Exceptions in Guide, an Object-Oriented Language for Distributed Applications

Serge Lacourte

Unité Mixte Bull-Imag, Z.I. de Mayencin, 38610 GIERES, FRANCE

email: Serge.Lacourte@gu.bull.fr

Abstract:

This paper describes the design of an exception handling mechanism for Guide, an object-oriented language based on a distributed system. We confront the usual exception techniques to the object formalism, and we propose conformance rules and an original association scheme. A specific tool to maintain the consistency of objects in the face of exceptions is provided. System and hardware exceptions are integrated to the mechanism, and parallelism is handled in an original manner. Some details of the implementation are given.

Key words: exceptions, consistency, object-oriented languages, concurrency

1 Introduction

The concept of exception handling has been refined in the last decade and is now an integral part of most high-level languages. Exceptions provide a means to separate a "normal" flow of control from an "exceptional" one, where the semantics of "normal" and "exceptional" may be predefined in the language or specified by the programmer. The advantages are twofold: the textual separation of the exception handling code from the normal one greatly improves the structure and readability of the program, while the semantical separation ensures that the normal flow is stopped when an exception occurs, and may only resume after the proper exceptional handling code is executed.

1.1 Issues

Most of the work on exception mechanisms has been done for traditional languages. Experience with exceptions in object-oriented languages is still limited. The integration of an exception mechanism in an object-oriented model should be coherent with the structuring principles of that model, especially as regards the delegation of responsibility and the internal consistency of the objects. This axiom has constantly directed our design, built around the method invocation on an object, the basic executing unit in object-oriented languages. An exception becomes an exceptional variation of the invocation, allowing the dialogue between caller and callee to be enriched.

When an operation cannot fulfil its requirements, it raises an exception. Control is then transferred to a calling entity which provides an exception handler. The handler executes some code and may choose between three main policies; *resumption* resumes the execution at the level of the signalling entity, *termination* (or *alternate* policy) resumes at the level of the handling entity, *propagation* signals a possibly different exception towards a higher calling entity. The handler may also retry the failed call, as a variation of *termination*. Handlers are associated to invocation instructions in the caller. These are parts of a standard mechanism that needs to be refined in an object-oriented language. Is resumption to be provided? How does exception raising relate to interface checking? What is the scope of handlers? What is the precise effect of termination? What is the impact of inheritance on all these points? How do we declare default handlers?

An object-oriented language must also address the issue of consistency of the objects, particularly if they are persistent. Are the standard mechanisms sufficient, or do we need a specific one? There is also the specific issue raised by concurrency. How is the normal termination of two parallel invocations related to the normal termination of the same invocations in sequence? The underlying system can also fail. How does it report exceptions? How are handled system and hardware exceptions?

This paper is an attempt to contribute to these issues. It is based on our experience of integrating an exception mechanism in Guide [1], a strongly typed object-oriented language for distributed applications, jointly developed with a supporting operating system environment [2].

1.2 Previous work

Most current object-oriented languages offer an exception handling mechanism similar to those of modular languages. Ada, CLU and Modula2+ have all three chosen a termination model. Their main characteristic is that they associate a handler with a block of instructions, syntactically as well as semantically. A handler with a termination policy resumes the execution at the instruction following the block. In Guide, execution resumes just after the call. Another point is the propagation "as is" of an exception through operation frontiers, except in CLU, where the *FAILURE* exception is propagated. In a similar way, the frontier of method invocation is not bypassed in Guide.

1.2.1 A termination model

The mechanisms offered in Modula-3 [8], Trellis-Owl [10], ANSA [3], Argus [4], NIL [9], are more or less related to the model shortly described above. Their main improvement consists of a strict control of the exceptions a method may raise, or propagate. Usually, when an exception is not handled by the caller, a special exception is propagated, like the *FAILURE* exception of CLU. Only Modula-3 does not conform to this rule. Guide propagates the *UNCAUGHT_EXCEPTION* system exception.

But all the above languages have kept the association of a handler with a block of instructions, with the related termination behaviour explained above. The drawback of this model is that the control flow depends on the place where the handler is declared. In particular if the handler offers an alternate computation to a precise method call, then it has to be associated to a block which contains this only call. In fact the advantage of this model is to avoid the handler having to execute a local control transfer instruction to exit the block. However it prevents a clear separation of the exception handling code from the main algorithm. In Guide the handler just applies to the raising invocation, without relation to its declaration.

Few of these languages answer the issue of the consistency of objects. Argus integrates a mechanism whose purpose is to restore the state of a guardian and to restart it after a failure, but nothing is done for internal objects. Modula-3 keeps the **try** ... **finally** tool provided in Modula2+, that ensures the **finally** part being executed whatever happens during the execution of the **try** body. This is more a finalization tool than a restoration one, because the code is executed even if the body ends normally.

1.2.2 Eiffel

Exceptions in Eiffel [6] are handled in a very specific way. The **rescue** clause can be simulated by a handler associated to the method with the above semantics. The only authorized policies are either to retry the execution of the whole body, or to propagate the exception. This tool is rather primitive, but it turns out to be appropriate to ensure the consistency of objects. In fact it has been designed for that purpose: it ensures the correctness of the postconditions of the method, which include the invariant of the object. However nothing special is offered to deal with an exception raised during the execution of this rescue code, as we have tried to do in Guide with the *RESTORATION_FAILED* exception.

1.2.3 Exceptions as objects

Exceptions have been considered as objects in some Lisp-based languages. To raise an exception is to create an instance of the related class, then to call it with a *raise* method. The behaviour of any of the *termination*, *retry*, *resumption* policies can be defined as a method of a root *Exception* class from which all exception classes inherit. A complete model is defined in [7] for the Lore language.

This approach is interesting because it tries to integrate exceptions into the standard invocation mechanism; however by doing so it complicates its semantics. A *raise* method call eventually resolve into a **return** (*termination*), a normal call (*resumption*), or something more complicated (*retry*). How then can such a method be formally defined? Keeping the exception as a possible response to the method invocation, as we have done in Guide, leads to a semantically simple model.

The remainder of this paper is organized as follows. Section 2 describes the main features of the mechanism. We first present in section 2.1 the Guide project, and the relevant concepts of the language. We then describe the core of the mechanism in three sections, touching in turn on exception raising, handler association, and handling policies. Section 2.5 tries to bring new arguments to the debate about resumption, and it is followed by a concise summary of our proposal in section 2.6. Section 3 touches on more specific issues. Section 3.1 shows how we integrate system and hardware exceptions to the main scheme. Section 3.2 describes a specific tool to maintain the consistency of objects. Relationship with inheritance and conformance is developed in section 3.3. Section 3.4 reports on the integration of exceptions in the parallel construct of Guide. In the final part of the paper, section 4, we discuss three future developments of the mechanism.

2 The Guide proposal: general features

2.1 The Guide language and system

The Guide programming language has been designed and implemented for the programming of distributed applications. Its run-time support is provided by the Guide object-oriented distributed operating system, designed and implemented at the University of Grenoble as a joint project of IMAG and Bull Research Center [2].

The object model implemented in Guide is characterized by the following main features:

- Objects are typed. A type describes a behavior shared by all objects of that type, in terms of the signatures of the operations applicable to the objects. Objects are accessed through views. A view is essentially a typed pointer; the effective type of the object must conform to that of its view. The conformance rule is statically checked. Types are declared apart from classes, which describe specific implementations of types.
- Subtyping defines a hierarchy of types. The subtyping hierarchy is paralleled by a subclassing hierarchy, with single inheritance. Subtyping and inheritance ensure conformance.
- Objects are persistent. They are named by system references (system-wide uniquely generated internal names). Complex structures may be constructed by embedding references to objects within other objects.
- Concurrency is provided by a **co_begin** ... **co_end** construct allowing for the creation of parallely-executed sub-activities.

The execution model involves distibuted jobs and activities. Execution structures extend to a remote site when needed.

The characteristics of our proposal are described in the following sections. We first present an example which will be extensively used throughout our presentation. A basic *Editor* allows the user to browse through a *Document, Page* by *Page*. It caches the *Page*s it has read into *ShadowPage* s until *Commit*. It gets the needed *ShadowPage* s from an *ObjectManager* of *ShadowPage* s, to lighten the load of the garbage collector. A *Garbage* method allows it to free its clean *ShadowPage* s. Two other methods are described to print the current *ShadowPage* and to create a new one. A *Page* is read *Char* by *Char*, and a *ShadowPage* is a *Page* with a "dirty" flag. An *ObjectManager* provides and gets objects back with the methods *Get* and *Free*. The keyword **type** introduces the definition of an interface. The keyword **class** introduces an implementation. Only state variables are given below in the implementation of *Editor*, the methods are described later in the document.

```
type Editor is
    method Commit;
    method PrintPage;
    method CreateNewPage;
    method Garbage;
end Editor.

type Document is
    pages: ref List of ref Page;
end Document.
```

```
type Page is
    method GetChar: Char;
        signals end_of_line,
            end_of_page,
            error;
end Page.

type ShadowPage
subtype of Page is
    isDirty: Boolean;
end ShadowPage.
```

```
class Editor implements Editor is
    // state variables
    shadowPages: ref List of ref ShadowPage;
    shadowPageManager: ref ObjectManager of ref ShadowPage;
    currentPage: ref ShadowPage;
end Editor.
```

2.2 Exceptions in the object-oriented design

Object-oriented languages encapsulate data and code in an object (an instance of a class). The basic computation unit is the method invocation on an object, in which a message is sent to and interpreted by the called object; this allows polymorphism. Exceptions naturally fit in this scheme, as a possible response to this message. Thus the exception handling mechanism should also be defined at object level.

Due to the choice of a termination model (see section 2.5), the calling object does not answer to the raising of an exception. Raising an exception exits the method in the same way as a return statement.

This dialog between caller and callee is controlled by the interface of the called object. As this interface is explicitly defined in Guide, it must include exception declarations. This affects the conformance rules checked by the compiler, which are detailed in section 3.3.

Another issue with exception raising is the definition of the parameters that may be associated with it. In Guide, a method cannot pass back new parameters when raising an exception, but it can use its standard "out" parameters instead. This solution has the advantages of allowing parameter passing, while keeping the simplicity of exception raising.

The implementation uses a string variable for each activity. Raising an exception sets this variable and returns the control to the calling entity, which is in charge of testing the variable. Since an activity may be distributed (see section 2.1), the system ensures that this variable remains consistent across multiple nodes. The compiler generates code to test it after each system/object call.

2.3 Handlers in the object-oriented design

We have just seen that exception raising is associated to a method call. This holds as well for exception handling. What it means is that the alternate or retry policies mentioned in section 1.1 only apply to the method call and not to a possible block in which the call is done. A characteristic of our proposal is that a handler may only be semantically associated to a method call, and not to a block of instructions. When the call *page.GetChar* raises an *end_of_line* exception, the handler supplies a replacement value (**replace** is discussed later) and *PrintPage* continues with a call to *output.WriteChar* to print it.

```
method PrintPage;    // of class Editor
begin
    while TRUE do
        output.WriteChar(currentPage.GetChar);
    end;
except
```

```
        end_of_line from Page: replace '\n';
            // gives the printable character for a newline
        end_of_page from Page: return;
            // exits the method PrintPage
    end PrintPage;
```

A handler can be associated to one instruction. Its declaration can be syntactically factorized to a block of instructions, to the whole method as in *PrintPage*, or even to a class of objects, but the handler is semantically associated to each operation (method call) of the block, or of the method, or of each method of the class. The programmer can then declare semantically precise handlers at the method level, and that allows him to clearly separate exception handling from the main flow of control.

In order to help the user to precisely define the scope of his handlers, we allow a handler to be associated not only to an exception name, but also to a type and a method name. The handler may only handle exceptions of the given name, raised by a method call of the given name on a reference to a type which conforms to the given one. The type and method name can be omitted when unnecessary. Besides, the keyword **ALL** may be used in each field, in order to factorize the declaration. Note that the handler in *PrintPage* is associated to the type *Page*, but it can handle the *end_of_line* exception raised by *currentPage* which is a *ShadowPage*, because *ShadowPage* is a subtype of *Page* and conforms to *Page* (see section 2.1 about the Guide model).

The implementation uses a stack variable, local to each method call, which keeps the in scope handlers with the mask of the exceptions (and optionally the related method and type) that they may handle. This stack is updated once for each handler declaration, and its order is significant, high priority first. When an exception is raised, the first handler with a matching mask is executed.

2.4 Exception handling policies

Separating the exceptional code from the normal one seems a clean technique, but it raises the following problem. In Guide a method may return a value, which may be used as parameter in another method call. This is the case with *GetChar*. If it raises an exception, the handler may want to do the work by another means, so it must be able to provide an alternate character to be printed. This is the aim of the **replace** keyword used in the handler. The conformity relationship between the replacement value and the one expected by the operation is checked at compile time.

The retry policy is provided through the **retry** keyword that only a handler may use. This is a generic way to ask for a new execution of the call that raised the exception; so it allows general fault recovery policies to be programmed. The new execution should be done in the same environment as the first one, with optionally new handlers set by the programmer. However this induces a risk of recursively calling the same handler. To prevent this, the system ensures that a handler cannot be called again while it has not terminated. The following handler associated to the class *Editor* tries to recover from a failure of the *ObjectManager* (called in method *CreateNewPage*).

```
class Editor implements Editor is
    ...
    method Garbage;
        page: ref shadowPage;
    begin
        page := shadowPages.First;
        while page do
            if page.isDirty then
                shadowPageManager.Free(page);
                page := shadowPages.Delete;
            else
                page := shadowPages.Next;
        end; end;
    end;
except
    noObjectLeft from ShadowPageManager.Get: begin
        self.Garbage;
        retry;
        end;
end Editor.
```

We also provide a default handler to handle otherwise unhandled exceptions. The encapsulation of code guarantees that the calling object only knows the signature of the called method. So, when the method *CreateNewPage* of *Editor* calls the method *Get* of the *ObjectManager*, it does not know how it is implemented. The *ObjectManager* can use a memory allocator which may raise a *NO_MORE_MEMORY* exception, but the *Editor* cannot understand this exception. It must be handled by the *ObjectManager*, or a default policy has to be applied, but it cannot be propagated unchanged. The Guide system provides a default handler which propagates the *UNCAUGHT_EXCEPTION* system exception (cf section 3.1). This insures that an exception will either be handled or will eventually terminate the task.

```
method CreateNewPage;    // of class Editor
begin
    currentPage  :=  shadowPageManager.Get;
end;
```

The implementation of **retry** and **replace** is allowed by the generic access respectively to the last method invocation, and to the optional return parameter of this invocation.

2.5 Resumption versus termination

Let us now explain why we have chosen a termination model in Guide. Whether resumption should be provided or not is an old debate. It has been supported in [11] and rejected in [5]. We now try to bring new arguments to this discussion, in relation to the object-oriented formalism.

We identify two main uses of resumption, depending on the provider of the handler. The handler may be provided by the user, through a debugger or a specialized shell, or it can be included by the programmer in the original code. The question is: what is the meaning of resumption after handler execution, and does it preserve the object encapsulation principle?

In the first case (debugger), the stack of the nested calls is somehow saved, and control is given to the user. He can browse the stack, change a value at some level, and then resume at a possibly different level, perhaps the deepest. This means that the part of the saved stack between the level of the changed value and the resumption level is considered to be valid. This supposes a complete analysis of the concerned code, so it violates the frontier between the calls. However it could be left to the user's responsibility, in such specialized tools as the debugger. But to give basic tools allowing other applications to do the same is another issue.

The call of a resumption handler must then be viewed from the other side, as a procedure call. A new context is pushed on the stack, the procedure executed being given by the caller. The question becomes: does passing a method as a parameter violate the encapsulation principle of objects? The aim of encapsulation is to allow to prove (formally or informally) the correctness of the implementation of an object interface, using only the interfaces called by this object. The correctness of an object may not depend on the implementation of the objects that use it. Passing methods as parameters in an object-oriented language should then be formalized, as is formalized the passing of objects as parameters. This is indeed done in Lisp-based languages, where methods are or tend to be first class objects [7], but it is generally not the case in strongly typed object-oriented languages like Trellis-Owl, Eiffel or Guide.

We can conclude from this discussion that resumption is related to the passing of methods as parameters. If the latter is correctly formalized in the language, then the former may be provided. In Guide this is not the case, so we adhere to a termination model, with the retry variation.

2.6 Summary

Exceptions are associated to object methods. Exceptions potentially raised by a method appear in its interface. Conformance rules are modified to take exceptions into account.

Handlers are associated to method invocations. The normal continuation after the execution of a handler is the point just after the raising method invocation. It has nothing to do with the syntactic declaration of the handler, which may be factorized at the method or class level. A type and a method modifiers help to refine the scope of the handler.

We have chosen a termination model. A handler may provide a replacement value when the raising method invocation is functional. The **retry** keyword allows a handler to reexecute the raising call in a generic manner. A default handler guarantees that an exception, once detected, cannot be unintentionnally discarded.

3 The Guide proposal: specific issues

3.1 System and hardware exceptions

Up to now we have only considered exceptions raised by method calls. However programs also ask services from the system, which may not be able to provide them. The problem is how to report these exceptions to the programmer.

In Guide we have decided to consider the system as a special class, and each instruction as a method call to the system class. In this way we handle system exceptions in the same way as other exceptions. They are predefined and may be handled with the keyword **SYSTEM** as a type, which may be omitted.

Examples of hardware exceptions are segmentation fault or arithmetic overflow. Such exceptions also appear as exceptions raised by the system. However this raises an implementation issue because such exceptions can effectively occur at nearly each instruction, so it is no longer possible to detect them by a test after each call, as it is done for system exceptions.

Interrupts are asynchronous events. They are not provided in Guide, except when an activity aborts. The system converts this event into a *QUIT* exception, which is detected later on, at the following system/method call.

3.2 The need for consistency

Our exception handling mechanism allows the programmer to implement alternate or retry policies when a call fails. But it does not insure that the called object is in a consistent state after it has raised an exception. Consistency depends on the validity of data relationships (invariants) that may be maintained by the programmer. An invariant is assumed to be true, except for the periods when an object operation is executed. The invariant must be restored at operation exiting. This is especially needed in a parallel environment where the object can be used by another task, or in a persistent object system in which objects may be reused.

In Guide, the **restore** keyword allows to define a restoration block which is executed whenever the method exits abnormally (raises an exception). It is not executed if the method returns normally, as would do a finalization mechanism, because we want to address the problem of consistency, and the object is assumed to be consistent in this case. Guide does not yet offer such a finalization tool [8], but our restoration mechanism partly solves the problem of the multiplication of exit points induced by an exception mechanism. Exit points can be reduced to two, a normal one and a unique exceptional one. As an example, a restoration block associated to the class *Editor* allows us to save the dirty *Pages* of the *Editor* whenever it exits abnormally.

```
class Editor implements Editor is
    ...
restore
    page: ref ShadowPage;
```

```
    shadowDocument: ref Document;
begin
    page := shadowPages.First;
    while page do
        if page.isDirty then
            shadowPageManager.Free(page);
        else
            shadowDocument.pages.Append(page);
        end;
        page := shadowPages.Delete;
    end;
    if shadowDocument.pages.nbItem > 0 then
        output.WriteString("abnormal  exit\n");
        output.WriteString("modifications saved in " + <name>);
        ... (shadowDocument, <name>);
    end;
end;
end Editor.
```

The programmer can optionally associate a restoration block with a class and with a method. If the method raises an exception, then the block associated to it is executed, after which the block associated to the effective class of the object is executed. Note that this class may be different from the definition class of the executed method, if this method is inherited.

A last characteristic of the Guide proposal is related to exception handling in the restoration code. Class handlers and method handlers are active during the execution of the restoration block associated to the method. Only the former are active for the restoration code of the class. The block can also declare its own additional handlers. The main point is that if the restoration code does not handle an exception, or if it raises an exception itself, then the system propagates the *RECOVERY_FAILED* system exception. This ensures that the caller will not try some recovery operations on a called object which has not restored a consistent state. This is a first step to address an issue which is further discussed in section 4.2.

The implementation is different depending on the association. The compiler ensures that a method has only one exit point, tags it with a flag, and puts the method restoration code just after the flag. The class restoration code is directly called by the system.

3.3 Inheritance and conformance

The conformance rule between types has to be modified to take exceptions into account. It has been stated that an exception cannot be raised by an operation if it was not declared in its signature.

Thus, in the method *PrintPage*, the call *currentPage.GetChar* cannot raise any other exception than *end_of_line*, *end_of_page* or *error* which are declared in and inherited from the type *Page*. It implies that the method *GetChar* of any type that conforms to *ShadowPage* may only raise a subset of the three exceptions. This is the same rule as for output parameters. In Guide a subtype conforms to its super-type, so the definition of type *ShadowSpecialPage* below is not correct.

```
type ShadowLongLine
subtype of ShadowPage is
    method GetChar: Char;
        signals end_of_line,
            error;
end ShadowLongLine.
// this type is correct
```

```
type ShadowSpecialPage
subtype of ShadowPage is
    method GetChar: Char;
        signals end_of_line,
            end_of_page,
            unprintable,
            error;
end ShadowSpecialPage.
// this type is incorrect
```

Inheritance is also a tool to factorize code, and it impacts the handling code and the restoration code. Handlers that are associated with a method or with some instructions of a method are not inherited apart from the method. This means that either the subclass inherits the method and then it gets the related handlers, or it redefines the method and then it has to redefine handlers. This is natural because these handlers are supposed to heavily depend on the method code. On the other hand, handlers that are associated with a class are automatically inherited in the subclasses. This would be the case with the "garbage handler" for a subclass of *Editor*. This also allows to define default handlers for an application by the means of a common super-class.

We conclude this section with the inheritance of restoration code. As for handlers, only the block associated to the class is inherited. However a restoration block defined in a subclass overrides the one defined in the class.

3.4 Managing concurrency

A sequential algorithm assumes that an operation has succeeded before executing the next one. This is no longer the case with a parallel algorithm where one may want to try simultaneously two operations and be satisfied with the first that returns a useful result.

In Guide, the **co_begin** statement enables the programmer to make concurrent calls. One variable per sub-activity is available, yielding *TRUE* if and only if the associated branch has terminated normally, i.e. without raising an exception. The programmer can express the termination condition with a **and** and **or** combination of these variables. If this condition yields *TRUE* after the normal termination of a branch, then the remaining branches are stopped and the parallel statement exits normally. On the other hand, if the failure of a branch prevents the normal termination condition from ever being verified, then the remaining branches are stopped and the system raises the *JOIN_FAILED* system exception.

This mechanism must be used with care: a *TRUE* branch variable means "I have successfully terminated", and not only "I have terminated". In the following example of a parallel producer and consumer, the condition **co_end***(producer* **and** *consumer)* means that the two tasks must correctly terminate. If one of them terminates abnormally, then *JOIN_FAILED* is raised and *<handler>* is executed. The condition that the consumer has to terminate when the producer has terminated must be expressed in another way, because a **co_end***(producer)* termination condition would induce an abnormal and rather rough termination of the consumer, and this would not detect a previous abnormal termination of the consumer.

```
method Main;
begin
    co_begin
        producer: ...    // producer code
        consumer: ...    // consumer code
    co_end(producer and consumer);
except
    JOIN_FAILED:  <handler>;
end Main;
```

Note that the parallel block may either satisfy its termination condition, or be sure that it will never satisfy it, while some of its branches still run. In this case the system stops them softly by raising a *QUIT* exception (see section 3.1). It allows them to perform some restoration before exiting.

The mother activity cannot access the possible exception raised by a branch in the termination condition. However it can interpret this exception by providing a new handler specific to the branch.

To implement the failure detection, the system maintains another array of boolean variables, one per branch, that yields *TRUE* if and only if the corresponding branch has failed. When a branch fails, the system applies the termination function to the negation of these variables, and a *FALSE* result indicates that the parallel block has failed. The proof is simple. The negation of the variables identifies the activities that has terminated or may still terminate normally. This is clearly a superset of the positive variables in the final state. As we use only AND and OR modifiers, if the termination function yields true in the final state, it yields true with this superset.

4 Future work

We have discussed so far the current design of the exception handling mechanism in Guide, and the reasons behind the choices. Everything that is described in the two previous sections is implemented and works, except for the aspects related to hardware exceptions. We now present the extensions we plan to implement.

4.1 Exception hierarchy

Exceptions currently are simple strings associated to a type. This solution has the major drawback that an overloaded method in a subclass cannot raise another exception than those declared by the method in the superclass (cf section 3.3). In fact what we want in a subclass is to specialize each exception of the superclass. Exceptions must then be organized in a hierarchy respecting the subtype hierarchy of the corresponding types. It now becomes possible to declare *ShadowSpecialPage* as a subtype of *ShadowPage* (cf example in section 3.3), given the needed syntax which could be:

```
method GetChar: Char;
    signals end_of_line, end_of_page,
        unprintable isa error, error;
```

This gives a new task to the system because when *PrintPage* is compiled, the compiler does not know that *currentPage* may be a *ShadowSpecialPage* . So the system must be able to turn an possible *unprintable* exception raised by the call *currentPage.GetChar* into an *error* exception that *Editor* knows.

Another advantage of this solution is that it would be easier for the compiler to check the validity of an exception raised in a handler defined at the class level, because this exception could be defined directly at the type level. Each exception would be a specialization of a primary *error* exception defined in type *Top*, which is the supertype of every Guide types. In the current design, one would have to check that the exception is declared in each method which can potentially cause the handler to be activated; this check is not done because it is complicated.

4.2 Restoration

The mechanism we propose still has some controversial points, such as the different handling of the restoration code defined in the method and in the class. These aspects will be evaluated by experience. The current design is also too rigid, because the restoration code is executed when the method raises any exception, even when the programmer is sure that the object is in a consistent state.

Two solutions can be developed. In relation to the previous section it is possible to declare two exceptions in the type *Top*: *error* which induces the execution of the restoration code when it is raised, and *exception* which does not. A more handy way is to consider the restoration code as a second level handler, which can be associated to the needed exceptions, and which is executed in addition to the regular handler. Note that the second solution cannot be simulated with the current default handlers associated to a class, because in this case the default handler is overridden by a more precise one. The aim here is to execute additional code.

A more complex issue has been introduced in section 3.2, about failure occurrence during the execution of the restoration code. Currently the sytem raises the *RECOVERY_FAILED* exception, preventing the caller from using the object in an inconsistent state. However objects in Guide may be shared and some action should be taken at least to warn the current and the future users of the object. This leads to the definition of a new variable in the object, its status, which controls the execution of the method. Standard methods later called with a bad object status raise an exception, while repair methods bypass this mechanism. Methods executing concurrently with the recovery actions are warned through another exception, when the recovery fails. All this needs further investigations.

4.3 Hardware exceptions

The aim is to handle hardware exceptions like the other system exceptions (cf section 3.1). In fact we want to be able either to continue the execution after the raising point if the handler exits normally, or to propagate the exception (i.e. to return with the exception parameter set). We can then attempt to provide the retry, but it is not imperative.

Using C on Unix makes the propagation difficult. A hardware exception is turned into a signal and the programmer can execute a standard procedure call, the signal handler. When the call ends the execution automatically continues. In the next phase of the project, we will use a low-level distributed kernel, Mach or Chorus, which can help. In Mach and Chorus, a hardware exception stops the thread and is turned into a message sent to the exception port of the thread. Another thread can then read this message, take some actions such as touching the stack of the stopped thread, and then restart it. To implement Unix signals is to push a procedure call on the stack. We propose instead to change the program counter of the stopped thread, so that it executes the handler when it resumes. The return and the propagation become easier to implement, and the continuation needs to get the former program counter to be able to resume correctly.

5 Conclusion

This paper has described the exception handling mechanism we have designed and implemented for Guide, a strongly typed object-oriented language supported by a distributed system. The choices have always been directed by the encapsulation principle of the object formalism, and by a concern for orthogonality to the other concepts of the language. This resulted in the following characteristics:

conformance : exceptions are associated to object methods. Exceptions potentially raised by a method appear in its interface. Conformance rules are modified to take exceptions into account.

association: handlers are associated to method invocations. The normal continuation after the execution of a handler is the invocation that immediately follows the invocation of the method in which the exception was raised. It has nothing to do with the syntactic declaration of the handler, which may be factorized at the method or class level. This contributes to satisfy the main goal of an exception handling mechanism: to separate exceptional cases from the main algorithm.

inheritance: a handler defined in a class is automatically inherited in the subclasses. This is a convenient means to declare default handlers.

restoration: restoration code may be provided at the method and class level. It is executed whenever the method exits abnormally. This is a useful tool to ensure the consistency of objects, and it also helps solving the finalization issue.

parallelism: exceptions are integrated in a natural and powerful way in concurrent computations, allowing complex termination policies to be implemented.

Everything that is described in section 2 and 3 is implemented. The rather primitive current implementation induces an additional cost of one assignment and one test per method call, inducing a small executing overhead when no exception is raised, and even no overhead considering the tests needed after the invocations when no exception handling mechanism is available, but increasing the code size by 25 per cent. It also adds a few assignments per handler declaration, but uses more execution time when an exception is raised, in order to find the right handler to execute.

Acknowledgments

I have been constantly supported by the whole Guide team while specifying and implementing this mechanism. I wish to thank Sacha Krakowiak, Véronique Normand and Xavier Rousset for their detailed criticisms of previous drafts of this paper. Project Guide is partly supported by the Commission of European Communities under the Comandos ESPRIT Project (no 2071).

Bibliography

[1] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill and C. Roisin. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming*, 3(3), pp. 11-22, September-October 1990.

[2] R. Balter and al. Architecture and Implementation of Guide, an Object-Oriented Distributed System. *to appear in Computing Systems*, 1991.

[3] *ANSA Reference Manual*. Architecture Projects Management Limited, 24 Hills Road, Cambridge CB2 1JP, United Kingdom, March 1989.

[4] B. Liskov, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler and W. Weihl. *Preliminary Argus Reference Manual.* October 1983.

[5] B. Liskov and A. Snyder. Exception Handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6), pp. 546-558, November 1979.

[6] B. Meyer. *Object-Oriented Software Construction*. Series in Computer Science Prentice Hall International, 1988.

[7] C. Dony. An Object-Oriented Exception Handling System for an Object-Oriented Language. *Proc. ECOOP '88*, pp. 146-161, August 1988.

[8] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow and G. Nelson. *Modula-3 Report (revised).* DEC SRC, October 1989.

[9] W.F. Burger, N. Halim, J.A. Pershing, R. Strom and S. Yemini. *Draft NIL Reference Manual.* (42993), IBM, TJ Watson RC, P.O. Box 218, Yorktown Heights, NY 10598, December 1982.

[10] C. Schaffert, T. Cooper and C. Wilpolt. *Trellis Object-Based Environment, Language Reference Manual.* (DEC-TR-372), DEC, Eastern Research Lab, Hudson, Masachusetts, November 1985.

[11] S. Yemini and D.M. Berry. A Modular Verifiable Exception-Handling Mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2), pp. 214-243, April 1985.

Exceptions in Guide, an Object-Oriented Language for Distributed Applications