Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 189–197.

A general method of low-level exception reporting and handling using System Predicate Classes

Ádám Balogh[a], Zoltán Csörnyei[b]

[a]Department of Algorithms and their Applications
Eötvös Loránd University, Budapest, Hungary
e-mail: bas@elte.hu

[b]Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
e-mail: csz@inf.elte.hu

Abstract

System software rarely benefits from the exception reporting and handling mechanisms of high-level programming languages, because they are always language-specific and indicate extra overhead. Instead, they provide some simple simple solution for reporting and handling of exceptional situations such as designated return values, global error flags, long distance jumps, event handlers (callbacks) or a combination of these mechanisms. However, none of these tools enable the correct and safe separation of code for normal and exceptional control flow themselves. Our presentation is about new language elements enabling separation of error-handling code from regular one, while relying on the above mentioned low-level solutions and being compatible with them.

1. Motivation

Several high-level programming languages support separation of error handling code from regular one by introducing an exception reporting and handling mechanism. Among many others, typical examples are mandatory exception handling of *Ada* and optional one of *C++* and *Java*. In these solutions, exception handling and reporting belong together, exceptions raised by the exception reporting tool of the language can only be handled by its exception handling tool and vice versa. Since the implementation of these mechanisms is language specific, it is impossible to use them in mixed-language software systems.

As general-purpose operating system operating systems must support applications written in any languages, they cannot make use of a language-specific exception reporting mechanism to report errors to user-level software. Instead, they have to chose simpler tools to do this, which disables the use of the high-level mechanisms on the application side as well. These tools are typically one or more of the following ones:

Special Return Values: The return value of a system call denotes whether the call could be performed properly. The specification of the system call describes which values mean successful completion and which one failure. The actual reason for the failure can either be determined directly from the return value, or the solution is combined with the next error-reporting mechanism:

Global Error Flags: A global variable, accessible for writing by the system call and for reading by the system call denotes the kind of the failure happened during the last call. Typically, *POSIX* uses this two approaches together: whenever a function returns negative value (or NULL in case of pointer returning functions), the actual kind of the error can be read from the global variable `errno`.

Callback Routines or Signals: In case of failure, the operating system calls a function belonging to the application, that is intended to handle the actual failure. It can either happen directly (callback) or through a standard mechanism of the operating system, the so called signals: the application gets interrupted, and the signal-handler function is executed. This function is either provided by the system, but runs on behalf of the application, or it is overridden by the application itself. There are usually different signals, reporting different events.

Beside reporting failures to the application, an operating system itself meets exceptional situations as well. If a high-level language supporting exception handling is chosen for development of the operating system, its mechanisms could be used for this purpose. However, these mechanisms cause some extra overhead which is hardly ever tolerated in an operating system. This implies that low-level construct are used for internal error-handling as well, such as the three described above, extended by a few other possibilities:

Goto: In failure situation, the code being executed get aborted, and a jump to the error handling code is performed. Sometimes it is the most efficient solutions, but the use of `goto` instruction corrupts the structure of the code and thus may result in hard discoverable programming errors.

Long Distance Jumps: The `setjmp()` and `longjmp()` functions of the programming language *C* enable transfer of control to a higher stack frame. The main drawback of this solution is the difficulty of freeing allocated resources properly before the handlers for such resources leave the variable scope.

Handling of exceptions raised by these low-level ways must be handled differently. Handling of special return values and global error flags require branches, which does not allow separation of error-handling code from regular one. Callbacks and signals must be registered beforehand, at the beginning of the block. Special care must be taken when overriding these functions by new ones, and the old one must be reset at the end of the block. Goto is static, the handling code is fixed. Long distance jumps must be handled by an unusual branch instruction, which – beside mixing of error-handling code with regular one – corrupts the structure of the code more than the check of return values. Macros may be used to make handling easier and code more readable, but they are language specific as well. Further problem with macros is that they are usually implemented by a preprocessor, as in *C*, not composing real part of the language, which can lead to nasty programming errors.

In this paper we describe language constructs that allow construction of a unique interface for the above mentioned exception reporting and handling mechanisms. By separation of reporting and handling, our tools are able to work asymmetrically as well: we can use the reporting part to report an exception to an application that handles it without our interface, as well as we can use the handling part to handle exceptions reported by a system call without our interface. Only the mechanism behind the interface must match. After describing the new language elements for low-level exception handling, we present how *System Predicate Classes* – a tool for object-oriented programming at system level – can make exception-handling easy.

The remaining of this paper is organised as follows. In Section 2 we give a short overview of *System Predicate Classes.* Section 3 presents new language elements for exception reporting and handling. In Section 4 we show how *System Predicate Classes* can be used for exception-handling. Section 5 discusses related work, while Section 6 concludes.

2. System Predicate Classes – Overview

Predicate Classes are a special approach of classes in *Object-Oriented Programming*, where instead of external identifiers or pointers to virtual method tables Boolean expressions on the values of an object determine its dynamic type. This allows an object's representation to contain only the fields explicitly defined by the programmer. Furthermore, *Predicate Classes* make possible for an object to change its dynamic type if its state is modified implying the change of the truth value of its classifying predicates.

System Predicate Classes are *Predicate Classes* with fixed representations, able to store system data structures, e.g. registers, hardware-defined data structures etc. They may contain special fields, denoted as *reserved*, which may be overlapped in their descendant classes. This enables polymorphism-by-value, which means that a variable defined for the type of the superclass may store an object of its superclass as well. Most high-level *Object-Oriented* languages allow this only for references. Dispatcher methods evaluating the predicates and jumping to the appropriate method body are generated during link time to enable modular software development. These jumps eliminate long call-chains, giving the technology a very good performance, since its time costs contain only the execution times of the simple jump instructions.

3. Exception Reporting and Handling

As mentioned earlier, we separate exception reporting from exception handling. Since our tools are interfaces to existing low-level mechanisms, their use is optional on both sides. This means, that exceptions reported using the new language elements may be handled manually, while manually reported exceptions can be handled by our tools. The only criteria is that the reporting and handling mechanisms must match each other.

To present the new elements, we use a *Pascal*-like syntax, used by our new language under development. More details on this syntax are to be found in [4].

3.1. Exception Reporting

Reporting of an exception is done by a sequence of instructions. Together with introducing exception names, we allow assignment of handling code to these names. Beside names, exceptions may take parameters as well. When raising an exception, the reporting code is copied to the place of the call after parameter substitution.

For example, for *POSIX* functions returning an integer, we introduce the exception name `Int_Error`, with an integer parameter, the error code, which must be assigned to the global variable `errno` before returning `-1` to the caller. This exception is reported by the following macro:

```
report Int_Error ( Code: Integer ) by
   begin
      errno := Code;
      return -1;
   end;
```

Other kinds of exception reporting can be defined similarly, only the actual code raising the exception differs. If the above definition exists somewhere in the definition part of a block, an exception for e.g. an interrupted instruction is raised by the statement `raise Int_Error ( EINTR );` inside the scope of the definition.

An exception is usually reported by a subroutine and must be handled by its caller. However, the caller must be prepared for the exceptions the subroutine may raise. Since the body of the callee is not always available when compiling the caller, it cannot be searched for exception reporting instructions. Instead, all exceptions the subroutine may raise must appear at its public declaration: `function Pause raises Int_Error`.

3.2. Exception Handling

Handling of an exception may be very different depending of the reporting mechanism used. Special return values and global error flags require a check after the call, callbacks or signal handlers must be registered before the call, and maybe reset after them, while goto instructions jump to a fixed handler. Long distance jumps compose a special case, since there the call for the function is syntactically conditional, while semantically the handling happens after it. To cover all these cases we introduce a general exception handling structure. For every exception, we allow definition of code that must be performed before and after the call, together with a condition for the call to be performed. In the exception handling code, a special instruction may appear that retries the call, and a special constant refers to the return value of the called function.

For example, an exception-handling code for the above presented `Int_Error` may be the following:

```
handle POSIX.Int_Error
   afterwards
      if return = -1 then
         if POSIX.errno = EINTR then
            retry;
         else
            POSIX.perror ( POSIX.errno );
         end if;
      end if;
```

The keyword `return` in this code is not used as an instruction, but as a value, which in an exception handling code refers to the return value of the called function. For an error which is reported by a long distance jump the handling code is more complicated:

```
handle An_Error
   beforehand
      V := Set_Jump;
   condition V = 0;
   afterwards
      if V <> 0 then ...
```

Here function `Set_Jump` corresponds to the function `setjmp()` of the programming language C. In this example, all three parts of the handling mechanisms were used. Every part is optional, if we omit them all, the exception is not handled. The same happens if there is no handling defined for an exception at all.

Whenever a subroutine defined to raise an exception is called, and handling for that exception exists in the scope of the call, the code for the call is extended automatically by the handling code. If the subroutine raises more exceptions, pre-call handling code is merged in the same order as the the exceptions appear in the declaration of the subroutine. Conditions are and-ed together, while post-handling code is merged in the opposite order as the pre-handling code. This way the handling code of the exceptions appearing later in the sequence are nested into the handling code of the exceptions appearing earlier.

3.3. Granularity

Exception reporting macros are defined in a block, their scope is the same as the scope of variables or subroutines defined in that block. In a subblock, the same name can be reused, but the reporting macro defined in the outer block can also be used by using a qualified name. Similarly, imported external reporting macros can be used as well, referred by qualified names.

At handling, almost always qualified names must by used, since it is very rare that an exception is defined in the same block or in the subblock of the caller. The name of the exception may optionally be qualified by the name of the raising subroutine as well, for the case the same exception must be handled differently when raised by different subroutines. However, even the same exception may need to be handled differently at different calls of the same function as well. For this reason we allow overriding of handlers in subblocks.

An interesting case is, if an exception must be handled the same way for a group of the subroutines that may raise it, but differently for another one. To avoid repeating of handlers, the programmer may define wrapper subroutines, which simply call the original one, but raise a different exception:

```
procedure Read ( ... ) raises IO_Int_Error is POSIX.Read ( ... );
```

When used in an object-oriented language, both reporting and handling mechanisms are inherited, but they may be overridden for new methods. It is important to note, that both mechanisms are statically bound.

4. Predicate Based Exception Handling

In this section, we discuss how exception handling can be done efficiently using *System Predicate Classes*. The basic idea is predicate based dispatching can be used to dispatch on the kind of the actual error. In the examples we focus on *POSIX* errors, thus exceptions reported by special return values and global error flags. However, similar classes may be constructed for the other mechanisms as well.

4.1. External fields in System Predicate Classes

To make predicate based exception handling possible, we first have to introduce a small extension to basic *System Predicate Classes*. This extension is not only useful in exception handling, but it is a general enhancement of the original concept with several possible applications.

Beside regular fields, we also allow global variables, or fields of other objects to be linked to classes as external fields. These fields do not occupy any space in the representation of the instances of the class, but can be used in the class as a real field. This implies that it can be used for inheritance as well.

4.2. Using System Predicate Classes for Exception Handling

To use automatically generated dispatchers, we need to define a class structure of the possible errors. The type of the error is either a regular field of the class, initialised by the constructor, or an external field. The classes must contain a virtual method for handling the exceptions, first defined in the root class, and optionally overridden in its descendants. After instantiating the class, this handling method has to be specified as the handler of the exception.

The following example shows how *POSIX* errors can be handled using *System Predicate Classes.* The method `Handler` method is abstract in the root class, and overridden in the class which handles the exception of already existing file. After classes are defined and instantiated, the same object is assigned to handle exceptions of functions returning both integer and pointer.

```
abstract class Exception is
   external Integer ErrorCode = POSIX.errno;
   virtual procedure Handler is abstract;
end Exception;

class ExistsException inherits Exception where ErrorCode = EEXIST is
   virtual procedure Handler is ...
end ExistsException;

var Error : Exception;

handle POSIX.IntError
   afterwards
      if return = -1 then Error.Handler;
handle POSIX.PtrError
   afterwards
      if return = nil by Error.Handler;
```

The main advantage of *System Predicate Classes* for exception handling instead of simple procedures containing branch instructions is that they can be extended dynamically when new kinds of exceptions are defined. If orthogonal subclasses are allowed, predicate based exception handling is still more flexible, since exception handler methods may invoke virtual methods dispatched on orthogonal predicates, thus allowing handling of the same exception differently by the same class, depending on other conditions.

5. Related Work

5.1. Programmer-Controlled Exception Handling

The idea to give the programmer control over the exception reporting and handling mechanism was presented in [1] by *Bagge*, *David*, *Havraaen* and *Kalleberg.* The paper uses the word *alert* instead of *exception* to distinguish it from the built-in exception concept of high-level programming languages. Their approach is very similar to ours, but is intended to be used in any *C*-like languages including *C* itself. In this sense, our solution is less general, since it especially extends object-oriented languages. However, it also takes advantages of it: no extra language elements are needed for refinement of its granularity, and it is more flexible due to the features of the *System Predicate Classes*.

5.2. System Predicate Classes

System Predicate Classes [2] and [4] were introduced by the authors to enable several features of object-oriented programming at the lowest level of software. With the help of overlappable reserved fields, predicate-based inheritance and polymorphism-by-value they provide a powerful *OOP*-like tool that can be used even for operating system development. An application of the technology for the programming language *C* is *SysObjC* [3].

6. Conclusion

In the paper we presented new language elements that allow separation of exception reporting and exception handling code from regular one. The novelty in these tools is that they enable the user to specify the underlying mechanism, which can be any well-known low-level method, as special return values, global error flags, callbacks, signals or long distance jumps. By separation of reporting and handling, we made it possible to use them together with existing software which uses the same mechanism, but its own way.

The exception handling side allows definition of the handling mechanism for all possible exceptions in the declaration part of the block, thus exception handling inside the block is generated by the compiler. Every instruction that may be affected by an error previously defined is extended by exception handling code. By using default scoping rules of the actual block-structured language we allow a fine granularity for the handling of different exceptions. The application of predicate classes enhances or tool to an extensible and flexible exception-handling environment.

References

[1] Bagge, A. H., David, V., Haveraaen, M., Kalleberg, K. T., Stayin' alert: moulding failure and exceptions to your needs, In *GPCE '06*, ACM Press, (2006), 265–274.

[2] Balogh, Á., Csörnyei, Z., Multiple inheritance of system predicate classes, In *MaCS '06*, Pécs, Hungary, (July 2006).

[3] Balogh, Á., Csörnyei, Z., SysObjC: C extension for development of object-oriented operating systems, In *PLOS 2006*, page Article No. 5, San José, CA, US, ACM Press, (October 2006).

[4] Balogh, Á., Csörnyei, Z., Objects and polymorphism in system programming languages: A new approach, *Periodica Politechnica Ser. Electrical Engineering*, (2007), (To appear).