An Object-oriented Exception Handling System for an Object-oriented Language.

Christophe Dony
Laboratoires de Marcoussis, CGE Research Center,
route de Nozay, 91460 Marcoussis, FRANCE.
&
LITP, University of Paris VI,
4 place Jussieu, 75005 Paris, FRANCE.

Abstract.

We present an original exception handling system especially designed for object-oriented languages, making actual information hiding possible and taking into account specific issues of object-oriented languages. It allows association of handlers with expressions as well as with object classes, using a well defined semantics. It offers an object-oriented and extensible representation of exceptions, handlers and knowledge about exceptions. Handlers can specify both resumption and termination. There are no distinctions between system and user defined exceptions. With this system, fault tolerant programs and well specified encapsulations can be written, simple and powerful integration of new user-defined exceptions and secure as well as readable non local moves can be implemented.
In this paper, we examine object oriented specific issues related to exception handling. We discuss the exception handling mechanisms available in current object-oriented languages and explain why they do not provide the ability to define fault tolerant encapsulations. Our system description shows how the utilization of the object-oriented formalism solves, in an efficient and simple way, some well known problems related to exception handling such as : how to create exception hierarchies, how to signal fatal or continuable exceptions with the same primitive, how to pass arguments to handlers, and so on.

Key words and phrases : Exception handling, Object-oriented programming, Fault-tolerant encapsulations, Resumption model, Debugging environments.

1. Introduction.

We present an original exception handling system, designed for an object-oriented language and taking advantage of the object-oriented formalism. This system is implemented and currently used.

Our system improves program reliability and reusability [Meyer 87] by allowing the creation of fault tolerant encapsulations, making actual information hiding possible. Indeed, each procedural abstraction (method) is able to handle lower level exceptions raised by inner method activations; and is able to specify all its possible responses, including exceptional ones.

Since atypical events can be dealt with, we improve the expressiveness of the language. We allow dialogues based on exceptional situations : a method M can answer "*I don't know*" or "*impossible to do this because ...*", by raising an exception. The method caller can anticipate and say "*if the response to this message is an impossibility then ...*". Control structures that make these dialogues

possible allow secure and clear non local moves [Testard-Vaillant 86] to be written, as they provide a way of communication between nested method invocations.

An important characteristic of the system relies on the object-oriented representation of exceptions and handlers. Some well-known exception handling problems are thus easily and efficiently solved by the possibility of defining slots and methods on the method, handler and exception classes. For example, two slots allow handlers to know precisely where the exception was raised. New possibilities arise, for example exceptions can be organized in hierarchies. The debugging possibilities of programs which use exception handling mechanisms are enhanced. For example the system can be requested to give the name of all the methods where a given exception could be raised.

We take into account the object-oriented representation of problems as well as the communication by message passing while allowing association of handlers with both expressions and classes. All handlers take into account the exception hierarchy.

Our system is based on an analysis of similar works for procedural languages such as *Pl/I* [Pl/I 78], *Clu* [Liskov 79], *Ada* [Ichbiah 79] and *Mesa* [Mitchell 79]. Basic definitions and references are provided in section 2.

In section 3, we discuss exception handling problems specific to object-oriented languages. We explain why attaching handlers to classes is insufficient and does not enhance reusability, why attaching handlers to dynamic units is also insufficient, why exceptions should be first class objects and why *unknown-selector* should not be the only fundamental exception.

Our system is also based on a study of exception handling systems in current object-oriented languages. In section 4, we review data structures within the *Lisp+Flavors* system [Moon 83]. We examine the signaling and handling possibilities of current object-oriented languages with special attention to *Smalltalk* [Goldberg 83].

Section 5 is a presentation of our system. We briefly introduce the language *Lore* for which it is implemented. We give a system overview and explain our main choices. We then describe how to create, signal and handle exceptions, how to restore environments when leaving a sequence of code, how to define default-handlers and interactive resumption propositions.

2. Exception handling basic definitions.

Research for improving verifiability, reliability and robustness of programs led to concepts such as modules [Parnas 72], data abstraction [Hoare 72], software engineering [Boehm 75] and software fault tolerance and avoidance [Liskov 79] [Christian 82]. The program structures for handling exceptional events [Goodenough 75] [Levin 77] [Berry 85] [Knudsen 87] are mainly designed to implement fault-tolerant encapsulations, i.e. software objects able to return well defined and foreseen answers, whatever may happen while they are active, even though an exceptional situation occurs.

An exception can be defined as a situation leading to an impossibility of finishing an operation. The term "exception" implies that this situation is not necessatily an error case. Three types of exceptions can be distinguished (see [Goodenough 75] for examples) : the **domain exceptions** raised when the input assertions of an operation are not verified, the **range exceptions** raised when the output

assertions of an operation are not verified or will never be, and the **monitoring exceptions** raised to implement controlled non-local moves.

To **raise** (a procedure invocation raises an exception) or to **signal** (a procedure activation signals an exception) an exception results in interruption of the usual sequence followed by search and invocation of a handler for that exception.

Handlers are attached to (or associated with) entities for one or several exceptions (according to the language, an entity can be a program, a process, a procedure, a statement, an expression, or a data). Handlers are invoked when an exception is signaled during the execution or the utilization of one of these protected entities. They can raise new exceptions, **exit** or **terminate** the entity invocation and sometimes **resume** the signaler, i.e. transfer control to the statement following the signaling statement.

3. What an exception handling system should provide.

Because of the lack of space, we focus on exception handling problems specific to object-oriented languages. Four major issues are developed : is there a need for new exceptions? Which handler attaching possibilities should be provided? How to define fault-tolerant encapsulations? What can be expected of object-oriented representation of exceptions? We outline eight main objectives that constitute our system basis.

3.1. New exceptions.

Object-oriented design allows operations to be broken down into sub-operations defined on relevant sub-domains of the whole definition domain. Then, message sending (as in *Smalltalk*) as well as generic functions and multi-methods (as in *Common-loops* [Bobrow 86]) may convert numerous domain exceptions (and some range exceptions[1]) into exception **unknown-selector** raised[2] by the message sending primitive. For example, evaluating the message ''*["abc" + 1]*'' raises : ''*Error: string "abc" does not understand message +*'', instead of a conventional domain exception ''*Error: +, argument "abc" is not a number*''. One may think that since all domain tests are performed via message sending (or generic function invocation), all classical exceptions are converted into *unknown-selector*. However, in many cases, specific exceptions have to be raised. On the one hand, this is the case (1) when the input assertions of an operation may be based on the types of several arguments and not only on the receiver's type (this remark does not apply to generic functions); (2) when the input assertions are not based on argument types or include complex calculations. On the other hand, it is not always possible (or it is too expensive) to divide an operation into sub-operations defined on sub-domains. This last point is true when sub-domains do not exist and cannot be defined (as classes) in the language (for example ''positive integers'' or ''[1 5] union [7 9]'') or when this breakdownwould lead to the creation of too many subclasses, etc.

---

[1]A range exception for an operation is sometimes nothing but a domain exception for an inner operation. E.g. *end-of-file* is a range exception for procedure *read-string* and is a domain exception for inner operation *read-char*.

[2]when no method is associated with the given selector in the given receiver class, or when there is no method the parameter type tuple of which matches the given argument tuple.

Objective 1 : Although there are fundamental exceptions, it should be possible for the users, to define new ones.

3.2. Where to define handlers.

One of the characteristics of the object-oriented design is that procedural knowledge is attached to object classes. We think that such a possibility is not powerful enough to store the variety of responses to exceptional situations. Three levels of knowledge about handling exceptional events should be distinguished : (1) knowledge that is independent from any execution context as well from the object data-base state, (2) knowledge that is common to a set of objects, (3) knowledge related to a language expression the evaluation of which may signal an exception All these points depend on handler attachment capabilities.

• **Exception handlers**. The most general handlers must be valid in any cases, e.g. they can print an error message or make a general correction proposition. They are always invoked when no more specific handlers can be found. They should be associated with exceptions themselves.

Objective 2 : It should be possible to attach handlers to exceptions.

• **Class handlers**. Attaching handlers to an object is not a new idea [Goodenough 75], such handlers are invoked when an exception is raised during an access to that object. From an object-oriented view, similar handlers can be attached to object classes. They allow to define, for all instances of a class, a common behavior in front of exceptional situations, which however is independent of any execution context. For this last reason, we will speak of static handlers[3].

Here is an example where static handlers are useful : encapsulation of all the messages to a particular object [Pascoe 86]. We want to perform some operations around the execution of the original message. Some applications are : debugging, concurrent access or context restoration. What is done in the handler is independent of the execution context. *Encapsulator* is a class on which a handler for exception "unknown-selector" has been defined. Each instance of this class can then encapsulate another object. When a message is sent to an encapsulator object instead of the encapsulated one, the exception *unknown-selector* is raised and handled, then additional operations are performed around the original message execution the result of which is used to resume the signaler.

Objective 3 : It should be possible to attach handlers to classes.

• **Expression handlers.** Handlers attached to expressions allow the specification of the program continuation when the invocation of the expression to which they are attached raises an exception that they handle.

Here is an example where expression handlers and context-dependent responses to an exception are needed : two methods defined on the same class have to react in different ways to the same exception, which is, in both cases, raised by sending the same message to the same object. Handlers defined on

[3][Liskov 79] spoke of static association to enhance the fact that in *Clu* it is possible to determine, by a static analysis of the program, the handlers to be invoked. Since new classes are rarely dynamically defined, this is also true of our "class handlers".

a class would be insufficient since they neither know where the exception was raised nor in which nesting context; they cannot provide a context-dependent answer. Let two methods *divide* and *modulo* be defined on the class of *natural numbers* using method *minus* which is itself defined with primitive method *predecessor*, raising the exception *negative-number* when the message predecessor is sent to 0. Division of x by y is defined by successive subtractions (*minus*),each time a counter is incremented, no entry assertion is tested but the exception *negative-number* caught and the counter value is returned. *Modulo* is defined in the same way but there is no counter and the handler for the same exception returns the value of x (cf. § 5.5).

Notice that, in these definitions, handling exceptional cases is an important feature because it avoids the test "$Y \leq X$" which is as costly to compute (in our example) as the operation "X minus Y". Moreover, since it is not possible (in general) to define the subclass of numbers that are lower or higher than others, this test could not be bypassed by subclass creation and message sending.

Objective 4 : It should be possible to attach handlers to dynamic entities.

3.3. Fault-tolerant encapsulation.

Two opposite solutions to information hiding can be found in object-oriented languages. In the first one, creating fault-tolerant encapsulation is viewed as the ability to handle exceptions within object classes, i.e exceptions are not propagated outside the method in which they are signaled but handled within the receiver's class. In the second one, a fault-tolerant encapsulation is an entity able to catch and hide the exceptions raised by their inner modules, but also able to propagate relevant exceptions and return exceptional answers.

Objective 5 : Exceptions should first be propagated along the invocation chain.

With such propagation rules, a method has a predefined protocol to answer "*I do not know*" or "*I cannot do this*" to its caller. Furthermore, it enhances reusability since message senders are able to give context-dependent answers to exceptional situations.

As a result of such a decision, all static handlers become default handlers since they are invoked only if no expression handlers has been found. A specific primitive then has to be designed in order to implement the encapsulator example.

3.4. Object-oriented knowledge representation.

In procedural or functional languages, exceptions are strings, symbols or variables declared to be of type: "exception". Knowledge about exceptions is scattered in the handlers. It is odd to see that most object-oriented languages do not take advantage of their data structuring possibilities to represent exceptional events. In *Smalltalk*, an exception is a selector but not a first class object. Thus it cannot own any characteristics, cannot be inspected, modified or upgraded.

Yet, exceptions are complex entities, they own not only attributes such as the context where they are raised, but also procedural characteristics that describe for example how to report an error message or how to propose solutions to the failure. As soon as exceptions are represented as data-types, following the ideas developed in the *flavors* system [Moon83] or in *Taxis* [Nixon 83], *it* becomes simple and

natural to specify their static characteristics as attributes (slots) and their dynamic knowledge as methods.

Objective 6 : Exceptions should be hierarchically organized classes.

Here are some of the main advantages of this choice.

- An exception (a concept) or an instance of an exception is, and can be used as, a first class object, with all debugging and extensibility consequences this can entail.

- Exception handling specification benefits from the object-oriented formalism advantages. Exceptions can be organized in a hierarchy based on common behaviors : exception *divide-by-zero* is naturally a subclass of *arithmetic-exception*. Properties can be shared : when *unbound-instance-variable* is raised and not handled, the following proposition ''P'' is made : ''supply a value to store as the value of X''. This proposition is not only valid for *unbound-instance-variable* but for *unbound-variable* too; for this reason, it is defined on an upper-class named *cell-content-error* as a *document-proceed-type* method. Besides, one method is defined on each exception to properly store the new value.

- All predefined properties are reusable via subclassing and method overriding. The new exceptions can be integrated as ''sub-exceptions'' of existing ones. As a consequence, the handlers for the new exceptions can take advantage of all the behaviors already defined on the upper exceptions. Furthermore the system and all the applications using this exception handling system will offer **in the same way the same basic default debugging environment.**

- Handler definition is powerful, since handlers do not only handle one (one kind of) exception but all exceptions that are sub-classes of it. E.g, handlers for *arithmetic-exception* catch *overflow*, *divide-by-zero* as well as *arithmetic-exception* itself.

Objective 7 : All handlers should take into account the exception hierarchy.

Such ideas can still be improved : e.g., specific handling possibilities may be defined on suitable exceptions. Consequently, thanks to message passing, it is implicit that the resumption message cannot be sent to a fatal exception just as the termination message cannot be sent to a proceedable one.

Objective 8 All handling primitives should be generic operations.

4. Object-oriented exception handling systems.

4.1. Data structures.

As we pointed out in the last section, there are few examples of object-oriented languages where exceptions are first class objects, *Flavors* being one example [Moon 83]. Main principles of this system (from the data structure viewpoint) relies on the fact that 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 argument. Thus handlers can access, via inspection and message sending, to the behavior defined on this exception (this flavor). Handlers are neither objects nor methods but *Lisp lambda*

expressions, they can only be attached to expressions, both resumption[4] and termination are possible. *Taxis* provides similar capabilities and also allows to associate default handlers with exceptions.

Important works, based on the object-oriented representation of exceptions in the language *Taxis*, have been done in languages designed to manage data-bases and information systems [Borgida 85]. The adaptation of these techniques to object-oriented languages [Borgida 86] brings efficient solutions to the problems raised by exceptional objects and values. Exceptional objects can be exceptional instances (objects with slot values breaking constraints) or exceptional subclasses (that do not want to inherit certain properties defined on upper-classes). Exceptional data are stored as instances of exception classes and do not interfere with "normal" ones. This prevent from creating multiple "abstract" classes or modifying the range of the slots. Exceptional data can be accessed after exceptions were raised, these accesses are slower than classical ones, but standard computations are not slowed down by management of possible occurrences of exceptional cases.

4.2. Signaling and handling capabilities.

• **Standard solutions.** A classification of object-oriented languages based on their exception handling possibilities brings out three categories, where the main specificities remain in the handler attachment possibilities. None of these categories is wholly satisfactory.

Smalltalk is the main example of the first one, where handlers for all exceptions can only be statically attached to classes.

The second kind of languages consists in extensions of procedural or applicative languages. These extensions are turned towards new data type creation without inheritance (for example *Clu*) or with inheritance (for example *Simula* [Dahl 70], *C++* [Stroustrup 86] or *Flavors*). These extensions have been done without modification of the standard or existing exception handling mechanisms. Regardless of the resulting systems - there is no room here to examine all the possibilities - these do not provide solutions to associate handlers with classes or with exceptions (objectives 2 & 3).

Finally, some languages (such as Loops [Bobrow 83] or Objvlisp [Cointe 84]) also built on top of procedural or functional ones, choose a compromise: some exceptions relative to object manipulations are raised and handled as in *Smalltalk*, the other ones depend on the underlying language. As a consequence, the handling possibilities are unequal : it is impossible to attach a handler for *unknown-selector* to an expression, and impossible to attach a handler for *divide-by-zero* to a class.

• **Smalltalk.** The *Smalltalk* solution to exception handling is a model for many object-oriented languages. In order to signal run-time exceptions, the *Smalltalk* evaluator sends, to the current object, a message corresponding to the current exception. Therefore handlers are methods pointed out by exception selectors, and are attached to classes. Thus, exceptions raised by methods defined on a class are handled within that class. There are three main possible selectors : *DoesNotUnderstand* (of which *ShouldNotImplement* and *SubclassResponsability* are variants) reports the *unknown-selector* exception, *PrimitiveFailed* reports system errors and lastly the selector *Error* reports all the other exceptions.

---

[4]Handlers can be executed in the lexical environment of the caller when they are defined using the *function* special form.

This system is simple and efficient since it only uses message sending and method definition which are basic operations. Besides its expressive power seems insufficient and some of the above-mentioned objectives are not fulfilled satisfied.

- Handlers cannot be attached to expressions or statements. Exceptions cannot be propagated to operation callers; as stated earlier, this limits the reutilization possibilities.
- There is no well suited location to store knowledge (about exceptions) which is independent of the class hierarchy. In other words, all of this knowledge has to be stored in handlers attached to the hierarchy root class (object).
- Exceptions are not objects, they cannot own properties, they are not organized in a hierarchy.
- There is no simple and predefined way to create new exceptions (this must not be confused with the possibility of signaling new exceptional cases using the selector "error").
- Exceptions, except the two basic ones, are anonymous; a handler cannot know which exception it handles unless some meaningful arguments are provided.
- All of the knowledge about an exception must remain in its handlers. Then, the same method "error" defined on a class may have to handle several exceptions. This may lead to confusion and non object-oriented programming style.

5. Description of the system.

5.1. The Lore language.

Our system has been developed and implemented for the object-oriented language *Lore* [Caseau 87] [Benoit 86] dedicated to knowledge representation. Non basic parts of our system are written in *Lore* and take advantage of its possibilities. Here are some important features. Classes are sets, properties defined on classes (slot, methods, etc) are relations whose domain and range are *Lore* sets. All properties such as *slots, relations* or *methods* are objects. Among all the consequences, one is, for example, that properties can have properties. Communication is based on message passing and on the multi-method concept : methods are defined on the message receiver class but several methods with the same name can be defined on a class. Here are a few syntax examples. (Words in italic are *Lore* predefined objects, bold words are existing slots, methods or special forms)

- Message sending ≡ [receiver selector arg1 ... argN]
- Class creation ≡ [square **isa** *class* **superset** figure]
- Relation definition ≡ [square **has** *relation* length **range** *integer* **init-value** 10]
- Method definition ≡ [square **has** *method* grow **comment** "..."
filter (i integer) ; Parameter name and type.
form [oself length **is** [[oself length] + i]]] ; The body of the method.
- Contracted definition ≡ [rectangle **isa** *class* **superset** figure **with**
(*slot* length **range** *integer*)
(*method* area **form** [[oself length] * [oself width]])]
- Square instantiation and slot assignment ≡ [s1 **isa** square length 10]

- Assignment ≡ [value -> variable-name]
- Modifying a slot value ≡ [s1 length is 20]
- Asking for squares the length of which is 20 ≡ [length.square of? 20]

5.2. System overview.

Our exception handling system can be used in the same way by the system and the by user programs. Our specification is based on the above-mentioned objectives.

• **Status of exceptions.** Exceptions are classes (objective 6); an instance of an exception X is created each time X is raised. Defining an exception is nothing but creating a new class (objective 1), new exceptions can be raised and handled exactly in the same way as predefined ones. Two basic exception types are defined : the fatal (**error**) and the proceedable (**warning**) ones.

• **Status of handlers.** Handlers are methods (i.e objects in *Lore*) or instances of a specific class named **protect-handler**. They can be attached to expressions, to classes and to exception classes (objectives 2, 3 & 4), they are searched and invoked in that order (see § 5.4). Static handlers (attached to classes) are default handlers. All handlers are aware of the exception hierarchy (objective 7), the handlers for **exceptional-event** (the exception lattice root) will thus catch all exceptions. The mechanisms used to search and invoke handlers are the standard *Lore* mechanisms used to find (using the receiver, the selector and the arguments) and invoke methods.

• **Which model?** Both the **resumption** and the **termination** models are supplied. Termination means that the method activation that raises the exception is exited. Resumption means that after the execution of the handler, control returns to the statement following the signaling one. Although the resumption model has some drawbacks [Liskov 79], we choose it for its expressive power useful for interactive or simulation applications, where operators, users or experts may bring interactive solutions; debugging propositions are based on the same principle [Moon 83] [Wertz 83]. This model is as efficient to handle and restart long computations (the same possibilities can be obtained with a termination model assuming association of numerous handlers which can be time and place consuming and makes programs tedious to write). Lastly, the resumption model offers to all users a well defined way to access calculation history [Testard-Vaillant 86], which can be very useful to compute special applications, to explain a calculation or to show the current state of a program execution.

• **Handling possibilities.** All handlers may reference the variable **xself**, they can access and modify global variables and objects. Three methods are defined on exception classes (objective 8). <u>Exit</u> and <u>retry</u> defined on **error** and <u>resume</u> defined on **warning** can be used within handlers. **Resume** causes the resumption of the interrupted computation, **exit** leads to the execution termination of the expression to which the handler was attached, **retry** is a variation of exit such that, after termination, the expression is once again executed under the same protections. It is possible to signal an exception within a handler.

• **Semantics of signaling.** The method <u>signal</u> is designed to raise any exception. Arguments can be submitted and will be transmitted to handlers. The signaler can choose the type of the signaled exception. The handlers are responsible for resuming or exiting. These choices are consistent with the

theory according to which the signaler knows the seriousness of a situation, whereas the caller knows the context of the operation invocation and can decide with full knowledge of the facts.

5.3. Exception classes.

Exceptions are instances of the class **exception-class**, itself a subclass of **class**. There are four basic instances of **exception-class**. <u>Exceptional-event</u> is the exception lattice root. <u>Error</u> is the set of exceptional-events for which resumption is impossible. <u>Warning</u> is the set of exceptional-events for which termination is impossible. For the instance of <u>exception</u> - subset of **error** and **warning** - both termination and resumption are possible. (cf. Fig. 1).

• **Creating new exceptions.** An instantiation of **exception-class** creates a new exception on which slots and methods can be defined; these slots can be assigned while signaling and accessed (modified) within handlers. This solution provides a communication mechanism between signalers and handlers. Three basic slots are defined on **exceptional-event** to contain the signaling context (**xobj xprop**) and general arguments (**xargs**). Several methods are defined, among which **print** is intended to display the basic error message.

The following example shows the definition of **unknown-selector** where the method **print** is redefined, and makes use of the slot values to report the error.

```
[unknown-selector isa exception-class superset exception with
  (slot receiver  range  object  comment  "The receiver of the wrong message.")
  (slot selector  range  symbol  comment "The selector of the wrong message.")
  (multi-slot args  range  object  comment "The Arguments of the wrong message.")
  (method print form .... [[oself receiver] print] ....)]
```

5.4. Signaling.

• **Syntax.** Any method can signal an exception. **Signal** is a method defined on the class **exception-class**, so signaling consists in sending the message **signal** to an instance of **exception-class**. **Signal** is nothing but a redefinition of the method **new** that first creates an instance of an exception and then searches an handler for it.

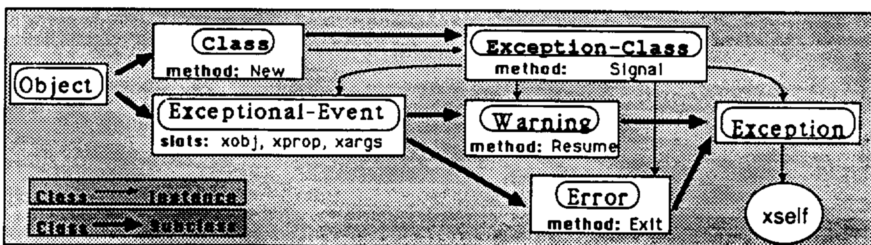

Fig. 1 : **Basic exception types and meta-type.**

It is important to understand the distinction between the generating class **exception-class** which is a meta-concept, an exception such as **error** which is an instance of it, and an instance of *error* created when *error* is raised (cf. fig. 1).

Exception instances are always named xself, (from now this term will be used to reference the instance of a current exception). Their slots xobj and xprop are automatically assigned to the object and to the property that are currently active when the exception is raised. All others slots defined on the exception are assigned to the given values (if not provided, default values are used). Any handler will receive **xself** as unique argument.

```
[unknown-selector signal  receiver 1  selector 'foo  args '(2)]
```

• **Defining method interfaces and retreiving information.** The two relations **signals** and **propagates** are defined on the set named **method** to store the exceptions that a method may raise. The following definition of the method *divide* specifies that it may signal the exception *divide-by-zero*.

```
[integer has method divide  signals divide-by-zero  filter (...)  form ...]
```

This information can be retreived by inspecting the method interface (the values of all its slots) or by searching all the methods where exception may be raised (this can be done in a simple way in *Lore* with the relation *of?* defined on the set *relation*)

```
[signals of? divide-by-zero]
= (divide.integer  modulo.integer  ....)
```

• **Looking for a handler.** To fulfil the fifth objective, the "call relation" [Levin 77] is first taken into account i.e. a handler attached to the expressions that dynamically include the signaling one is searched. **The search stops as soon as a handler, the parameter type of which is an upper type of the signaled exception, is found.** By default, the dynamic handler attached by the system to the top-level loop is found and invoked. It first looks for default-handlers attached to the class or upper classes of the signaling-time active object[5] (value of *[xself xobj]*); if none can be found, it looks for default-handlers attached to the signaled exception itself (*xself*'s type).

5.5. Handling.

One or several handlers can be attached to any expression or class. All handlers have one parameter named xself the type of which is the exception to be handled and which will be bound to the instance of the current exception as soon as the handler is invoked. Determining whether a handler is valid consists in matching its parameter type with the current exception one.

• **Handler bodies.** The methods **resume**, **exit** and **retry** can be used within handlers, they accept an optional argument which is evaluated in the handler environment. Of course, exceptions can be signaled within a handler.

---

[5] i.e. the last receiver of a message. For all exceptions raised by the system while analyzing a message invocation (ex. unknown-selector), this definition is ambiguous; in such cases, we consider the receiver of the wrong message as the active object.

• **Resumption** : The method **resume** is defined on the class **warning** (cf. fig. 1); thus, this message cannot be sent to an instance of **error**. The argument value becomes the value returned by the method **signal**. This implies that a method which signals a non fatal exception should foresee what it will do with the signaling result, in case the exception is resumed.

• **Termination** : The method **exit** is defined on the class **error**. The context between the signaler and the association is discarded, the argument value is returned as value of the expression to which the handler was attached.

• **Defining dynamic handlers.** Dynamic handlers are instances of the class **protect-handler**, internally attached to expressions by using the primitive <u>protect</u>, their extent is dynamic and their scope is undefined, they are executed in the lexical environment of the association operation. The definition syntax is "[*handler protect expression*]", a handler takes the following form "*(exception-name form handler-body)*". Within the handler body, the implicit parameter **xself** can be referenced.

• Here is an example using termination which implements the *modulo* examples, where *minus* raises the exception *negative-number* (cf. § 3.2).

```
[natural has method modulo filter (p natural)
  signals divide-by-zero  ; May signal this exception.
  form if [p = 0] then [divide-by-zero signal] else
          ; A handler for negative-number exits the while loop
          ; and returns current oself value.
          [(negative-number form [xself exit oself]) ; The handler.
            protect ; Attaches the handler to the while loop.
            [t while [[oself minus p] -> oself]]]]]
```

• Here is another example using **exit** and **retry** where **two handlers are attached to the same expression**. This is a very simple version of the *Lore* top-level generator. The method *loop* is designed to prevent a top-level exit after an error occurred. Signaling the exception *exit-top-level* forces the legal exit. The idea is to easily create new instances and new kinds of top-levels.

```
[exit-top-level isa exception-class superset error] ; In order to force exit.
[top-level isa class with
  (method prompt form ...)
  (method body form [t while [oself prompt] [[[standard-io read] eval] print]])
  (method quit form [exit-top-level signal]) ; The only way to quit.
  (method loop form
     [((exit-top-level form [xself exit 'bye]) ; Catches exit-top-level and force exit.
       (exceptional-event form [xself default-handles])) ; a simplified version of default-handling.
       protect ; Attaches the two handlers to the following expression.
       [t while [oself body]] ] ) ] ; Infinite loop until an exception occurs.
```

The method **default-handles** may use **retry** to cause a restart (cf. the following box). It is now very simple to create subclasses and override methods *prompt* or *body*, e.g. "*[debugger isa class superset top-level ....]*". New instances are obtained and invoked by typing for instance "*[debug isa debugger]*" and then "*[debug loop]*".

• **Automatic creation of dynamic handlers.** A problem appears when trying to implement the encapsulator example since we have to attach a handler to the class *encapsulator* (cf. § 3.2). A **class-handler** cannot be used since such handlers are default handlers and may be overriden by a dynamic one; consequently, we cannot ensure that exception *unknown-selector* will be handled as it should to fulfil the requirement.

In order to solve such problems, a primitive has been designed wich takes a class argument and ensures that an expression handler will be dynamically attached to each message toward an object of that class. Enabling this primitive costs one slot access for each message sending, to check whether a handler is defined on the receiver's class or upper classes and has to be attached to the transmission. This access can generally be computed at compilation time.

• **Defining default handlers.** All static handlers are standard methods named **default-handles**. The most general ones are defined on the exceptions and have no parameters. Those attached to classes must be defined with a parameter the type of which is the exception to be handled. To look for default-handlers, the system first sends, to the signaling-time active object, the message **default-handles** with the instance of the current exception (**xself**) as argument. The *Lore* message passing mechanism automatically selects the appropriate handler if one exists. If not, the system sends the same message to **xself**.

As an example, let us consider the most general default-handler defined on **exceptional-event**.

```
[exceptional-event has method default-handles
  form [oself describe-error] ; provides information about the error
       [oself display-propositions] ; displays interactive propositions (cf. § 5.7)
       ; If no proposition is chosen
       [xself retry]] ; Executes the top-level actions again, (cf. the above box)
```

5.6. When-exit or how to restore contexts whatever happened.

One problem with the resumption model is the restoration of valid contexts. Indeed, signalers as well as handlers that propagate exceptions do not know whether there will be resumption or termination. Thus, they do not know whether they have to perform restorations since resumption may occur (cf. "cleanup handlers" [Goodenough 75]). To prevent making association syntax and our model implementation more complex, we provide the primitive **when-exit** allowing to attach restoration actions to expressions. Whatever happens, these restorations will be performed when the expression execution will end. **When-exit** returns the expression value.

As an example, consider the method **signal** which creates an instance of the signaled exception and destroys it after it has been handled, in resumption cases as well as in termination cases.

```
[exception-class has method signal ....
  form [xself as [oself new ....] ; instanciation of the current exception
          [[xself search-and-invoke-handler] when-exit [xself kill]] ] ]
```

5.7. Interactive propositions.

We provide a *Lore* implementation of interactive propositions. These are simpler to define and to use than the *Flavor* one [Moon 83]. The principle consists in storing, within exceptions, correction propositions and correlated actions to be executed when a proposition is chosen. The propositions can be displayed one at a time or all together (as done by the default-handlers). "Sub-exceptions" inherit from propositions defined on upper exceptions.

```
> [1 + [5 foo]]
*** <Unknown-Selector> : foo is not a selector for receiver 5.
*** In the <method> <+.number> while sending + to the <integer> 1
------- 1 : Supply a value to use as result of this message.
------- 2 : Supply a new selector.
------- 3 : Supply a new receiver.
>>> 2
Enter the new selector, please : factorial
121
```

Implementation is based on the ability of defining new kinds of methods (more precisely new sets of properties), and on the ability of defining slots on properties. We thus define a subclass of *method* called **propose-method** with a slot named **when-chosen**. This slot is designed to contain the method to be invoked when the proposition is chosen. In order to display all propositions, default handlers only have to look for and invoke all the properties of type **propose-method** in the current exception dictionary. Example : here is the creation of a proposition on exception **unknown-selector**.

```
[unknown-selector has propose-method ask-new-selector when-chosen 'new-selector
   form ["Supply another selector." print] ]

[Unknown-Selector has method new-selector
   form ["Enter the new selector, please : " print]
        [xself resume .....][6]
```

6. Conclusion.

Object oriented programming has led to many improvements in system specification, software quality and reusability. However most current object oriented languages do not take advantage of past studies about software fault tolerance.

We have presented a mechanism for exception handling in object oriented languages designed mainly to implement fault-tolerant encapsulations. Its originality lies first in the fact that we have paid attention to the specific characteristics of such languages : a method is able to react when an inner method invocation raises an exception, and a solution is provided to associate handlers with object and exception classes. Second, we provide a is a full object-oriented implementation of the main well known exception handling techniques coming both from procedural and object-oriented languages. Our system has been implemented within the *Lore* object-oriented language and is currently used in simulation and AI applications.

---

[6] Since it is possible to resume with a new receiver or a new selector, there must be a resumption protocol between the signaler and the handler.

This system has also been designed to be a powerful basis for an interactive debugging environment which is intended for detection, localization and correction of errors, i.e. of non-handled exceptions. It allows each debugging tool to handle exceptional situations. For example, the stepper handles exceptional situations so that users do not have to follow all the details of an execution (cf. [Lieberman 84]) to localize an error context. We have also taken advantage of the non-local exit and resumption possibilities to write a stack examiner in *Lore*. Any user could in the same way write his own.

Extensions are planned. It is yet possible, without significant modifications, to include, in our system, the possibilities of storing and accessing exceptional data through exception handling mechanisms, as described in [Borgida 86]. These features are a powerful alternative to ''mixin'' and method combination in order to describe inheritance hierarchies with exceptions. They make it possible to create *exceptional classes* or *exceptional-instances.*

Another possible extension lies in the search for handlers via the ''use'' relation (vs. the ''call'' relation) as presented in [Levin 77]. This would consist in looking for handlers attached to entities that use (vs. ''are currently using'')[7] the object within which the exception was raised.

Acknowledgements.

I have greatly benefited, while specifying this system, from the comments abd suggestions of Michel Bidoit. I wish to thank Francoise Carre, Yves Caseau and Olivier Danvy for commenting drafts of this paper, Francois-Xavier Testard-Vaillant for interesting discussions, Hélène Kawa and Catherine Jourdan for commenting english style.

References.

[Benoit, Caseau, Pherivong 86] C.Benoit, Y.Caseau, C.Pherivong : Knowledge Representation and Communication Mechanisms in Lore. Proc. of ECAI'86, Brighton, July 1986.

[Berry 85] D.M.Berry, S.Yemini : A Modular Verifiable Exception-Handling Mechanism. ACM Transaction on Programming Languages and Systems, Vol. 7, No. 2, pp. 213-243, April 1985.

[Bidoit 85] M. Bidoit et al. : Exception Handling: Formal Specification and Systematic Program Construction I.E.E.E. Transactions on Software Engineering, Vol. SE-11, Number 3, March 1985, pp.242-252.

[Bobrow 83] D.G.Bobrow, M.Stefik : The Loops Manual. Xerox Parc, 1983.

[Bobrow 86] D.G.Bobrow & al : Merging Lisp and Object-Oriented Programming. Proc. of OOPSLA'86, Special issue of Sigplan Notices, Vol. 21, No. 11, pp. 17-29, November 1986.

[Borgida 85] A.Borgida : Language Features for Flexible Handling of Exceptions in Information Systems. ACM Transactions on Database Systems, Vol. 10, No. 4, pp. 565-603, December 1985.

[Borgida 86] A.Borgida : Exceptions in Object-Oriented Languages. ACM Sigplan Notices, Vol. 21, No. 10, pp. 107-119, October 1986.

[Caseau 87] Y.Caseau : Etude et Réalisation d'un langage objet : LORE. *Thèse de l'université Paris-Sud*, Orsay, France, Novembre 1987.

---

[7]These entities may be inactive at raising time.

[Christian 82] F.Christian : Exception Handling and Software Fault Tolerance, IEEE Transactions on Computers, Vol. C-31, No. 6, pp. 531-540, June 1982.

[Cointe 84] P.Cointe Implémentation et interprétation des langages objets Applications aux langages Formes, Objvlisp et Smalltalk. *Thèse d'état*, Université Paris 6, IRCAM, France, Décembre 1984.

[Dahl 70] O.Dahl, B.Myhrhaug, K.Nygaard : SIMULA-67 Common Base Language. SIMULA Information, S-22 Norwegian Computing Center, Oslo, Norway, October 1970.

[Etherington 83] D.Etherington, R.Reiter : On inheritance hierarchies with exceptions. Proc. of AAAI-83, pp. 104-108, August 1983.

[Goldberg,Robson 83] A. Goldberg, D. Robson : SMALLTALK 80, the language and its implementation. Addison Wesley 1983.

[Goodenough 75] J.B.Goodenough : Exception Handling: Issues and a Proposed Notation. Communication of the ACM, Vol. 18, No. 12, pp. 683-696, December 1975.

[Ichbiah 79] J.Ichbiah & al : Preliminary ADA Reference Manual. Rationale for the Design of the ADA Programming Language. Sigplan Notices Vol. 14, No. 6, June 1979.

[Knudsen 87] J.L.Knudsen : Better Exception Handling in Block Structured Systems. IEEE Software, pp. 40-49, May 1987.

[Lieberman 84] H.Lieberman : Step Toward Better Debugging Tools For Lisp. ACM, Conference Record of the 1984 ACM Symposium on LISP and Functional Programming.

[Levin 77] R.Levin : Program structures for exceptinal condition handling. Ph.D. dissertation, Dept. Comput. Sci., Carnegie-Mellon University Pittsburg, June 1977.

[Nixon 83] B.A.Nixon : A Taxis Compiler. Tech. Report 33, Comp. Sci. Dept., Univ. of Toronto, April 83.

[Liskov 79] B.Liskov, A.Snyder : Exception Handling in CLU. IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, pp. 546-558, Nov 1979.

[Meyer 87] Reusability: The Case for Object-Oriented Design. IEEE Software, pp. 51-64, Mars 1987.

[Mitchell 77] J.G.Mitchell, W.Maybury, R.Sweet : MESA Language Manual. Xerox Research Center, Palo Alto, Calif., Mars 1979.

[Moon 83] D. Moon, D. Weinreb : LISP Machine Manual, Fourth Edition. MIT Artificial Intelligence Lab., Cambridge, Massachussets, (July 1981).

[Pascoe 86] G.A.Pascoe : Encapsulators: A New Software Paradigm in Smalltalk-80. Proc. of OOPSLA'86, Special issue of Sigplan Notices, Vol. 21, No. 11, pp. 341-346, November 1986.

[PL/I 78] Multics PL/I Reference Manual, Cedoc 68, Louveciennes, Septembre 1978.

[Stroustrup 86] B.Stroustrup : The C++ Programming Language. AT&T Bell Laboratories, Murray Hill, New Jersay. Addison-Wesley, March 1986.

[Testard-Vaillant 86] F.X.Testard-Vaillant : Exceptions and Interpreters. AIMSA'86, Varna, September 1986.

[Wertz 83] H.Wertz : An Integrated, Interactive and Incremental Programming Environment for the Development of Complex Systems. in Integrated Interactive Computing Systems, pp. 235-250, ED P.Degano & E.Sandewall, North-Holland 1983.