an argument. For those users wanting to write more sophisticated handlers in the rescue clause, Nerson presented the class "Exception" which provides a set of features to be used in rescue clauses. This library allows users to identify the current exception, to provide different responses for different exceptions (recall that the rescue clause traps all exceptions), to know in which routine the exception occurred, etc. Unfortunately, this information is propagated and can violation encapsulation (see later). In order that such features be accessible in the rescue clause of a method, the class in which the method is defined has to inherit from the class Exception which some considered to be counter-intuitive.

3.4 Jørgen Knudsen & Ole Madsen — Exceptions in the Mjølner BETA System

Jørgen Knudsen presented the exception handling system used in the BETA language. The language has no formal support for exception handling, instead a language convention, using existing syntax, is adopted. Exceptions are represented as BETA "virtual patterns", a variant of the pattern describing the construction of classes or individual objects. It is possible to associate handlers at the class, method, instruction and individual-object level. The mechanism is based on a static approach, i.e., there is no run-time search mechanism to find handlers. The BETA system allows termination, a variant of propagation, and resumption. The latter means that handlers have to be executed on top of the stack, which in turns means that the handler should not destroy any part of it. In classical systems allowing resumption, when an exception is raised, the handler is searched in the stack without destroying it and is invoked on top of the stack. In the BETA system, handlers are explicitly passed as arguments to the protected routine when the exception is raised. We focus on several aspects of this system which differ from classical designs. As BETA's syntax and vocabulary are somewhat idiosyncratic, we will try to use more common-place terms whilst discussing these issues.

Firstly, the static approach, with no language primitives dedicated to exception handling, requires a usage convention of the language features. It is not clear how this affects readability of BETA code or the discipline of the exception handling mechanism. Handlers can be defined at the program level (programs can be divided into sub-programs in order to have different handlers for the same exception in different part of the program), at the class level, at the object level and at the message-sending level. Defining a handler at the message-sending level means explicitly passing the handler as an argument of the message. In all cases, handlers are executable objects (called exception patterns), and are referenced by the name of an exception (e.g., Overflow). Signalling an exception amounts to directly calling the handler by its name, doing so will invoke the first visible exception pattern of that name, that is: the handler passed as an argument; the handler associated with the receiver of the current method; the handler associated with the receiver's class (or one of its super-classes); or the handler associated with the program (or sub-program). In this hierarchy of handlers, each one can pass control to the next higher-level one. BETA's designers call this propagation, but the reader should beware of the subtle difference between this propagation and propagation through the invocation chain (see discussion below). If we exclude the ability to pass handlers as arguments when calling a service, this system resembles (although it is significantly less powerful) the original Smalltalk exception handling system (see the introduction and the report of Deutsch's presentation). In both systems, handlers are searched in the lexically or statically visible part of the program (if we consider that within a method, the class of the receiver and its super-classes are lexically visible).

The BETA system's exception handling mechanism has several advantages, e.g., no addition to the language has to be made to support it. However, since handlers are statically bound to a context, on occasion the programmer has to manually check whether a handler name has a further binding in a subpattern. Also, when a caller does not pass a handler as argument when invoking a service, the exception will be handled locally by the handler defined on the receiver or on its class and will never be resignalled to the caller. It is debatable whether this restricts reusability (see the discussion). It does mean that blind propagation, as defined in the glossary, is impossible. That is, it is impossible to go down in the invocation chain to find a handler which is aware of the semantics of the initial call that finally led to the exception and which is able to handle it with more specificity than the general statically defined handlers could.

3.5 Heinz Schmidt — Exception in pSather

Sather is a (sequential) language derived from Eiffel. pSather is an extension that introduces parallelism and exception handling. Deferred assignment (futures), low level synchronization primitives and multiple threads, synchronized by monitor objects, are the language constructs used to control concurrency. Like Solve, the exception handling system is designed to enhance robustness and debuggability. Hardware and software exceptions are treated identically and handling may be by termination or resumption. Exceptions are raised explicitly using a catch–throw mechanism which, like that of C++, uses the type of the argument thrown as a search criterion for the handler. The mechanism is defined in the language using a series of low level primitives, the default system being inherited from abstract classes. pSather emphasizes the physical separation of normal and exception handling code.

One major issue addressed by pSather is what action to take in the case of a terminating thread. Since all thread manipulation is done via monitor objects, they alone "see" all the concurrent threads. If one of these threads terminates propagating an exception, the monitor *breaks*. Thereafter, any thread touching it is terminated by an exception and termination of the monitor object handles the error. Although exceptions are not first class objects in pSather (whereas they are in MultiLisp), the solution to the threads problem employed a similar solution to Deutsch's implementation of the MultiLisp style futures in Smalltalk.

3.6 Ashutosh Tiwary & Henry Levy — Exceptions in the University of Washington Distributed System

Tiwary introduced their problem as being the development of an exception handling mechanism for distributed systems that retains all the semantics of classical sequential exception handling systems. Since such systems involve subsystems implemented in different languages running concurrently, the requirement is for a policy rather than an implementation. He claimed that exception handling issues are orthogonal to language issues, that a given exception handling policy and architecture can be implemented in any language. Issues of stack unwinding appear to present some of the biggest problems. Tiwary argued against restricting the scope of handler to the thread in which it was raised or requiring parent and child processes to synchronize for the exception delivery. He maintained that exceptions should be allowed to travel asynchronously across thread and processor boundaries.

Tiwary explicitly avoided issues of implementation, emphasizing the design issues such as where to hold the exception handler environment, how to deal with synchronous versus asynchronous events and handlers etc. He identified the difficulty with exception handling in parallel as finding the context of error; it is frequently not preserved at a process fork. Insisting on synchronization to deliver exceptions is not a "natural" answer. Problems also arise if the parent of the handler is killed before exception can be raised — giving rise to non-determinism due to "orphaned" handlers. Tiwary proposed a handler forwarding mechanism to overcome this, forcing all processes to leave a forwarding address for exceptions before they die. Language-wise, a toolkit approach was proposed using objects to represent exceptions and signals, not dissimilar to that presented by Deutsch earlier.

3.7 Andrew Watson — Exceptions in ANSA DPL

Watson's presentation also concerned distributed systems. ANSA is a distributed architecture with an autonomous object-based (no classes) programming model. DPL is a statically typed language, its blocks are instruction sequences which successfully result in one anonymous termination (normal case) or fail with one of a set of named terminations (exceptions). It uses these terminations to provide a dynamically scoped result mechanism, not dissimilar to the catch/throw mechanism in Lisp. DPL terminations can be used as an exception signaling mechanism and to provide multiple result signatures for operations.

Exceptions are used as a natural programming technique for handling asynchrony; errors become part of the technique and style of programming. Resumption is not an acceptable option for exception handling in this system mainly because of the heterogeneous and distributed nature of the applications. Further, all actions regarding exceptions appear to be handled as synchronous events to fit in with the transaction-based model for all activity in an application.

The system does not use stack searching or other such mechanisms to associate handlers with exceptions. Rather the handler is a hidden parameter in a function call. This is reminiscent of the BETA system. A type error results if an object has terminations its clients cannot handle (except communication failure, which, given its ubiquitous presence in distributed environments, all objects can handle). Note that analogies can be drawn between terminations and continuations (as in functional languages), and between exceptions and closures being passed around.

4 The Discussion

The discussion session was relatively anarchic. The main reason for this was not so much lack of control by the chair, rather that we were trying to find out what the crucial questions were as well as answering them as opposed to investigating solutions to particular, well known questions. What we have attempted to do in this report is factor out various thoughts and comments related to a number of specific questions identified by the workshop.

4.1 Does exception handling violate encapsulation and what are its connections with reusability?

Serge Lacourte explained that if a programmer pops data from a stack object and gets for example an `emptyList` exception, this can reveal too much about the underlying implementation of the stack. That is, it can violate encapsulation. Also, information about how the exception was caused and where exactly it came from, which is often passed to the handler (see the instance variable `originator` in (ParcPlace Systems, 1989) or `signalingContext` in (Dony, 1990)) violate encapsulation when passed up the invocation chain.

The group agreed that each method or class should have the exceptions it can signal defined in its signature and that these exceptions should be of the same conceptual level as the service performed by the method. Encapsulation need not be violated if, at each stage of propagation, the propagating object alters the name of the exception (and its details) to reflect the increased level of abstraction at that stage. Conversely, exceptions that are not at the same conceptual level as the service offered should not be propagated outside of the method. This last case has been called blind propagation because it generally happens when a method does not trap the exceptions raised by the inner method call it performs. However, Knudsen and Schmidt pointed out that for some general purpose exceptions like "failure", or in all cases in which a method does not know exactly which operations it will call, blind propagation was unavoidable and useful. This situation can always happen wherever there is genericity in dynamically-typed languages or indeed whenever there are generic types. It can also be a problem in the presence of any higher-order programming capability (methods or blocks passed as arguments).

With respect to reusability, it has been said that not to propagate exceptional results of a conceptual level equal to that of the operation breaks reusability, since callers of operations generally have better solutions for handling than general default handlers or static handlers that are unaware of the computation history.

4.2 What Parts of Exception Handling must be Implemented in the Language and What Parts as Language Primitives?

This question caused a distinct dichotomy in the group between those who felt that an exception handling system should be implementable on top of a language and those that felt that exception handling should be embodied in primitives of the language.

One argument against exception handling being defined in terms of language objects (instead of primitives) is that constructs like unwind protection cannot be made atomic since exception objects can always interrupt it. Deutsch countered that this was just an argument for better atomicity constructs. Winder asked, in a system defining exception handling mechanisms using language objects and classes, how does one combine code using different exception handling policies?

There seemed to be little on which to base a reasoned decision and in the end, it was felt that either route was possible (as long as some technique for creating atomicity existed) and it was really a question of language and exception interpretation philosophy.

4.3 Separation of Main Code and Exception Handlers

The separation of the code dealing with exceptional situations from the standard code of an application seems to be an appealing problem for many people. Separation is not a clear issue.

On the one hand it can aid code readability by extracting main code from detailed and perhaps complex exception detecting and handling code. It can also enhance reusability if the handlers associated to expressions in a method are separated from the method code. The handling part of the method can be then redefined in a subclass without the necessity of modifying the method itself which would simply be inherited.

On the other hand, this separation can hinder readability by making exception code difficult to find or refer to. It can also make programs difficult to understand in the cases where the semantics of exception handling is very dependent on the positioning of the statements within the main code (for an example of this problem, see (Lacourte, 1991) in which some ways to achieve such a separation are provided). In practice, we have noticed that separation was already achieved in existing languages whenever it was possible i.e., when the handlers are general enough to be disconnected from specific code. Examples of this include default handlers, class handlers, rescue clauses and the specifications of exceptional behavior, as proposed in (Purchase & Winder, 1990).

In conclusion, most workshop attendees thought separation was a good idea, so long as the exception control flow was not especially interwoven with the main code. Physical separation is fine as long as it does not imply scope separation.

4.4 At What Level of Modularity Should One be Able to Associate Handlers with Code

By far the most common association mechanism, in modern languages, is to associate handlers with expressions. Such a handler will trap the exception signalled by the execution of the protected instruction.

To associate handlers with classes is less common and raises several issues. In Smalltalk, the role of a class handler defined on a class A is to trap all exceptions for which the invocation of the signaling primitive is lexically included in a method defined on A or on A's subclasses. With such handler search semantics, exceptions are not propagated along the invocation chain, which leads to many problems of reusability. An alternative strategy (Dony, 1990) is to allow a class handler defined on class A to trap all exceptions signaled by the invocation of the method defined on A and on A's subclasses, at the location of the signaling primitive's invocation. For example, if M is defined on A and calls a method M′ defined on class B which raises an exception, then the class-handler on A will be invoked if not hidden by a class-handler on B.

Several questions were asked about class handlers. The first, by Watson, concerned what to put in them. The answer was that we were lacking the experience of using them but, what can be said is that class handlers are the perfect place to store behavior about exceptions that are relevant wherever the exception is raised in the class. For example, when considering the class stack, ensuring no exception except "underflow" or "overflow" should ever be propagated outside a method of that class. The second question, from Snyder, was: are class handlers simply a shorthand? They can be seen as a very powerful shorthand when considering that a class handler defined on a class A could be replaced by the association of the same handler with the body of all the methods defined on the A sub-hierarchy. They are, in fact, more than a shorthand because the class handler will automatically be applicable for any new method defined in the A sub-hierarchy. This gives the implementer a way of controlling reusability. This immediately raised the question: why does Eiffel not support inheritance of handlers? Nerson asserted that it would be too complex especially in the presence of multiple inheritance and that in Eiffel rescue clauses were part of the implementation of classes and should not be inherited.

Handlers associated with classes and methods were thought to be justified, but Dony challenged whether handlers associated with individual instances of classes (as in BETA) could create problems. However, this is a problem not directly connected with exception handling but with the semantics of object-oriented languages: should classes own the whole structure and behavior of their instances or simply a part of it? This was clearly a question out of the scope of this workshop.

Glossary

Activation: invocation results in the activation of the invoked procedure.

Association of a handler to an instruction(s) for an exception: states that the handler will be executed if the execution of the instruction(s) signals the exception.

Caller: The method which has invoked the current method.

Exception: Apart from the debate about what is an exception from a philosophical point of view, we could, from an operational point of view in programming languages, define an exception as an abnormal event occurring at run-time and making it impossible to execute the next instruction in the program.

Handler: the exceptional continuation.

Invocation: textual part of a procedure denoting the call to an operation.

To Signal (or Raise) an exception: to interrupt the usual sequence of operations, then to search and to invoke an exceptional continuation. "An activation may signal an exception, the invocation that caused the activation raises that exception" (Liskov & Snyder, 1979).

Signaller: the method activation that signals the exception.

Single level signaling mechanism: this definition and the following one only apply if, when an exception is raised, the invocation chain is dynamically searched for a handler. It denotes that a handler has to be defined by the caller of the current method activation.

Multi-level signaling mechanism: the handler does not have to be defined by the direct caller of the current method activation.

Exception propagation: usually has two meanings:

1. denotes the multi-level traversal of the stack during handler search (the term of "blind propagation" has been used during the workshop),
2. denotes the ability to re-signal the trapped exception within a handler.

Resumption: after handler execution, the errant method restarts at the instruction that follows the signaling one. The handler may be able to pass a value to be returned as the value of the signaling method.

Termination: after handler execution, the errant method restarts at the instruction that follows the association point. If not done during handler search, the part of the stack located between the signaling point and the handling point is discarded. The retry variant of termination causes the re-execution of the protected expressions after the stack has been discarded.

Acknowledgments

Although the authors of the paper were the organizers of the workshop, there would have been no workshop without the participants. It is therefore appropriate to give a full list of attendees (in alphabetic order of surname): Brian Brown, Peter Deutsch, Christophe Dony, Sabine Finke, Franz Hauck, Mats Henricson, Jørgen Lindskov Knudsen, Serge Lacourte, Ole Lehrmann Madsen, Boris Magnusson, Jean-Marc Nerson, Oskar Permvall, Jan Purchase, Markku Sakkinen, Heinz Schmidt, Alan Snyder, Ashutosh Tiwary, Andrew Watson, Russel Winder.

We would like to thank all these people for a good session which we feel tied up a lot of loose ends regarding exceptions and enabled us to create coherent questions that need answering.

We would also like to thank the ECOOP'91 organization for giving time, space and support for the workshop.

References

C. Dony, "An Exception Handling System for an Object-Oriented Language," Lecture Notes in Computer Science, vol. 322, pp. 146–161, Springer–Verlag, August 1988, Proceedings of ECOOP'88, Oslo.

C. Dony, "Exception Handling and Object Oriented Programming: Towards a Synthesis," ACM SIGPLAN Notices, vol. 25, no. 10, pp. 322–330, October 1990, Proceedings of ECOOP/OOPSLA '90.

J. B. Goodenough, "Exception Handling: Issues and a Proposed Notation," Communications of the ACM, vol. 18, no. 12, pp. 683–696, December 1975.

J. Ichbiah, "Preliminary ADA Reference Manual: Rationale for the Design of the ADA Programming Language," ACM SIGPLAN Notices, vol. 14, no. 6, June 1979.

J. L. Knudsen, "Better Exception Handling in Block Structured Systems," IEEE Software, pp. 40–49, May 1987.

A. Koenig & B. Stroustrup, "Exception Handling for C++," in *Proceedings of Usenix'90*, pp. 149–176, San Francisco, USA, April 1990.

S. Lacourte, "Exceptions in Guide, an Object-Oriented Language for Distributed Applications," Lecture Notes in Computer Science, vol. 512, pp. 268–287, Springer–Verlag, July 1991, Proceedings of ECOOP'91, Genève, July 1991.

B. Liskov & A. Snyder, "Exception Handling in CLU," IEEE Transactions on Software Engineering, vol. SE-5, no. 6, pp. 546–558, November 1979.

B. Meyer, *Object Oriented Software Construction*, Prentice–Hall, 1988.

J. G. Mitchell, W. Maybury & R. Sweet, *The MESA Language Manual*, Xerox Research Center, March 1979.

D. Moon & D. Weinreb, *Signalling and Handling Conditions, LISP Machine Manual*, MIT AI Laboratory, 1983.

Objectworks for Smalltalk-80, version 2.5, Advanced User's Guide, Exception Handling, ParcPlace Systems, 1989.

K. Pitman, "Error/Condition Handling," Contribution to WG16, revision 18, Proposals for ISO-LISP. AFNOR, ISO/IEC JTC1/SC 22/WG 16N15, April 1988.

Jan Purchase & Russel Winder, "Debugging Tools for Object-Oriented Languages," RN/89/77, pp. 10–27, June 1991.

Jan Purchase & Russel Winder, "Message Pattern Specifications: A New Technique for Handling Bugs in Parallel Object Oriented Systems," ACM SIGPLAN Notices, vol. 25, no. 10, pp. 116–125, October 1990, Proceedings of ECOOP/OOPSLA'90.

G. A. Roberts, M. Wei & R. L. Winder, "The Solve Object Oriented Programming System for Parallel Computers," SPAN-WP10-Deliverable-28, Department of Computer Science, University College London, March 1990.

R. L. Winder, J. Purchase, G. A. Roberts & M. Wei, "The Solve Object Oriented Programming System," 1991, (In preparation).