A Fully Object-Oriented Exception Handling System: Rationale and Smalltalk Implementation

Christophe Dony

Montpellier-II University - LIRMM Laboratory
161 rue ADA, 34392.Montpellier Cedex 05.
dony@lirmm.fr
http://www.lirmm.fr/~dony

Abstract. This paper motivates and presents the specification and the implementation of an exception handling system for an dynamically typed object-oriented language. A full object-oriented representation of exceptions and handlers, a meta-object protocol to handle using either termination or resumption makes the system powerful as well as extendible and solves various exception handling issues. Three kind of dynamic scope handlers (expression handlers, class handlers and default ones) make it possible to define reusable and fault-tolerant software modules. The implementation of the system is readable and simple to understand because achieved entirely in Objectworks Smalltalk, thanks to the reflective facilities of this language.

1 Introduction

The program structures for handling exceptional events [7] [12] [13] [2] [24] [9] have been designed to implement software entities able to return well defined and foreseen answers, whatever may happen while they are active, even though an exceptional situation occurs. The end of the 1970s saw the development of exception handling systems dedicated to procedural programming. All specifications have all been influenced by Goodenough's seminal paper [7]. Well known implementations include MESA [15], CLU [13] or ADA [8]. Exception handling systems have later been integrated into object-oriented languages at the end of the 1980s (Zetalisp+Flavors [17], CommonLisp(+CLOS) [19], Eiffel [14], Objectworks Smalltalk [21], C++ [11], or more recently in Java.

This papers presents an overview of the specification and implementation of an exception handling system initially conceived [3] for the Lore Object-Oriented Language and adapted to Smalltalk [4]. The key-ideas of this system are (1) to apply object-oriented design to the whole system, to define a reusable and open class library for exception handling allowing systems designers to reuse it to produce dedicated exception handling systems and (2) to take into account the specificity of object-oriented programming by integrating expression and class handlers allowing users to define functional, class-based or even component-based fault tolerant modules.

Implementations of exception handling systems are rarely presented because they are generally done at a low level (compilers, virtual machines) and hard to describe in the context of a paper. The implementation of this system is reasonably readable and

simple to understand because achieved entirely in Objectworks Smalltalk, thanks to the reflective facilities of this language. The main implementation issues detailed in the paper are: the internal representation of handlers, the algorithm for searching them - knowing that both resumption and termination are allowed, the achievement of termination and resumption, which takes into account some possible user-defined unwind protections. Our EHS has been specified and implemented as the same period than Objectworks Smalltalk's one and both share many similarities (except for class and default handlers) but none of their implementations have been published yet.

Section 2 recalls some definitions and introduces our notations. Section 3 presents our EHS specification and motivates the main choices. Section 4 presents the implementation. Point to point comparison with related works is scattered in the different sections. Sections 3 require that readers have a minimal knowledge of the Smalltalk syntax. Section 4 supposes a higher knowledge of that system but should be globally readable by anyone knowing object-oriented languages.

2 Definitions, Terminology, Notation

Software failures reveal either programming errors or the application of correct programs to an ill-formed set of data. An exception can be defined as a situation leading to an impossibility of finishing an operation. The problem of handling exceptions is to provide materials allowing to establish communication between a function or procedure which detects an exceptional condition, while performing an operation, and those functions or objects or modules that are clients of this operation (or have something to do with it) and wish to dynamically handle the situation. An exception handling system (**EHS**) allow users to signal exceptions and to associate handlers to entities (according to the language, an entity may be a program, a class, an object, a procedure, a statement or an expression). To *signal* an exception amounts to identify the exceptional situation, to interrupt the usual sequence of operations, to look for a relevant handler, to invoke it and to pass it relevant information about the exceptional situation, such as the context in which the situation arose. To *handle* means to set the system back to a coherent state. Handlers can usually choose, knowing about their definition context and using arguments provided by the signaler, whether to (1) transfer control to the statement following the signaling one (*resumption*), (2) discard the execution context between the signaling statement and the one to which the handler is attached (*termination*) or (3) signal the same or a new exception, which is generally called *propagation* and should be interpreted as a delegation of responsibility to a more general handler.

For the examples in this paper, we use two different syntax (cf. Figure 1) for handlers declaration and definitions: firstly a general one inspired of what can be found in procedural languages, secondly the Smalltalk syntax used in our system.

```
        "General Syntax":
{protected-instruction1; ...; protected-instructionN;
    {when exception (parameter) do
        {handling-instruction1; ...; handling-instructionN;}}}

        "Smalltalk Syntax":
[protected-expression1 ... protected-expressionN]
    when: exception:
    do: [parameter | handling-instruction1 ... handling-instructionN]
```

Fig. 1. Syntax for associating handlers to instructions.

3 Specifications

This section discusses the main issues related to the design of an EHS in a non concurrent context, explains our choices and presents the specifications of our system.

3.1 Resumption and Termination: a Dual Model

Choosing which control structures are available to write handlers bodies is one of the first crucial decision to be taken when designing an EHS and impacts the whole specification and implementation. Most exception handling systems only propose the termination model, others propose both termination and resumption (let us call this the **dual** model) and a few ones only propose resumption [12]. The respective merits of termination and resumption have already been widely discussed in many papers, e.g. [7, 13]. Let us just recall that to forbid termination is a very specific choice because many exceptions are really fatal. To forbid resumption is a way to produce EHS simple to use and to implement, although reducing the expressive power since some exceptions are really proceedable in some contexts. The resumption model is indeed more expensive in computation time and space, more complex to implement (see section 4) and also makes program static analysis more complex. It is however useful and time-saving in any application in which proceedable exceptions are raised, especially in interactive application in which users or operators, can choose a solution to recover from an exceptional situation. For example, interactive WEB applications can take benefit of resumption to restart calculus after a network interruption.

3.2 Handlers Scope and Fault-Tolerant Encapsulations

The scope of handlers, and as a consequence the way they are searched when an exception is signaled, determines what kinds of fault tolerant modules are offered by a system. The issue is the same whatever kind of modules are considered, classes, methods, etc.

Lexical Scope Handlers. Lexical scope handlers are by definition accessible when located in the program textual part in which they are lexically visible. A handler for an

exception raised within an inner module, if not found locally, is searched in a lexically englobing one. Lexical scope handlers allow users to define modules that handle all exceptions raised within their scope, they allow to check statically which handler will be invoked for each exception signaled within the module. Their main drawback is that exceptions are never propagated to modules clients.

Various systems (e.g. Beta [9] or Smalltalk-80 [6, p.102]) provide lexical scope handlers. As an example, let us consider the standard Smalltalk-80 ones (cf. Figure 2). The Smalltalk historical EHS specification only uses the language standard constructions; this is the same for Beta and this is one of the great advantages of this static approach. Exceptions are signaled within methods by sending to *self*, a message (e.g. *error:*). Handlers are standard methods defined on classes (let us call them class handlers) and are invoked by a standard message sending: they can only be found in the class (or one of its super-classes, they are of course inherited) in which the signaling method is defined. This means that, as far as exceptions are never propagated to operations callers, a method has no way to regain control, either to hide the occurrence of an exception (modularity) or to execute some recovery actions, when one of the methods it has invoked failed. We come back in section 3.5 on the potential interest of class handlers in term of expressive power and reusability.

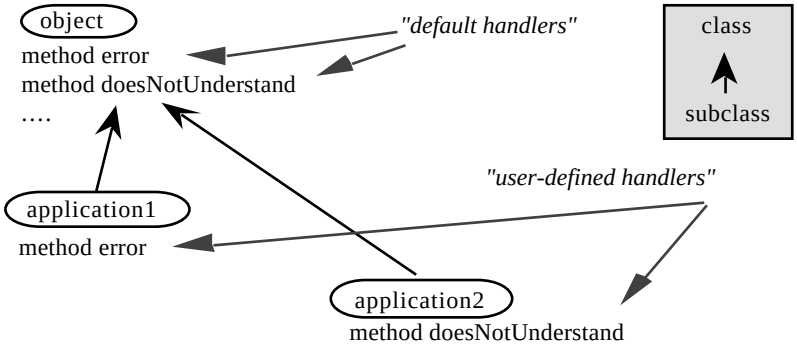

Fig. 2. Smalltalk-80 lexical scope class handlers

Dynamic Scope Handlers. On the other hand, dynamic scope handlers are searched in the execution stack when an exception is raised. They allow standard and exceptional responses to be specified in software modules interfaces. For a given module, internal exceptions are those handled internally, and external exceptions (sometimes called failures) those propagated to the module's clients. Dynamic scope handlers allow clients to retrieve control when the modules they use fail and to give execution context dependent answers to these failures. The semantic of the signaling process is then to transfer the problem from a place where it can only be noted to a place where it could be interpreted. The following example illustrates the first idea: the client module, a procedure *process-yield* correctly encapsulates its private's use of its a list by trapping the exceptions possibly raised by the list manipulation and by propagating an exception (*InactiveProcess*) of the same conceptual level than the achieved service. The second idea (giving caller context dependent answers) is illustrated the *pgcd* example. This is just a toy example, more expensive than the standard version without exception handling, but short enough to be presented here.

```
procedure process-yeld (a-process)
    {remove(Active-process-list, a-process);
        {when itemNotFound(e) do
            {signal (InactiveProcess)}}}

function pgcd (int a, b)
    {loop {aux := a;  a := b  ;  b := modulo(aux, b)};
        {when division-by-zero(e) do {exit(aux)}}}
```

All handlers in our system have a dynamic scope. Most of earlier EHS for procedural languages such as *PL/I*, *Clu*, or *Mesa*, and recent ones, for object oriented languages (Clos, C++, Java) are based on a stack-oriented research of dynamic scope handlers (limited to one stack level in CLU). New evolution of the *Beta* EHS integrates such handlers [10].

3.3 Status of Exceptions

The next issue to be discussed is the status of exceptions and of exceptional events. How are exceptions represented and referenced? How can they be manipulated or inspected?

Exceptional Events as First-Class Objects. The idea that consists in representing each conceptual exception by a class and each of its concrete occurrences (what we call exceptional events) as an instance (an exception object) of that class can initially be found in Taxis [18], *Zetalisp* [17] or [1] and is now almost a consensus; all todays object oriented systems have integrated it. Let us shortly recall its main interests.

- Exceptions can be organized into a knowledge sharing inheritance hierarchy.
- It is possible trap different events with a single handler.
- Signalers can communicate with handlers [16] pass to handlers the instance of the signaled exception which holds in its slots all the information about the exceptional situation.
- New user-defined exceptions can be created as subclasses of existing ones. There is no distinction between system and user-defined ones, all can be signaled and handled in the same way.

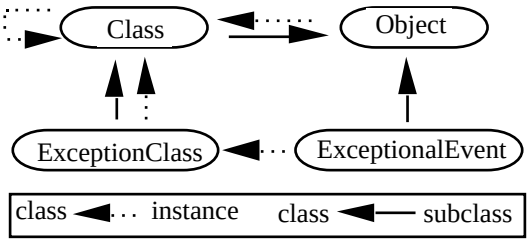

Fig. 3. Kernel exception classes.

Exceptions as First Class Entities. The systems that pioneered the above idea did not brought it to its limits; for example in Zetalisp, signaling and handling primitive are not standard methods invocable by sending messages to exceptions

objects. We have extended the above idea towards a complete object-oriented representation of all entities composing the EHS and towards an EH meta-object protocol to handle exceptions. Another language in which similar ideas can be found is Objectworks Smalltalk. The first step in that direction has led us to make conceptual exception first class entities by defining exception classes as instances of a metaclass. The Figure 3 shows the two kernel classes of our specification for what concerns exceptions representation. Each occurrence of an exception is an instance of a subclass of the class *ExceptionalEvent* that holds basic protocols for handling All exceptions classes are instances of the meta-class *ExceptionClass*[1] that holds basic protocols for signaling and are subclass of *ExceptionalEvent.* The next sections detail the advantages of that organization.

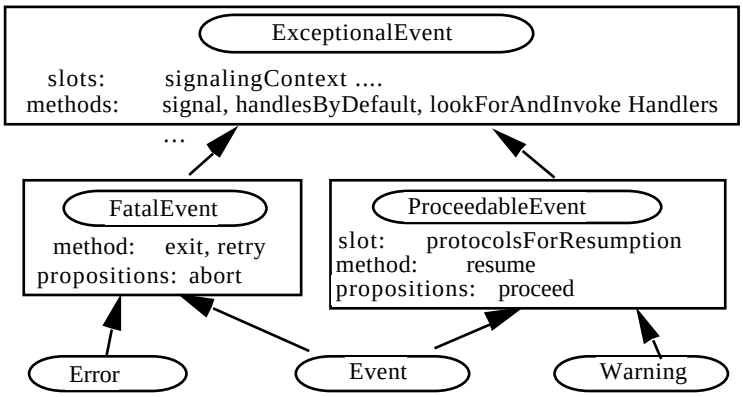

Fig. 4. Basic exception classes, associated attributes and methods.

3.4 Basic Primitives

The dual model of exception handling imposes that primitives for termination and resumption be available to write handler bodies. All our basic primitives to handle exceptions (*exit* and *retry* for termination, *resume* for resumption and *signal* for propagation) are implemented by standard methods defined on a set of kernel exception classes (cf. Figure 4) and constitute a meta-object protocol (following the CLos definition of term) for exception handling.

Kernel Exception Classes and Basic Handling Primitives. *ExceptionalEvent* is then divided into *FatalEvent,* to which are attached termination primitives, and *ProceedableEvent* to which are attached those for resumption. The slot *signalingContext* is to be dynamically bound at each occurrence of an exception to the signaling context. The slot *propositions* (instance variable of the meta-class) is used to store for each exception some propositions for interactive handling as initially

[1] When explicitly manipulated, meta-classes are *Class* subclasses. In our *Smalltalk-80* implementation, *ExceptionClass* is implemented by the automatically created meta-class *ExceptionalEvent class*. Each exception class has its own (automatically created) meta-class subclass of *ExceptionalEvent class*.

proposed in Zetalisp. From the user's viewpoint, the system is then based on three predefined exception classes.

- *Error* is the class of exceptional events for which resumption is never possible whatever the context in which the event is signaled.
- *Warning* is the class of exceptional events for which the termination is impossible.
- Finally, multiple inheritance is simulated to create the exception-class *Event* in order to allow both capabilities.

Basic Signaling Primitive. Within EHSs supporting the dual model, a set of primitives is generally provided to support the various signaling cases. E.g., in Goodenough's proposal, signaling with *escape* states that termination is mandatory, *notify* forces resumption and *signal* lets the handler responsible for the decision. In our system, *signal* is the single basic signaling primitive because knowing whether the signaled exception is proceedable or not only depends of its type (its position in the exception hierarchy). To signal an exception amounts to send the exception class the message *signal* whose corresponding method is defined on *ExceptionClass* (i.e. on *ExceptionalEvent class* in the Smalltalk implementation). *Signal* creates the "exception object" and assigns its slots with, on the one hand values given by the signaler and on the other hand, values owned by the system (e.g. *signalingContext*), cf. Figure 5 for an example. *Signal* finally sends to the initialized instance the message *lookForAndInvokeHandler* (cf. Figure 4), which will find and invoke a handler.

3.5 Additional Primitives and Control Structures

Unconditional Restorations. The dual model raises various issues that require additional primitives. The first issue is the restoration of coherent program states. Any method has to ensure that it will leave data, memory and resources in a coherent state whatever happens during its execution. A first solution to that problem, found in many systems is to give programmers the ability to define handlers that trap all exceptions to re-signal (propagate) the trapped one. The following example illustrates that solution, in a procedural-like syntax with the classical file example. It also highlights the fact that this solutions forces programmers to write restoration actions twice, once for normal and once for exceptional exit.

```
File f;.
{open(f); workOn(f);. close(f);
    {when any-exception (e) do
        {close(f); signal(e);}}}
```

However, that solution does not work properly in the dual model because a later handler may entail resumption and put the system back into an incoherent state in which f would be closed but should not be. To avoid writing more complex handlers and the duplication of the *close* instruction, an ad-hoc. primitive (cf. cleanup handlers in [7] or Lisp's *unwind-protect*) is necessary to allow users to write unconditional restoration statements executed whenever the procedure's related stack frame is really discarded. The exception handling system has to take this primitive into account while performing termination, by executing in the right order and in the right environment the restorations (cf. Section 4.5). In our system, the file example can thus be written as follows:

```
f := File new.
[f.open. f.workOn.]
    unwindProtect: [f.close]]
```

Cooperation for Resumption. Resumption raises another important issue: it should not be achieved without the agreement of both the signaler and the handler when, although the handler is responsible for saying what to do, the effective computation restart is performed by the signaler in its environment. In any cases, the signaler should be able to predict which kind of restarts he is ready to achieve. A slot named *protocolForResumption*, defined on *ProceedableEvent* provides a basic solution to this problem[2]. The signaler can use it to indicate, at signaling time, the options among which a handler can choose in order to achieve resumption. Assigning it to *nil* means that resumption is impossible. In conjunction, a handler wanting to entail resumption has to indicate which protocols it has chosen. In the figure 5 example, handlers can for example use the message *messageValue:* which itself entails resumption by using the correct protocol.

```
Event subclass: #DoesNotUnderstand
    instanceVariableNames: 'messageReceiver messageSelector messageArgs '
    methodsFor: 'handling'
        messageValue:                    "resume with a message value ..."
        newSelector:                     "resume with a new selector ..."
        newReceiver:                     "resume with a new receiver ..."
.....
Signaling the exception with propositions for resumption
result :=
    DoesNotUnderstand
        signalWithProtocolsForResumption:
                #(supplyValue newReceiver newSelector)
            messageReceiver: anObject
            messageSelector: aSymbol
            messageArgs: anArray.
"if control returns here, result is tested and the corresponding actions performed"
```

Fig. 5. Example of definition of a new exception, to represent runtime message sending failure. Its occurrences are either proceedable or fatal; thus it is a subclass of *Event*. While signaling it, the signaler can pass arguments to handlers to indicate which kind of resumption it is ready to achieve.

3.6 Various Kind of Handlers

This section deals with issues related to handler definition and shows how to create various kind of handlers in our system. It is first classical to associate handlers to pieces of code (expressions, blocks, procedure or programs). Besides, different researches have investigated the idea of associating handlers with data structures [12]

[2] [19] has proposed for this problem a more sophisticated solution: some new dedicated control structures (e.g. *restart-case*) provide a user-friendly way (with a case-like syntax) of writing code such as the one in the example and allow users to dynamically create new options for proceeding.

thus controlling exceptional situations arising when manipulating them. Within an object-oriented language, it is also natural to wonder whether it is interesting to associate handlers to objects or to classes and with which semantics Finally many solutions have been proposed in existing systems to store the most general, execution context independent, default handlers

Expression handlers. For what concerns pieces of code, we offer the possibility to associate handlers to any kind of Smalltalk expressions. This is done by grouping the expressions into a lexical closure[3] (a block in Smalltalk) and by sending this block the message *when:do.* The first argument <*exception-name*> is the exception to be trapped and the second one is the handler. Handlers have one parameter bound at handling-time to the current exception object.

```
[<protected expression>]
    when: <exception-name>
    do: [<handler parameter> | <handler body>]
```

Class Handlers. To associate handlers with individual objects is not compatible with the class-based model in which all instances of a class have to share the same behavior. Besides, we have quoted in section 3.2 the existence of handlers associated with classes in Smalltalk-80 and in Beta. We propose to define equivalent handlers but to give them a dynamic scope. A class handler associated with a class C for an exception E will thus be able to trap all occurrences of E raised anywhere during the execution of any method of C or of C's subclasses. Such class-handlers allow programmers to control which exceptions can be propagated outside of any methods of a class, to control for example, that *overflow* and *EmptyStack* are the only exceptions that can be propagated outside of any method defined on the class *Stack*. These class handlers also induce original reusability schemes based on inheritance. Consider again the class *Stack*. Now suppose that a class of stacks that are able to grow is needed. A simple solution to this consists is creating a *Stack* subclass named *GrowingStack*, on which is defined a handler for *overflow* and a method *grow*, this handler can resume the interrupted method, whatever its name and its location, after having grown the stack buffer. Class-handlers have been widely used in this way in Smalltalk-80 extensions to modify message sending, e.g. to implement *Encapsulator* or to implement asynchronous message sending in the Briot's *Actalk* System.

A few systems provide dynamic scope class handlers (see. e.g. [22]). In our system, class handlers can be attached to any class by using the method *when:do:*, defined on *ClassDescription*, with the following syntax:

```
<protected class>
    when: <exception-name>
    do: '<handler parameter> | <handler body>'
```

The first argument is the exception to be trapped and the second one is a string. This method *when:do:* first calls the Smalltalk compiler to compile the handler string in the environment of the protected class so that instance and class variables defined on the class can be accessed. Then it inserts the created handler in the handler collection of the class and of its subclasses. For each class, class-handlers are ordered compared to

[3]This is the price to pay to implement handler definition by a message sending.

the exceptions they are defined for. Class-handlers cost nothing while exceptions are not signaled; they only are taken into account at signaling time.

Default Handers. Interesting but somehow semantically complex propositions to define default handlers at various program levels can be found e.g. in [20]. We have chosen a simpler point of view in which default handlers are considered as the place where the most general information regarding how to handle an occurrence of an exception, independently of any execution context, should be stored. We propose to associate them to exceptions by defining them as methods (named *defaultHandle*) defined on exception classes. The system most general default handler is defined on *ExceptionalEvent* and can be overridden in subclasses, each exception can thus own its specific default handler. Default handlers are invoked by sending the message *defaultHandle* to the exception object as shown in the top-level loop example (cf. Section 3.7). We have integrated the idea of interactive propositions found in Zetalisp, which exploits exception hierarchies. A proposition is a couple of two method names, one to display a string and one to execute a corresponding action. Propositions are stored for each exception in the slot named *propositions* defined on *ExceptionClass* and displayed when the most general default handler is invoked.

3.7 Writing Handlers Bodies in a Generic Way

All kind of handlers can use the same primitives in the same generic way to put the program execution back into a coherent state. Genericity first means that neither programmers nor implementors have to perform tests to ensure that operations incompatible with the signaled exception will not be invoked - Note that his rule is violated for resumption where the slot *protocolForResumption* is tested by the system. For example, any attempt to send the message *exit:* to an object which is not a *FatalEvent* will fail. Genericity also means that the operations relevant to the current exception object will automatically be selected even though an abstract (multiple) exception has been caught.

Termination Examples. Sending to the exception object the message *exitWith:* entails termination. The execution stack frames located between the signaler and the handler are discarded while recovery blocks are executed. The argument's value becomes the value returned by the expressions to which the handler is attached. Here is our system's version of the above function *pgcd*, now defined on class *Integer*, that uses termination.

```
"computes the pgcd of a and b"
    [[true] whiletrue: [aux := a.    a := b.    b := aux modulo: b]]
        when: division-by-zero
        do: [:e | e exitWith: aux]
```

For what concerns a class-handler, which is invoked when an exception is about to be propagated outside of a method *C* defined on a protected class, termination ends C's execution and the exit value becomes the value returned by C's invocation. The following example is an implementation of the growing stack example described in Section 3.6, it highlights the interest of the *retry* primitive.

```
GrowingStack
    when: Overflow
    do: [:anOverflow | self grow.
                       anOverflow retry].
```

Finally, default-handlers are conceptually attached to the program main procedure (or to the top-level loop in an interpreted environment), thus termination in a default-handler ends the program execution (or returns to the top-level). Here is an example of applying termination that uses both *exit* and *retry* , to implement a top-level loop. The only way to exit the loop is to signal the exception *LoopExit*. If any other exception is trapped, its default handler is invoked and finally, whatever it does, the loop is re-entered.

```
[[true] whileTrue: [((aStream.read).eval).print]]
    when: LoopExit
    do: [aLoopExit: | aLoopExit exitWith: #bye]
    when: ExceptionalEvent
    do: [anExcEvent: | anExcEvent defaultHandle.
                       anExcEvent retry]
```

Resumption Example. Sending to the exception object the message "*resumeWith: <aResumptionOption> with: <aValue>*" entails resumption. The couple *<option, value>* becomes the value returned by the method *signal* provided that the option belongs to the *protocolsForResumption* collection of the signaled exception.

```
[anObject aMessage]
    when: DoesNotUnderstand
    do: [e: | e resumeWith: #SupplyValue with: 33]
```

Propagation Example. Signaling a new exception or propagating the trapped one can be done by sending the message *signal* either to a new exception (a class), or to the exception object. Here is an illustration with the previously described *stack* examples (cf. Section 3.5) which shows how to control, with class handlers, which exceptions will be propagated outside of any methods defined on *Stack* or on its subclasses. The second handler for *ExceptionalEvent* traps all exceptions, except *Overflow* and *EmptyStack*, and propagates *StackInternalException.*

```
Stack
    when: #(Overflow EmptyStack)
    do: ':exceptionObject | exceptionObject signal'          "propagation"
    when: ExceptionalEvent
    do: 'exceptObject: | StackInternalException signal'   "new exception signaled"
```

4 Implementation

This section describes some key-points of the above specification implementation which is entirely achieved in *Objectworks Smalltalk* without any modification to the virtual machine. This has been possible thanks to the reflective capabilities offered by this programming environment, particularly because methods, lexical closures and the

execution stack are or can be made first class objects. The main focus is put on the representation of exceptions, of handlers and on the signaling algorithm taking into account expression, class and default handlers within a context in which both resumption and termination are allowed. The interest of this section is to describe this algorithm in its real implantation context.

Objectworks Smalltalk EHS specification has considerably evolved since the blue book specification and share many common points with our specification, except for what concerns class handlers and less importantly interactive propositions. The implementation of Objectworks EHS, as far as I know never described in any paper, also shares common point with ours but is more efficient because part of it have been moved to the virtual machine. In particular, stack frames are no more reified but are accessed at the virtual machine level.

4.1 Exception Classes

The class *ExceptionalEvent* (cf. Figure 6), the root of our exception classes hierarchy, declares four instance variables, three of them are of interest here. *SignalingContext* is used to store the stack frame (we will frequently call stack frame "contexts" because Smalltalk objects that represent them are called "contexts objects") in which the exception has been signaled. *HandlerContext* is used to store the context in which a handler is found. *ErrorString* allows users to pass a string argument to handlers. *ExceptionalEvent* also declares different class variables, four of which (*BottomStackMethod*, *HandleMethod*, *InvokeHandlerMethod* and *UnwindMethod*) designed to store references to particular methods addresses that will be used during handler research and invocation.

```
Object subclass: #ExceptionalEvent
   instanceVariableNames: 'errorString signalingContext handlerContext private '
   classVariableNames:    'BottomStackMethod HandleMethod InvokeHandlerMethod
                           RetryMark UnwindMethod '

initialize                                   "defined on ExceptionalEvent class"
   HandleMethod := BlockClosure compiledMethodAt: #when:do:.
   UnwindMethod := BlockClosure compiledMethodAt: #unwindProtect:.
   BottomStackMethod
      := SmalltalkCompiler compiledMethodAt: #evaluate:in:to:notifying:ifFail:.
   InvokeHandlerMethod := ExceptionalEvent compiledMethodAt: #invokeHandler:with:.
   RetryMark := #().
```

Fig. 6. *ExceptionalEvent* class (detail)

Note for example that the method *when:do:* allowing users to define expression handlers is represented by a Smalltalk object that can be retrieved by sending to the class *BlockClosure*, on which the method is defined, the message c*ompiledMethodAt:*. *SpecialMark* contains a unique mark used to implement the *retry* method Finally, this class also defines an interactive proposition named *askForRetry*.

4.2 Status and Storage of Handlers

Default handlers are standard methods defined on exceptions classes under the selector *handlesDefault*. They do not raise any structure or storage problems.

Handlers associated to expressions. Handlers associated to expressions have to be executed in the environment in which they have been created. The resumption model forbids destroying the stack frames located between the signaler frame and the handler frame in order to allow the calculus to be eventually restarted. To invoke a handler thus supposes to go back to a previously defined environment without destroying the execution stack. A first solutions to this problem is to copy the stack at signaling time (e.g. with an equivalent of the scheme *call-cc* primitive), to destructively search a handler, to execute the handler in its context, now located on top of the stack, and finally to replace the current stack by the copy made at signaling time. We have neither implemented this solution nor seen it implemented. A second solution is to execute the handler on top of the stack but in its definition context, i.e. in such a way that free variables of the handler get their value and are assigned in the handler definition context. This supposes that handlers be lexical closures.

The method *when:do:* to associate handlers to expressions is defined (cf. below) on the class *BlockClosure* that represents lexical closures in Smalltalk. The receiver (*self*) is a block containing the protected expressions. The method simply sends *self* the message *value*, which entails the execution of the protected expressions. If an exception is raised during this execution, the system will find the handler as the second argument (*handlerBlock*) stored in the stack frame created by the method *when:do:*.

```
when: exception do: handlerBlock                    "Defined on BlockClosure"
    ^self value
```

Handlers Associated to Classes. Handlers associated to classes are some kind of compiled methods, of which they inherit the basic structure, the specific part of their structure being described by the *ClassHandler* class (see below). The instance variable *domain* stores the class on which the class handler has been defined, for example, the class *GrowingStack* for our Section 3.6 example. The *event* instance variable stores the exception for which the class is protected (*Overflow* in our example). The instance variable *receiver* is to be bound at handler invocation time to the object that is active when the trapped exception is raised.

```
CompiledMethod subclass: #ClassHandler
    instanceVariableNames: 'event domain receiver '
    classVariableNames: 'SortBlock '
```

Class-handlers are stored for each class into a sorted collection from the most specific to the most general. For that purpose, we have added an instance variable named *classHandlerSet* to the kernel class *ClassDescription* with defines the basic structure of all Smalltalk classes. Albeit they have a compiled method status, class-handlers are not stored in the classes method dictionary for three reasons: they should not to be directly invoked by users, they are not connected to external selectors and they have to be stored in a specific order.

4.3 Signaling

In its simplest form, signaling simply consists in creating the exception object, an instance of the class that receives the *signal* message, in initializing its fields and in sending to the exception object a message to look for handlers. See next section for an explanation about the *thisContext* variable.

```
signal                                  "defined on ExceptionalEvent class"
    ^self new initialize signal

signal                                  "defined on ExceptionalEvent "
    signalingContext := thisContext.
    ^self lookForAndInvokeHandler
```

4.4 Handler Search

This section describes the method that looks for a handler after an exception has been raised. A simplified version is primarily presented. The complete version is described afterwards.

Accessing the Stack. As far as handlers are searched into the stack, the first issue is to access it. The Objectworks environment is able, when asked, to represent the execution stack frames as first-class objects, instances of various subclasses of the *Context* class. This is powerful example of reflection because that object can not only be viewed but also modified. Modifying the slot sender of such an object effectively produces a non local jump when returning from the method in which the modification is done. At any time during a computation, creating the object representing the current stack frame can be done by accessing the read-only variable called *thisContext*. Its slots contain all the information needed to implement our system:
- the receiver of the message the execution of which has created the frame,
- the method that has been invoked as a consequence of this message,
- the sender of the current frame, i.e. the calling frame lower in the stack.

A simplified Algorithm for Finding Handlers. Figure 7 describes a simplified algorithm that search a handler without taking into account exceptions signaled within handlers. When a handler is found, it is immediately invoked.

The implementation of the simplified algorithm. Figure 8 shows the method implementing this simplified algorithm. The *lookForAndInvokeHandler* method is defined on *ExceptionalEvent* and invoked while signaling by sending this message to the exception object. Within this method, *self* is the current exception object for which a handler is searched.

- The search starts at the signaling context sender. The signaling context is retrieved in the *signalingContext* instance variable of the exception object (Figure 8, line 1). *UnwindContexts* is a local variable used to monitor the collection of recovery actions found while going down the stack.

- A loop is entered (2) and will be exited, by returning ("^" is the smalltalk's return instruction.) the value of the method *invokeHandler:with:*; control never returns to the instruction following the *invokeHandler:with:* message sending.

```
Let E be the signaled exception,
Let F initially be the sender of the stack frame in which the message signal has been sent,
Let UnwindContexts be an empty ordered collection.

L:  If F is the bottom of the stack frame
    then invoke default handler for E
    else   If the frame F has been established by an invocation of the method when:do:
           and if the associated handler H traps the exception E,
           then invoke H.
           else   let C be the class of the receiver of the current frame method.
                  If a handler H for the exception E is defined on C,
                  then invoke H.
                  else if F has been established by an invocation of unwindProtect:,
                       then  append the argument to UnwindContexts end-if
                       let F be F 's sender (stack previous frame) and goto L.
                  end-if
           end-if
    end-if
```

Fig. 7. A simplified version of the algorithm to search a handler.

```
lookForAndInvokeHandler
        "defined on ExceptionalEvent"
  | currentContext classHandler method |                          "local variables"
  currentContext := signalingContext sender.                                  (1)
  unwindContexts := OrderedCollection new.
  [true] whileTrue:                                                           (2)
    [method := currentContext method.
    "if the bottom of the stack is reached, invoke default handlers"
    (method == BottomStackMethod)                                             (3)
      ifTrue: [handlerContext := currentContext sender.                       (4)
               ^self invokeHandler: [:e | e handlesDefault] with: unwindContexts]   (5)
      ifFalse: [
    "looking for an expression handler"
        (method == HandleMethod and:
                    [self isKindOf: (currentContext localAt: 1)])             (6)
          ifTrue:
            [handlerContext := currentContext.                                (7)
             ^self invokeHandler: (currentContext localAt: 2)
                         with: unwindContexts]                                (8)
    "looking for a class handler in the class of the current context receiver"
        classHandler := currentContext receiver class isProtectedFor: self.   (9)
        classHandler isNil ifFalse: [
          handlerContext := currentContext.                                   (10)
          classHandler receiver: currentContext receiver.                     (11)
          ^self invokeHandler: classHandler with: unwindContexts]             (12)
    "no handler here, but check if this context contains an unwind blocks"
        (method == UnwindMethod)                                              (13)
          ifTrue: [unwindContexts addLast: (currentContext localAt: 1)].      (14)
    "no handler here, going down one frame"
        currentContext := currentContext sender                               (15)
    ] "end of ifFalse: method == BottomStackMethod"
  ] "end of the while loop
```

Fig. 8. An implementation of the simplified algorithm in Fifure 7.

- When a handler is found, this method (cf. section 4.5) is called (lines 5,8,12) with the handler as first argument and the unwind blocks collection monitored during the search as second argument. The handler is either a block or a kind of compiled method. The stack frame in which it is located is stored in the *handlerContext* slot of the exception object (4, 7, 10) and will be used to achieve handling.

- The test in (3) is true when the bottom of the stack is reached. This means that no expression or class handler has been found. A default handler is invoked by sending the message *handlesDefault* to the exception object (5).

- The test in (6) is true if the current frame method is *when:do:* and if the exception object is an element of the class a reference to which is stored in the first argument of the method (arguments of the current frame method can be accessed by sending the message *localAt:* to the context object). This means that an expression handler has been found. Its body is stored in the second argument of the *when:do:* method(8).

- The method *isProtectedFor:* sent to the class C of the receiver of the current frame method, returns either a class-handler for the current exception if one is present on C or nil. In the first case, the handler is invoked (12). Before that, the receiver in the method that raised the exception is stored in the class handler's *receiver* slot (11).

- Lines 13 and 14 deal with unconditional recovery actions defined in unwind blocks. If only termination was supported, this would be the place to execute these actions. Supporting the dual model (termination and resumption) imposes to monitor all unwind blocks found between the signaling context and the handler context, to pass that collection to the handler and to execute them if the handler entails termination.

- In (15), no handler has been found in the current frame, the loop's body is entered again with the variable *currentContext* pointing to the previous stack frame.

The complete version taking into account exceptions signaled within handlers. The real algorithm is more complex since it supports the dual model (termination and resumption) which imposes that handlers be executed while the signaling context has not been destroyed. The algorithm has thus to ensure that, when an exception is signaled within a handler (either an expression, class or default one), the new search starts just below the frame in which the current handler has been found, thus preventing its recursive invocation. Signaling the exception *InactiveProcess* in the handler for *ItemNotFound* is an example of such as situation as shown in Section 3.2. Figure 9 presents the complete version of the method that looks for and invoke handlers.

- The test in line 9 determines whether the current frame has been created by the invocation of *invokeHandler:with:*. If true, this means that the current exception has been raised within a handler and execution continues in line 9. Otherwise the standard algorithm described in the previous section is executed (line 21).

- In line 9, let e2 be the current exception object and h1 be the handler that as been invoked to trap the first exception e1. The current context objet represents the frame created by the invocation of h1; its *receiver* slot contains the exception object e1.

- It is first necessary to update the *unwindContexts* collection by concatenating ("," is the concatenation operation) (lines 11 and 12) the current recovery action collection to the collection collected during the search for h1 which is stored in the stack as the second argument of the current context method. When found, the handler h2 for e2 will have in hand the whole set of recovery actions found between e1 signaling frame and h2 definition frame, to be executed if h2 entails a termination.

```
lookForAndInvokeHandler                                   "defined on ExceptionalEvent"
  | currentContext classHandler method unwindContexts dejaVus |
  unwindContexts := OrderedCollection new.                                        (2)
  currentContext := signalingContext.                                             (3)
  [true]  whileTrue: [                (4)
    method := currentContext method.                                              (5)
    (method == BottomStackMethod)                                                 (6)
      ifTrue: [handlerContext := currentContext sender.                           (7)
            ^self invokeHandler: [:e | e handlesDefault] with: unwindContexts.].  (8)
    "Detection of exceptions signaled within handlers"
    [method == InvokeHandlerMethod                                                (9)
      ifTrue: [ "The current exception has been raised within a handler"          (10)
        "a) dealing with unwind-protections"
        dejaVus := (currentContext localAt: 2) copy.                              (11)
        dejaVus isNil ifFalse: [unwindContexts := unwindContexts , dejaVus].      (12)
        "b) jump to the handler context"
        currentContext := currentContext receiver handlerContext.                 (13)
        "c) Was the exception signaled within a default handler?"
        currentContext method == BottomStackMethod                                (14)
          ifTrue: ["Direct invocation of the most general default handler"         (15)
            handlerContext :=  currentContext.                                    (17)
            ^self invokeHandler:
              [:e | e basicHandlesDefault] with: unwindContexts]                  (18)
          ifFalse: ["search will continue at the handler context sender"          (19)
                currentContext := currentContext sender]]                         (20)
      ifFalse: [                                                                  (21)
    "same code than lines 6 to 15 in Figure 8"
```

Fig. 9. The complete version of handler search.

- The handler for e2 now has to be searched below the frame in which h1 has been defined and which is stored in e1's *handlerContext* slot. As specified by the instruction in line 13, the *currentContext* is assigned to h1 definition context.
- Before continuing the search in the previous frame (line 20) a special case has to be handled. If h1 is a default handler, its definition context is the bottom of the stack frame and in such a case, tested in line 14, the search is stopped and the most general default handler has to be invoked (line 18).

4.5 Handler invocation

The method to invoke handlers is shown in Figure 10. All handlers (either lexical closure or class-handlers) are invoked by receiving the *value:* message with argument the current exception objet (cf. Figure 10, line 5). Two marks are stored in the stack (lines 3 and 5) just below that invocation point using the method *mark:catch:* which is an equivalent of the classical lisp *catch* function.

Termination will be implemented by a non local exit to the mark named *#exit* and resumption by a non local exit to the mark named *#resume* (see next section). If termination is ordered, control reaches line 7. There, all unconditional restorations monitored during the handler search are executed by the method *fastUnwind:*. Then the current execution frame is assigned to the handler context (line 8); this effectively discards all stack frames between the signaling and the handling point. Finally, the

given exit value is tested (line 9). If this value equals the retry dedicated special mark, a retry has been ordered and the expressions to which the handler were associated are executed again (line 10), otherwise the exit value is simply returned as the value of these expressions (line 11). If resumption is ordered, control reaches line (12) and the resume value is simply returned as the value of the handler invocation. It is the responsibility of the signaler to examine the returned value and to achieve the selected solution to restart the standard execution.

```
invokeHandler: aHandler with: unwindContexts
    "defined on ExceptionalEvent"
    | exitValue resumeValue |
    "local variables"                                          (1)
    resumeValue :=                                             (2)
        self mark: #resume catch:                              (3)
            [exitValue :=                                      (4)
                self mark: #exit catch:                        (5)
                    [aHandler value: self].                    (6)
            self fastUnwind: unwindContexts.                   (7)
            thisContext sender: handlerContext                 (8)
            exitValue == SpecialMark                           (9)
                ifTrue: [handlerContext restart]               (10)
                ifFalse: [^exitValue]].                        (11)
        "control reaches that point if resumption has been ordered."
        ^resumeValue                                           (12)
```

Fig. 10. Handler Invocation.

Termination. Termination is simply implemented within the *exit* primitive (cf. Figure 11) by a destructive non local exit down to the *#exit* mark previously stored in the stack. The method *mark:exit:* is an equivalent of the *throw* classical lisp function. If the retry protocol is chosen, the value passed to *throw* is our registered special mark.

```
"Methods defined on FatalEvent class"
retry
    "exit and execute protected operation again."
    self mark: #exit throw: SpecialMark.
exit
    self exitWith: nil.!
exitWith: aValue
    self mark: #exit throw: aValue.
```

Fig. 11. Implementing Termination

Resumption. Resumption (cf. Figure 12) is simply implemented by a destructive non local exit towards the #resume mark previously stored in the stack at handler invocation time. The *resumeWith:with:* primitive checks that the handler has chosen a protocol for resumption effectively proposed by the signaler before entailing the non local exit.

```
"Methods defined on ProceedableEvent class"
resume
    self resumeWith: #resume with: nil!
resumeWith: aSymbol with: aValue
    (protocolForResumption indexOf: aSymbol) == 0
        ifTrue: [Error signal:
                    The proposed option for proceeding is not valid ...']
        ifFalse: [self mark: #resume
                        throw: (Association key: aSymbol value: aValue)]
```

Fig. 12. Implementing resumption.

There is no room to present other aspects of the system such as interactive propositions; however, their implementation does not raise any problem. The complete implementation can be downloaded from the author WEB page.

5 Conclusion

We have presented a specification and implementation of an open and expressive exception handling system for a dynamically typed object-oriented language. It provides a full object-oriented representation of exceptions and handlers. This now classical organization allows users to organize exceptions in an inheritance hierarchy reflecting the possible sharing of structures and behavior, and to trap any subset of exceptions with a single handler. The ability to define handling primitive on classes and to invoke them via message sending to the exception object makes it impossible to perform an inappropriate action for a given exception. The distribution of handling primitives on various abstract exception classes simplifies the signaling process by restricting the number of signaling primitives. Handlers can be associated to expressions and classes. We have shown the interest of associating dynamic scope handlers with classes. Class handlers are more than a powerful shorthand, they induce original ways to use inclusion polymorphism reusability.

This system architecture can also be considered as a framework for developing dedicated exception handling systems. It is for example very easy to use it to generate, by subclassing, other systems in which, for example, resumption or termination are forbidden. Its meta-object protocol for handing, made of a set of methods defined on exception classes, can be used as a basis to add new and dedicated EH control structures. We can see today a renewest interest for open, reflective and dynamically typed languages used for example to assemble components (cf. [5]) or to develop WEB applications that require powerful and flexible exception handling systems similar to the one presented here.

We finally have presented the key issues of the implementation of the dual model of exception handling in the context of a reflective, dynamically.typed object-oriented langage.

References

1. 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.
2. F.Christian: Exception Handling and Software Fault-Tolerance. IEEE Trans. on Computers, Vol. C-31, No. 6, pp. 531-540, June 1982.
3. C.Dony: An Exception Handling System for an Object-Oriented Language. Procs of.ECOOP'88, 1988; Lectures Notes in Comp. Sci. 322, pp. 146-161.
4. C.Dony: Exception handling & Object-Oriented Programming: Towards a Synthesis. Proceedings of the Joint conference ECOOP-OOPSLA'90, Ottawa, Oct. 1990. Special issue of Sigplan Notices, Vol. 25, No 10, pp. 322-330.
5. A.F. Garcia, C.M.F.Rubira; Architectural-based Reflective Approach to Incorporating Exception Handling into Dependable Software. In [23].
6. A. Goldberg, D. Robson: SMALLTALK 80, the language and its implementation. Addison Wesley 1983.
7. J.B.Goodenough: Exception Handling: Issues and a Proposed Notation. Communication of the ACM, Vol. 18, No. 12, pp. 683-696, December 1975.
8. J.Ichbiah & al: Preliminary ADA Reference Manual. Rationale for the Design of the ADA Programming Language. Sigplan Notices Vol. 14, No. 6, June 1979.
9. J.L.Knudsen: Better Exception Handling in Block Structured Systems. IEEE Software, pp 40-49, May 1987.
10. J.L.Knudsen: Exception Handling and Fault Tolerance in Beta. In [23].
11. A. Koenig, B. Stroustrup: Exception Handling for C++. Proceedings of Usenix'90, pp. 149--176, San Francisco, USA, April 1990.
12. R.Levin: Program structures for exceptional condition handling. Ph.D. dissertation, Dept. Comp. Sci., Carnegie-Mellon University Pittsburg, June 1977.
13. B.Liskov, A.Snyder: Exception Handling in CLU. IEEE Trans. on Software Engineering, Vol. SE-5, No. 6, pp. 546-558, Nov 1979.
14. B.Meyer: Disciplined exceptions. Interactive Software Engineering, TR-EI-22/EX, 1988.
15. J.G.Mitchell, W.Maybury, R.Sweet: MESA Language Manual. Xerox Research Center, Palo Alto, California, Mars 1979.
16. R. Miller, A. Tripathi: Issues with Exception Handling in Object-Oriented Systems. ECOOP '97 proceedings, Lecture Notes in Computer Science", Vol. 1241, pp. 85--103, Mehmet Aksit and Satoshi Matsuoka editors, Springer-Verlag 1997.
17. D. Moon, D. Weinreb: Signaling and Handling Conditions. LISP Machine Manual, MIT AI-Lab., Cambridge, Massachussets, 1983.
18. B.A.Nixon: A Taxis Compiler. Tech. Report 33, Comp. Sci. Dept., Univ. of Toronto, April 83.
19. K.Pitman: Error/Condition Handling. Contribution to WG16. Revision 18.Propositions pour ISO-LISP. AFNOR, ISO/IEC JTC1/SC 22/WG 16N15, April 1988.
20. K.Pitman: Condition Handling in the Lisp Language Family. In [23].
21. Objectworks for Smalltalk-80, version 2.5. Advanced User's Guide - Exception Handling. ParcPlace systems, 1989.
22. 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.
23. Advances in Exception Handling Techniques, Alexander Romanovsky, Christophe Dony, Jorgen Knudsen, Anand Tripathy Editors, Springer-Verlag, 2001.
24. S.Yemini, D.M.Berry: A Modular Verifiable Exception-Handling Mechanism. ACM Trans. on Progr. Languages and Systems, Vol. 7, No. 2, pp. 213-243, April 1985