A Modular Verifiable Exception-Handling Mechanism

SHAULA YEMINI
IBM T. J. Watson Research Center

and

DANIEL M. BERRY
University of California at Los Angeles

This paper presents a new model for exception handling, called the replacement model. The replacement model, in contrast to other exception-handling proposals, supports all the handler responses of resumption, termination, retry, and exception propagation, within both statements and expressions, in a modular, simple, and uniform fashion. The model can be embedded in any expression-oriented language and can also be adapted to languages which are not expression oriented with almost all the above advantages. This paper presents the syntactic extensions for embedding the replacement model into Algol 68 and its operational semantics. An axiomatic semantic definition for the model can be found in [27].

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types; modules, packages;* D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces;* D.2.5 [**Software Engineering**]: Testing and Debugging—*error handling and recovery;*

General Terms: Design, Languages, Reliability, Security, Verification

Additional Key Words and Phrases: Information hiding, module strength and coupling, compile-time checking, exception handling

1. INTRODUCTION

1.1 Modularity

The recommended approach to designing complex software systems is to develop independent program modules, each of which implements a single

This research was supported in part by the Department of Energy, Contract No. DE-AS03-76SF0034 P.A. No. DE-AT0376, ER70214, Mod. A006.

Authors' addresses: S. Yemini, IBM T. J. Watson Research Center, P. O. B. 218, Yorktown Heights, NY 10598; D. M. Berry, Computer Science Department, 3531 Boelter Hall, UCLA, Los Angeles, CA 90024.

procedural or data abstraction. Each procedural abstraction is either a subroutine or a function. Each data abstraction provides a collection of procedural abstractions that can be applied to the data items defined in and hidden by the module. A data abstraction is either an abstract data type or an abstract data object, depending on whether the module provides a type or a hidden data object to its users. The abstractions offered by a module are those whose names are *exported* from the module. The exported names of a module include its own name as well as those of the types and procedural abstractions contained inside a data abstraction module. These modules can be individually documented, programmed, tested, verified, and then used in different applications.

In order for this process to work, it is necessary that the modules possess a high degree of what is termed *modularity*. It is not the goal of this paper to define "modularity," as it is adequately defined and described elsewhere (e.g., [6,18,19,21]). Thus, only the specific properties of modularity used in this paper are alluded to here. This paper takes the position that a group of modules have modularity if they practice *information hiding*, if the modules have high *intramodule strength*, and if they have low *intermodule coupling*. The modules of a system are said to practice information hiding if they hide implementation details to the extent that if a single change is made to the implementation of a system, then only one module's contents need to be changed. A module is said to have high intramodule strength if it is devoted to implementing one and only one abstraction, be it procedural or data. A set of modules has low intermodule coupling if the data passed between them are minimized and comprised only of single (abstract) objects.

While modularity is itself impossible to measure, and thus impossible to enforce algorithmically, there are a number of syntactic mechanisms that can be used to help achieve modularity.

Scope rules can be used to restrict visibility of information that should be hidden. This is taken advantage of in many languages, in which the construct for implementing data abstractions has means to state which of the declarations inside it are exported and which are not. Compilers for such languages are able to check that no abstraction is used unless it has been defined and is visible in the scope.

Type rules can be used to ensure the consistency of interfaces. This enables a compiler to check, for example, that all invocations of a particular abstraction have the same number and same types of parameters as each other and as the definition of the abstraction. As a result, it becomes possible to check the consistency of interfaces at compile time, at link time, or at some other time prior to execution. It is even desirable to be able to check for interface consistency on the basis of only the exported parts of the modules so that the interface can be demonstrated consistent even *before* coding of the modules begins.

1.2 Exception Handling

One of the strengths of the module construct derives from its ability to provide useful, reusable abstractions. For this purpose, a module must be designed to meet the most general common needs of potential module users. This generality is also a source of weakness, since some situations do not admit the definition of a generally useful result within the module.

Each procedural abstraction or *operation* is programmed to perform some computation on inputs in its domain of definition. These inputs are characterized by that operation's *normal case* input assertion. When the program has been verified with respect to the normal case input-output assertions, applying the operation to inputs satisfying the normal case input assertion should yield a result satisfying the prescribed output assertion.

However, what happens if the operation is applied to an input that does not satisfy the normal case input assertion? Then some out-of-the-ordinary processing must be done. The normal case input assertion for read from a sequential file is that there remain records to be read from the file. If none remain, that assertion does not hold, and something other than reading the next record should be done. Other examples are dividing by zero, popping an empty stack, selecting a substring using an index out of the string's bounds, or applying a merge expecting sorted files to an unsorted file.

Forcing the invoker of an operation to apply a check for the operation's applicability prior to each invocation of an operation gives rise to several problems. It leads to cumbersome code in nested expressions; the operation may be performing the check anyway; the check could be easier to perform during the application of the operation; and some exceptions (e.g., those due to real-time conditions, such as resource depletion) cannot be usefully checked for in advance. In addition, program reliability requires that an operation be designed to handle all possible inputs of the correct data type without crashing, even when the normal case input assertion does not hold.

Therefore the approach taken here is to have the input assertion for an operation assert nothing beyond the information that can be obtained by a static examination of the program text, and have the operation itself check for its own applicability, identify exceptional cases, and notify its invoker about them. Those states not satisfying the normal case input assertion of the operation are called *exceptions of that operation*, and the operation is considered the *signaller* of these exceptions. The predicate describing the exceptional state is called the *exception condition*. Exceptions are characterized by a pair of assertions, the exception condition, and the corresponding *resumption condition*. The resumption condition describes the condition that any handler resuming the signaller must satisfy in order for the operation to eventually satisfy its normal case output assertion. (Formal definitions of these concepts can be found in [26].) Notifying the invoker of an operation of an exception is called *signalling* the exception.

The full significance of an exception may not be apparent in the context of the detecting module (otherwise it would not be an exception). Rather, it is the invoker of the operation which knows the purpose of the application of the operation and therefore the significance of the detection of the exception. Therefore, the processing under exception conditions is left to the invoker of a signalling operation. The code for dealing with an exception condition, that is, for responding to the signalling of an exception, is called the *exception handler*.

Indeed, this is the very reason that exception handling is critical for support of modularity. Without it, too much information is not hidden and coupling is high. Either the signaller has to be told more about what the invoker is doing, so that the signaller can do what the invoker would want done, or else the invoker has to be given more implementation details so that it can do the exception checking [20].

1.3 Modular Exception Handling

The previous subsections have discussed modularity and have demonstrated the necessity to deal with exceptional conditions in the invocation of module operations in order to preserve the desired modularity. In addition, the importance of early checking of interface properties of the modules has been stressed. Therefore, it is desirable to find a modular means to handle exceptional conditions which can be checked for interface consistency at compile time. By a modular means for handling exceptional conditions is meant one that permits practice of information hiding and maintenance of high intramodule strength and low intermodule coupling. This paper proposes one such exception-handling scheme with the desired properties. In addition, the scheme is simple enough that it can be axiomatized so that the software using it can be subjected to formal verification of correctness [27]. This proposal has its origins in earlier work by the present authors and others [1].

The second section lists the criteria for a modular verifiable exception-handling mechanism. The third section reviews previous attempts to build exception-handling schemes. The section after that introduces, by way of an example, the replacement model. The replacement model places some requirements on the containing language, which are enumerated in Section 5. Such a language is assumed, and then in Section 6 the details of the scheme are presented, describing its syntax, its context conditions, and an operational semantics. Section 7 evaluates the proposed scheme relative to the requirements established earlier and discusses other issues orthogonal to the scheme, which are nevertheless important. Section 8 concludes the paper.

2. REQUIREMENTS

An exception-handling mechanism must be able to support flexible responses of invokers to the detection of exceptions without compromising modularity. The challenge is that these goals are somewhat conflicting, since for maximum flexibility of response it is important to allow communication of as much information as possible about exceptions. The communicated information and the invoker's response, however, must be restricted in order to maintain low coupling and information hiding. In subsequent subsections, the specific elements of these requirements are enumerated and discussed.

2.1 Orthogonality

A programming language is said to be orthogonal if it is constructed from a small set of primitive features whose functions have no overlap and which can be composed arbitrarily, with no or few exceptions, to obtain its full coverage.

Exception-handling constructs should be orthogonal to the rest of the language so that, when they are not used, the rest of the language is unaffected, and so that the rest of the language may be used arbitrarily in conjunction with the exception-handling constructs to build its flexible responses and communication. An example of lack of orthogonality is the inability to handle exceptions in expressions.

Although many disagree with us, we believe that the restrictions born of the lack of orthogonality in a language are a drawback far outweighing the seemingly excess generality resulting from orthogonality in a language. Excess generality can simply not be used, while restrictions must be known, adhered to, and sometimes programmed around, even if they are present for the purpose of keeping programs simple. A fuller discussion of the advantages of orthogonality in keeping a language simple, small, and clean may be found in [23].

2.2 Handler Responses

The following varieties of handler responses to an exception can be identified in the literature [1,7]:

1. *Resume the signaller*: Do something, then resume the operation where it left off.
2. *Terminate the signaller*: Do something, then return a substitute result of the required type for the signalling operation; if the operation is not a value returning operation, this reduces to doing something and returning to the construct following the invocation of the operation. This includes using alternative resources, alternative algorithms, and so on.
3. *Retry the signaller*: Do something, then invoke the signaller again.
4. *Propagate the exception*: Do something, then allow the invoker of the invoker of the signalling operation to respond to the detection of the exception.
5. *Transfer control*: Do something, then transfer control to another location in the program. This includes doing something and then terminating a closed construct containing the invocation.

For maximum flexibility, an exception-handling mechanism shculd be able to support all of the above handler responses.

2.3 Parameterization

When exceptions are signalled, it should be possible to pass parameters to whatever handler is prepared to field the exception. This in turn requires that handlers have formal parameters. Without this capability,

1. either a handler must make use of global variables, if indeed such exist and are visible, in order to determine the circumstances under which it is invoked, thus increasing the coupling between the signaller, the invoker, and other unrelated sections of the program that can see the global variables, or
2. a separate handler for each possible raising point must be provided, which knows its unique circumstances for raising. This decreases strength because

many more handlers with similar, though not exactly the same, function must be provided.

2.4 Explicit Propagation of Exceptions

Automatic propagation of unhandled exceptions along the chain of invokers permits violation of information hiding when exceptions are propagated through levels of abstraction, and it can thereby increase coupling. As an exception is propagated, it reveals information about the implementation of the abstraction its original signaller is implementing. In addition, the propagated exception may very well be meaningless in contexts to which it is propagated. Finally, it is impossible to determine at compile time the handlers that might field the exception, even if all handlers are constants, thus making compile-time checking of interfaces impossible. As a typical example of this situation, consider the exception *subscript out of bounds* being propagated to the invoker of a function using stack operations. The parameter of this exception is the invalid subscript. By getting this exception, the invoker of the stack operation is told that an array is used to implement the stack. There is very little the invoker of the stack operation can do in terms of stacks to handle the *subscript out of bounds* exception. Finally, even if the invoker of the stack operation were to anticipate the possibility of the *subscript out of bounds* exception, there is no guarantee that its handler would be used, as it is possible that another routine invoked later, and thus ahead of it on the invocation chain, provides a handler for the same exception. There is little reason to assume that this other handler's effect will be the one desired by the stack user.

It is true that not every invocation represents a change in level of abstraction, that is, a visible function *f* of a module may call hidden functions to any invocation depth unbeknownst to the invoker of *f*. However, even here automatic propagation is inappropriate for two reasons. First, to the return mechanism, there is no distinction between the two kinds of functions. Second, it would be particularly bad if the invoker of *f* were to get an exception raised by one of the hidden functions that was not translated to abstraction-relevant terms by *f*. The cleanest, most uniform way to insure that the exception will get properly translated is to disallow all automatic propagation.

In cases in which the exception can be meaningfully propagated to the invoker's invoker, it is as a *different* exception relevant to the level of abstraction visible to the invoker's invoker. To achieve this, the handler of the first exception can signal an exception which is visible in the scope of the invoker's invoker. This exception is likely to have a different identifier and a different set of parameters from the original exception. In the case of the stack operation which invokes array subscripting, a *subscripts out of bounds* exception might be propagated to the stack operation's invoker as the parameterless exception *stack underflow* which *is* relevant to the user of stacks.

In summary, automatic propagation of unhandled exceptions may compromise information hiding, while explicit propagation can be used to properly rename propagated exceptions. The exception-handling mechanism should therefore not provide automatic propagation and should force the users to explicitly rename any propagated exception.

2.5 Scope Rules and Compile-Time Checking

The language's scope rules and type system should be taken advantage of, to allow enforcement of information hiding and checking the consistency of the interfaces by the language compiler. Above, it was argued that automatic propagation should be disallowed in order to permit information hiding. By appropriate choice of visibility of exception names, it is easy to enforce this rule. Automatic propagation would mean taking the exception name out of the scope in which it is visible. When automatic propagation is disallowed, the set of handlers that can field a particular exception can be statically determined, thus allowing additional compiler checks. The requirements that can be checked by extending type and scope rules to the exception-handling constructs include:

1. checking that each exception is signalled with the correct set of actual parameters,
2. checking that each handler for an exception is defined with the correct set of formal parameters,
3. checking that only those exceptions that are defined by a signaller are signalled by that signaller, in effect enforcing the explicit propagation requirement, and
4. checking that all exceptions that can be raised in a given scope are handled in that scope.

2.6 Verifiability

Exception-handling features should be such that it is possible to give axioms and rules of inference for them. Supplying these axioms and rules of inference will enable programs using them to be subjected to formal verification. Being able to supply the axioms and rules of inference requires that the features be simple and orthogonal to all other language constructs.

3. PREVIOUS WORK

There has been much work on the topic of exception handling over the past dozen years. This work has provided the basis for our proposal, and in fact, the requirements given in the previous section were obtained from our examination and synthesis of this work. This section describes this previous work briefly and points out the bases of the various parts of the present proposal. First the philosophical, then the linguistic, and finally the axiomatic bases are described. A more detailed discussion can be found in [26]. Section 7, which evaluates the present proposal against the requirements, includes some discussion of these earlier proposals.

Parnas [20] lays the foundation for the present proposal by pointing out that exceptions are as much a part of an abstraction as are the data types, the data objects, and the operations of the abstraction. Thus, the specification of an abstraction includes a list of the exceptional conditions that may be signalled by the operations of the abstraction. In addition, an exception is as much a part of an operation that detects the condition as are its parameters and return values, if any. Also, it is Parnas who points out that it is the *invoker* of the operation that signals an exception that is in the position of determining the proper handler response because only it knows to what use the results of the operation are being put.

There are several actual and proposed collections of exception-handling features in existing and proposed languages.

PL/I's exception-handling scheme [8] provides nonparametrized exceptions and generally supports resumption, although determination of flow control after handling is often exception-specific [16]. The ability to retry, available on some built-in exceptions, is not extended to programmer-defined exceptions. Exceptions are automatically propagated along the invocation chain until a corresponding handler is found. Because the exceptions are not typed and their scope is global, there is very little that can be checked at compile time.

Goodenough's proposal [7] catalogs a large variety of useful handler responses for nonparameterized, nontyped exceptions. The proposal also describes how the signaller may wish to require or disallow the various kinds of handler responses. Exceptions must be explicitly propagated along the invocation chain. This proposal has become the standard against which other proposals are compared for flexibility.

Liskov and Atkinson describe exception handling in CLU [12,14]. CLU [13] is one of the first languages providing explicit features for constructing data abstractions. Their purpose in providing exception handling is to support software which is able to respond reasonably to wide variety of circumstances (i.e., which is fault-tolerant). Their exception mechanism is based on a simple model of exception handling that is anticipated to lead to well-structured programs. Accordingly, only statements and procedures can raise exceptions; expressions cannot, except to terminate the innermost containing statement. The location of a handler in a program text determines both the scope in which that handler is designated and the location to which control will transfer after handling. Consistent with the goal of simplicity, only termination handlers are supported. Parameterization of handlers is supported. Exceptions must be explicitly propagated along the invocation chain. The mechanism is strongly typed, exceptions appear in the external interface specification of their signaller, thus allowing a compiler to identify where they can be raised.

Levin's proposal [11] provides exceptions with parameters and resume-only handlers. There is no automatic propagation of exceptions along the invocation chain so exceptions must be explicitly propagated. Like CLU's mechanism, it provides extensive support for compile-time checking. Levin supplies axioms and rules of inference for his proposal. Apart from exceptions associated with invocation of operations, Levin also considers exceptions which are associated with the state of shared data structures in a concurrent programming environment. Such exceptions may be propagated to all potential users of the data structure and thus can be used to build daemons. This is a powerful notion that is not included in the present proposal; although Section 7.4 discusses this notion further.

Ada's[1] mechanism [9] provides nonparameterized exceptions with termination-only handlers. The lack of parameters in exceptions is particularly aggravating in the case of Ada; very often the nonlocal access necessary to determine the circumstances of signalling allows the program to determine how the implementation has passed parameters, rendering the program *erroneous.* Exceptions are automatically propagated along the invocation chain until a corresponding handler, if one exists, is found. Exceptions are not declared in the external interface of their signallers, and are not typed. Thus, there is very little that can be checked at compile time. Ada's exception mechanism cannot be subjected to verification without major changes [15]. Specifically, it is necessary to require explicit propagation of exceptions to invokers when they are not handled locally.

Mesa's exception-handling mechanism [17] provides parameterized exceptions and supports all the listed handler responses. Mesa's exceptions are automatically propagated arbitrarily along the invocation chain. Since exceptions are not associated explicitly with their signallers, some of the language rules and restrictions are not compile-time checkable. Instead, there is an elaborate run-time checking mechanism.

Cocco and Dulli [2] propose a scheme providing parameterized exceptions with the full complement of Goodenough's proposals of the control flow during handling. That is, the exception can require the handler to terminate, forbid the handler from terminating and require it to resume the signaller, give the handler the choice between the two, and arrange that the signaller be retried. The signaller can either raise an exception for local handling or exit-raise it for handling by the invoker. Curiously, exit-raised exception handlers cannot resume their signallers. Exceptions are not propagated implicitly, and the full declaration of exceptions allows compiler detection of the failure to handle exceptions. They provide examples of the scheme's use and axioms to allow proving the behavior of programs making use of their proposal.

Several authors have investigated formal specification of exception handling in order to allow programs using it to be verified to do what they are claimed to do. In fact some have even used verifiability as the basis of their design. Berry, *et al.* [1] describe an earlier version of the present proposal and how it might be axiomatized. This axiomatization allows both proof of the behavior of the using programs and determination as to when certain kinds of handler responses are to be allowed. For example, the signaller of an exception specifies the exception condition itself as the precondition of a handler and what is required from the handler before resumption as the postcondition of the handler. Only handlers that satisfy that postcondition, given the precondition, are allowed, thus insuring that handlers do what is necessary before resumption.

[1] Ada is a trademark of the U. S. Department of Defense (AJPO).

Cristian [3,4,5] has proposed a fully axiomatized set of exception-handling features that allow construction of and reasoning about robust programs which achieve robustness through exception handling. In these proposals, the specification of a data type includes a list of exceptions as well as the operations of the type. Any operation, procedure, or function is viewed as a multiexit construct with one standard exit and possibly a number of exception exits. The specification of such a construct includes a full specification of the standard and all the exception entry conditions together with the exit effect of the construct in each case. Associated with each exceptional exit is a specification of the conditions a handler must satisfy under the assumption of the exceptional condition. In this way, the signaller of the exception can force particular handler responses such as resumption, termination, and so on. Cristian uses a predicate transformer approach to specifying the features and their semantics.

Luckham and Polak [15] attempt to axiomatize the exception-handling facilities of Ada and find that only with major changes can it be axiomatized. Specifically, exceptions cannot be propagated arbitrarily along the invocation chain and further exceptions must be propagated explicitly to invokers.

As mentioned above, Levin's [11] and Cocco and Dulli's [2] proposals are accompanied by sets of axioms describing the semantics of their features and allowing proofs of correctness of programs using them.

The bases for the present proposal are clear. The present proposal differs from the previous ones in that it constitutes an attempt to play the orthogonality game to the hilt. That is, we try to obtain as much capability as possible at the cost of as few new primitives as possible. The power is obtained by picking the primitives in such a way that no restrictions on their composability with each other and the rest of the language is required or even desirable. When done properly, it is possible to avoid synergistic problems resulting from the combination of ill-fitting features and to get full coverage of the desired capabilities just by combination of the few primitives. In fact, we claim that by starting with a strongly-typed, expression-oriented language such as Algol 68, and adding an encapsulating construct such as the Ada package, one can gain all the coverage implied by the requirements of the previous section, with the addition of only one new closed construct, **on** ... **no**, for handler bodies (strongly resembling procedure bodies), one new type constructor, **handler**, one new completer, **replace**, two new items of punctuation, **signals** and **exc**, and an extension to the return types of units to indicate that they may signal exceptions. All the rest is done by use of mechanisms, such as procedure invocation and return, that already exist in the language. Furthermore, the new features are described by just two new axiom schemes.

4. THE REPLACEMENT MODEL

The replacement model of exception handling adopts an expression-oriented view. A program is a composite expression; an exception is explicitly associated with an expression—its potential signaller; a signaller's exception corresponds to a subexpression which could not be completed in the signaller; and the handler's result replaces the result of the subexpression, or the result of the signaller invocation—hence the name "replacement model." In this context "results" mean both side effects and return values.

The reason for taking an expression-oriented view is that it provides a unifying model for exception handling in both statements and expressions. Exception handling naturally causes side effects in expression evaluation. Anything done by a handler for an exception signalled within an expression is a side effect of evaluating that expression. Expressions must therefore be allowed to produce side effects if useful exception handling is to be supported.

Since an expression-oriented model contains a statement-oriented model as a special case, a mechanism developed for an expression-oriented language is also applicable to a statement-oriented language, although it may lose some of its power in a strict statement-oriented setting such as CLU [13]. However, the replacement model, even when restricted to a statement-oriented language, is still more powerful than other existing or proposed mechanisms in statement-oriented languages because of the handler responses it supports.

5. EMBEDDING LANGUAGE

Support for modularity, the requirements for the exception-handling mechanism, and the choice of the replacement model place a number of requirements on the programming language in which the proposed exception-handling mechanism is embedded. The language must have features for building procedural and data abstraction modules. It must be orthogonal and strongly (compile-time) typed, and it must have been axiomatized. Finally, it must be expression oriented in order to permit side effects in all expressions. Basically, no existing language meets the requirements. However, Algol 68 [25] meets all the requirements except that of having data abstraction modules.[2] However, since the language is orthogonal, it is possible to superimpose an orthogonal modules facility to correct this defect. Ada has such a facility in its packages. A package is basically a means to group an arbitrary collection of declarations of any kind into a single unit such that some of the declarations, under the programmer's explicit control, are exported and the others are kept hidden. If the package exports a type, then it builds an abstract type. If the package has hidden data and exports only operations, then it builds an abstract object. Thus all a package does is to make some declarations that would normally be visible, were the package not there, invisible. All other aspects of the declarations are unchanged, and hence orthogonal to the notion of the package.

Therefore, this paper uses Algol 68 extended with Ada packages as the vehicle to carry its exception-handling proposal. The vocabulary used here attempts to steer clear of the more esoteric vocabulary of the Algol 68 Report in favor of a more conventional explanation.[3]

As an introduction to the mechanism and to the language in which it is embedded, consider the Algol 68 procedure *convert* in Figure 1, which takes an array-of-integers variable as a parameter and returns the string of the integers'

[2] For an axiomatic semantic definition of Algol 68, see [24].

[3] There is, however, the danger of being sloppy in the explanation.

```
proc convert=(ref[ ]int code)string:
begin
    string s:="";
    for i from lwb code to upb code do
        int code i=code[i];
        s:=s+if code i≤char hi and code i≥char lo then
                    repr code i
                else
                    ???
                fi
        # append to s the char represented by code[i]
            if code i is in the defined range #
    od;
    s
end
```

Figure 1

character representations. *Convert* invokes the operator **repr**, which returns the character represented by an integer if the integer is in the range [*char lo*,*char hi*]. The data type [ ]**int** of Algol 68 is that of one-dimensional integer arrays. The symbol + is the string concatenation operator. *???* stands for the case in which an element of *code* does not lie in the expected interval, that is, does not represent a valid character code. The **if then else fi** either returns the required character or falls into *???*. The value yielded by this conditional is concatenated onto the current string. This example is used in the discussions of the next section, in which *???* is replaced by an explicit signalling of an exception.

6. EXCEPTION HANDLING IN THE REPLACEMENT MODEL

This section gives the full syntactic and and semantic details of the replacement model of exception handling as embedded in Algol 68 extended by Ada packages. The discussion is motivated by expressing the example procedure of the previous section using the proposed exception-handling features. This new procedure is given in Figure 2.

In the new example, *???* has been replaced by a signalling of the exception *badcode* with the integer *code i*, for which no representation exists, passed as its parameter. *Convert* is declared as possibly signalling an exception *badcode*, and thus any invocation of *convert* must have a handler for this exception associated with it. The declaration says that any handler for *badcode* is prepared to accept a single **int** parameter, namely the code that cannot be represented, and will return either a **char** value to use as the representation anyway or a **string** value to use as the final value of the invocation of *convert*. Thus, if no integer that is not representable is ever found, then *convert* returns its normal result, that is, the accumulated **string** value of *s*. The subsequent subsections describe all the parts of this example.

6.1 Exception and Handler Types

In the declaration of the procedure *convert*, in its **signals** clause, the exception

```
proc convert=(ref[ ]int code)string
        signals(exc(int)(char,string)badcode):
begin
    string s:="";
    for i from lwb code to upb code do
        int code i=code[i];
        s:=s+if code i≤char hi and code i≥char lo then
                    repr code i
                else
                    badcode(code i)
                fi
        # append to s the char represented by code[i]
            if code i is in the defined range #
    od;
    s
end
```

Figure 2

badcode is specified to require passing one parameter to any handler designated for an invocation of *convert*, namely, the **int** value of the invalid code. The handler may either

1. return a **char** value to be used as the value of the expression that detected and signalled the exception, thus resuming the execution of invocation of *convert*, or,
2. return a **string** value to be used as the final value of the invocation of *convert*

The first possible return value is known as the *resumption* value and the second is known as the *replacement* value. Every exception is specified as returning one of each kind of value.

Later, handlers will be provided for responding to these exceptions. Each handler provides both kinds of return values. Handlers in the replacement model are considered as objects of the data types formed by the type constructor **handler**. These types carry the types of the parameters and the two return value types. Handlers are typed very much like procedures, except that there are two return value types instead of only one. As with procedure identifiers and their bodies, type checking is used to make sure that all handlers associated with a particular exception are prepared for the correct parameters and provide the correct return values.

6.2 Signalling Exceptions

The semantics of signalling an exception is very similar to that of calling a procedure. Consequently, signalling an exception has the same syntax as calling a procedure, namely first is given an exception identifier and then if there are actual parameters to be passed to the handler, they are given in a parenthesized list. Of course, it is necessary to check that the actual parameters are of the types required by the exception's type and that the context of the signaller is proper for a value of the resumption type of the exception.

In the example, the signalling of *badcode* has a single **int** actual parameter, as is required, and sits in a position in which a **char** is required, since it is the expression of the **else** arm of a conditional of type **char**.

6.3 Associating Handlers with Exceptions

It is important to devise rules for associating handlers with exceptions so that it is possible for a compiler to check whether or not handlers have been supplied for all exceptions that can be raised. The immediate invoker of an operation is considered responsible for determining how to handle that operation's exceptions. Therefore, handlers for all exceptions signalled by an operation are required to be visible within the *static* environment of each invocation of that operation.

Handlers are designated for an invocation by attaching an **on** clause, containing designations of handlers, to some closed construct (e.g., block, loop, conditional, etc.) statically containing that invocation. The specific handler designated for a given exception of a given signaller invocation is that appearing in the **on** clause attached to the innermost enclosing closed construct designating a handler for that exception. The place where the **on** clause is attached is completely independent of the flow control after handling and serves only to delimit the scope within which the handler is designated, in a manner consistent with normal static scoping of identifiers. Thus, attaching an **on** clause to a particular closed construct *C* and designating in it a handler *H* for an exception *E* of an operation *O* is tantamount to putting a block around each invocation of *O* in *C* and attaching to each a copy of the same **on** clause designating *H* for *E*.

Figure 3 shows the skeleton of a block that contains several invocations of *convert* and which has an **on** clause containing a handler *H1* for *badcode*. Inside this block is another block with still more invocations of *convert* and with its own **on** clause containing a handler *H2* for *badcode*. If the first, second, and fifth invocations of *convert* signal *badcode*, then *H1* will be used, if the third and fourth invocations signal, then *H2* will be used. If *H1* and *H2* are terminating handlers, then as each invocation of *convert* signals *badcode*, that invocation alone is terminated and control transfers to the construct just following that invocation. This is as if each invocation had its own private copy of the designated handler attached to a block surrounding only the invocation.

In Figure 3, if no other handler designations for *badcode* appear in the program containing the outer block, and the outer block's **on** clause were not present, then the compiler would complain that the first, second, and fifth invocations of convert do not have handlers for *badcode* designated for them. The compiler can do this because *convert* is declared as possibly signalling *badcode*.

This proposal is in contrast to CLU and Ada, in which the construct to which a handler designation is attached is also the construct that is terminated. In these languages, difficulties arise in the case where two invocations of the same signaller appear in the same statement, different handling actions are required, and control is then required to transfer to the same location. By considering the place of the designation of handlers orthogonal to the places involved in the flow of control of exception handlers, the problems observed in CLU and Ada

```
begin ...

    #1# convert(...)

    #2# convert(...)

    begin ...

        #3# convert(...)

        #4# convert(...)

    end
    on badcode=H2
    no

    #5# convert(...)

end
on badcode=H1
no
```

Figure 3

have been avoided.

The static binding mechanism described above is general enough to provide for system- and globally-defined exceptions that are available for signalling throughout the program and for providing system- and globally-defined handlers that are used when no other handler has been provided by the user's program. All user programs are assumed to be nested inside a standard prelude block which, among other things, declares the so-called predefined types, constants, variables, procedures, and functions (Ada has the package *STANDARD* serving the same function). It is a simple matter to declare all of the language-defined exceptions such as *overflow*, *constraint error*, and *failure*, and to provide handlers for these exceptions in the prelude block. These exceptions and handlers are visible in any program scope not declaring the same identifiers, and such handlers are used in any scope that does not supply another handler for these exceptions. Thus, these handlers serve as default handlers for these exceptions. The default handler for the language-defined exceptions could be defined to print a message and terminate the program; such default handlers would suffice in many cases. In fact, the programmer who does not declare any exceptions need never supply a handler for any exception.

One additional boon of this proposed mechanism is that when handlers are provided for these exceptions they can be made to terminate constructs other than the global one or any other one to which they happen to be attached. This is because the flow of control after finishing a handler is determined by means other than the physical placement of the handler. That is, it is determined solely by what is written inside the handler itself and what is written at the signalling site. This is no different from for procedures in which the flow of control after finishing a procedure body is independent of where the body is written and is a function of the texts of the body and the calling site.

Still another boon of the proposed mechanism is that the ability to have a default handler for an exception is not restricted to the system-defined procedures. For any exception of any procedure, a handler can be attached to the same block declaring the procedure. This handler is thus visible at all places in which the procedure and its exceptions are visible, except for all places lying within the scope of an internal unit to which another handler is attached. Thus, the system default handlers are using a generally available mechanism.

6.4 Signaller Types

It is necessary to ensure that invokers of *convert* are aware of the fact that *convert* may signal an exception, called *badcode*, of the given type, so that they supply a suitable handler. In order to explicitly identify an operation as being a *signaller* of a given set of exceptions, its exceptions must be mentioned in a **signals** clause in its heading introduced for this purpose. Thus, in Figure 2, *convert* is declared as a signaller of *badcode.*

In order to support replacing subexpressions uniformly at any level, without also coupling the effects made by a handler to the flow control required after the handling is completed, any closed construct in the embedding programming language is allowed to become a signaller of exceptions. This is done by declaring the construct's exceptions in a **signals** clause following the opening bracket of the construct. Thus, for example, a block, loop, procedure, or conditional can become a signaller of exceptions. The rules for handling exceptions apply uniformly to procedure signallers and any closed construct signallers.

A **signals** clause on a construct declares the exception identifiers in the construct. A signalling of an exception identifies the declaration of that exception in the **signals** clause of the innermost-containing closed construct declaring that exception. In this manner, the rules of static scoping are used to ensure that only exceptions defined in the given scope are signalled and that all signalled exceptions are handled. The first of these helps enforce information hiding. An exception cannot be signalled in a context in which the exception identifier is not visible, and this restriction is enforceable at compile time. Since handlers can be defined in the context of the invoker, they can propagate exceptions of an invoked operation as exceptions of the invoker.

The identifiers, the parameter types, and the two return value types of the exceptions signalled by a signaller are made part of the signaller's type. This enables a compiler to identify where each exception may be signalled, even when a signaller is passed as a parameter out of its defining environment, so that the compiler can check whether or not there exists a handler designation for that exception in the enclosing static environment. In addition, the compiler can check that any such handler has the proper set of parameters and the proper pair of return values. (Note that including identifiers as part of the type is similar to the inclusion of the identifiers of the components of a structured (record) type as parts of the type.)

6.5 Handler Responses in the Replacement Model

A handler in the replacement model may either

1. replace the signalling expression, thus resuming the signaller, or
2. replace the signaller invocation, thus terminating the signaller.

Resumption of the signaller happens if the handler returns normally to its signaller.

Termination of the signaller is indicated by using a **replace** completer[4] within the handler body. The value returned by the completer is used to replace the value the signaller would have returned. The addition of the **replace** completer suffices to allow supporting all of the handler responses enumerated above. This is demonstrated by the following examples.

1. *Resumption*: Supply a "?" as a replacement for the **char** corresponding to *badcode*. *Convert* then resumes. The result is therefore a string in which the characters corresponding to unconvertible codes are "?":

```
do
    ...
    print(convert(nums))
    ...
od
on badcode=(int i)(char,string):
    "?"
no
```

2. *Termination of the signaller*: Supply the the empty string as a replacement for the **string** returned by *convert*. Note that the replace type of an exception is always the type returned by that exception's signaller:

```
do
    ...
    print(convert(nums))
    ...
od
on badcode=(int i)(char,string):
    "" replace
no
```

3. *Retry*: Retry after changing the *badcode* to a zero. The **string** returned by the new invocation of *convert* is returned as (a replacement for) the value of the original invocation:

```
do
    ...
    print(convert(nums))
    ...
od
```

[4] A completer is a value returning exit which causes some specific enclosing construct to be exited while returning as the construct's value the value of the expression just preceding the completer.

```
on badcode=(int i)(char,string):
   begin
      nums[i]:=0;
      convert(nums) replace
      # call convert again,
      to replace the result of the previous call #
   end
no
```

4. *Termination of a closed construct and exception propagation*: Terminate a block and propagate *badcode* as another exception:

```
begin
   do signals (exc (char,void)finish)
      begin
         ...
         print(convert(nums))
         ...
      end
      on badcode=(int i)(char,string):
         finish
      no
   od
end
on finish=(char,void):
   skip replace
no
# when finish is signalled, replace its signaller's
   invocation by the value yielded by skip #
```

(**Skip** in Algol 68 yields an undefined value of whatever type is required by the context.) This example demonstrates two of the handler responses supported by the replacement model: termination of a closed construct containing the invocation signalling an exception (as in [9,12,14]) and modular exception propagation.

In order to obtain termination of the loop after *badcode* has been detected, the loop is made a signaller of a parameterless exception called *finish.* This is done by attaching a **signals** clause after the **do**, which declares *finish. Finish* is signalled in the handler for *badcode*, that is, it is the propagation of *badcode.* When the handler for *finish* replaces the invocation of *finish*'s signaller (i.e., the loop invocation), the loop is terminated, as required. Thus, the signalling of *finish* in this example is an example of propagating the exception *badcode* in a modular manner. The exception *finish* is an exception of an invoker of *convert* (i.e., the loop that is *finish*'s signaller).

5. *Transfer of control*: The handler's body contains some transfer of control such as a **goto** or an **exit**:

```
do
   ...
   print(convert(nums))
   ...
od
```

```
on badcode=(int i)(char,string):
    goto print error message
no
```

Note that the handler in this case has the same parameter type and the same return value types as in all previous cases. (A **goto**'s type is determined in the same way as a **skip**'s.) In this situation, however, no value is ever returned. Rather, the goto causes abrupt termination of all invocations entered after entry to the block in which the label is declared. It is recognized that this feature will not be used very often, but, due to the orthogonality requirement of the language, it is available.

6.6 SYNTACTIC EXTENSIONS FOR THE REPLACEMENT MODEL

The previous subsection introduced most of the syntactic extensions to Algol 68 required for supporting the replacement model. A formal definition of these extensions can be found in [26]. This subsection points out features that were not exposed by the examples.

The **on** clause may contain an arbitrary expression yielding a handler of the type corresponding to the exception. New types can be constructed using the mode constructor **handler** in the same manner as other mode constructors such as **proc**. In particular, it is possible to have handler variables, handler pointers (of types **ref handler** ... and **ref ref handler** ...), procedures yielding handlers, and so on.

An **on** clause is considered nested at the same nesting level as the construct it postfixes (i.e., next to it), not inside it as in Ada or CLU. This avoids the problem that a handler may be invoked before some of the objects it references have been allocated.

There may be more than one occurrence of **replace** in a handler, and **replace** can appear within any expression. **Replace** is considered as a completer for the smallest enclosing handler text, and can be nested arbitrarily deep within a handler text.

Still another issue is what to do with two operations that have the same name exception. This can be dealt with by appeal to a language feature that is orthogonal to those discussed in this paper. Namely, this can be considered as a case of overloading of identifiers, which is to be resolved by context. Thus if it desired to have *overflow* exceptions for several operations of the same type, then *overflow* becomes overloaded and the context in which *overflow* is mentioned becomes significant in determining which *overflow* it is. This context would probably include the invoked operation, the parameters passed, and the two return types expected. Further discussion is outside the scope of this paper. In the same category is the issue of using the language's coercion system to help the user abbreviate some of the writing that must be done in using the exception-handling mechanism.

6.7 SEMANTICS

An axiomatic semantic definition of the replacement model exception-handling mechanism can be found in [26, 27]. An operational definition appears here, using the standard stack model implementation of block structured languages [10, 22].

6.7.1 *Extending the Stack Model for Exception Handling.* The declaration array of an activation record (AR) normally contains cells for the block marker (static and dynamic links), the return label, one cell for each parameter, and one cell for each locally declared identifier. The declaration array of a signaller is extended to include one cell for each exception declared by that signaller. The cell for an exception contains the (instruction pointer, environment pointer), that is, the (ip, ep) pair of the handler bound to that exception.

The declaration array of a handler contains cells for the block marker, the return label, one cell for each parameter, one cell for each locally declared identifier, and one cell for the replace label, designating the place to which control transfers upon encountering a **replace**.

6.7.2 *Semantics of Signaller Invocation.* Recall that any closed construct and any procedure, handler, and so on, can be a signaller. Thus signaller invocation is just the normal invocation of the closed construct, procedure, handler, and so forth, with some additional activities performed. The semantics of invoking an enclosed clause signaller is a particular case of the semantics of invoking a routine signaller. We point out the steps that are unnecessary for an enclosed clause signaller in the following.

1. Obtain the procedure value, that is, the (ip,ep) pair, to be called (unnecessary for an enclosed clause signaller).
2. Evaluate the actual parameter expressions, if any (none for an enclosed clause signaller), to get the actual parameter values, and evaluate the handler expressions to get the actual handler values. Handler values are (ip, ep) pairs. Observe that their eps are generally equal to that of the signaller's return label or to the signaller's dynamic link.
3. Allocate the new AR for the signaller.
4. Pass parameters, if any, to the AR, and pass handlers to the AR by filling the corresponding parameter and handler cells in the declaration array with the actual values passed.
5. Set the signaller's return label, (ip, ep) pair, to designate the first instruction following this invocation.
6. Reset the processor to the signaller's (ip, ep) pair and proceed to evaluate the signaller.

The evaluation of the parameters and handlers is done collaterally, that is, no order of evaluation is implied. Likewise, the binding of the parameters and handlers is done collaterally.

6.7.3 *Semantics of signalling an exception.* Raising an exception consists in doing the following steps:

1. Obtain the handler value, that is, the (ip,ep) pair of the handler designated for the exception being signalled.
2. Evaluate the actual parameter expressions passed to the handler.

3. Allocate an AR for the handler.
4. Pass parameters to the AR.
5. Set the handler's return label to designate the first instruction following the signalling.
6. Set the handler's replace label from the return label found in the same AR in which the handler's (ip, ep) pair was found. This is the AR for the signaller of the signalled exception, and, therefore, its return label should be used for **replace**.
7. Reset the processor's (ip, ep) pair to those of the handler and proceed to evaluate the handler.

6.7.4 *Semantics of Replace.* Encountering a **replace** completer results in the following steps being taken:

1. Get the value v of the expression preceding the **replace**.
2. Use the replace label of the innermost (the first, following the static chain) handler AR to reset the processor (ip,ep) and pop the AR.
3. Push v into the expression stack.

The replace label is the same as the return label of the invocation signalling the exception. Thus this semantics allows the handler to compute a replacement value and effect for the signaller of the exception.

6.7.5 *Notes.*

1. **End** or **exit** within the handler have the same meaning they have anywhere else; use the return label to reset the processor and then exit the topmost AR.
2. The binding mechanism for handlers is identical to that for actual parameters, as handlers are analogous to parameters of **proc** types.
3. Raising an exception is a call to the handler bound to the exception.
4. The access path to the replace label can be determined at compile time and is the same as that to the return label of the call signalling the exception that the handler is bound to.
5. The interpretation of **replace** is independent of the place to which the handler is attached. Thus the range of replacement is always under control of the signaller.

7. ASSESSING THE REPLACEMENT MODEL

The previous sections have already touched on a number of the issues. Specifically, it has already been demonstrated that the replacement model of exception handling as proposed is orthogonal, flexible enough to support the full range of handler responses, allows exceptions to have parameters, requires that exceptions be propagated explicitly, and supports compile-time checking of the various interfacing requirements, including that handlers have been supplied for all possible signallings. The only issue remaining is verifiability. Space does not permit development of the axioms for dealing with the mechanism here. However, they are given in [26] and [27]. It was necessary to add only two rules to the formal system developed by Schwartz for Algol 68 [24] in order to be able to deal with the exception-handling scheme presented herein. The remainder of the section deals with a more detailed evaluation of the present proposal.

7.1 Resumption

Some seem to believe that supporting resumption requires a high coupling of a signaller with its invoker, which needs to be controlled with an abundance of key words and restrictions. This is reflected in the designs of [7] and [17] and in the discussions in [12] and [14]. The following quotation is borrowed from [12].

"The complexity of the linguistic mechanism [required for the resumption model] is illustrated by Goodenough's proposal [7] which is a carefully considered design of a complete mechanism supporting the resumption model. Three types of signals (exceptions) are recognized corresponding to cases where the signaller may not be resumed, must be resumed, or where resumption is optional. In case the caller is unable to resume a signaller that must or could be resumed, a special ability is provided to permit the signaller to cleanup (restore non-local variables to a consistent state) before its activation is terminated. In addition a default mechanism is provided to permit the signaller to handle its own exception in case the caller does not."

Goodenough distinguishes three cases.

Case 1. The signaller has to resume. The usual argument given for this case is "When an operation is not resumed [when resumption is optional] ... it may be necessary first to release certain areas of storage, close files, restore data structures to a consistent state, etc." [7]. These actions are generally called *cleanup* actions. Examination of this argument reveals that there are two separate issues in cleanup actions. One is the need to restore data structures to a possible state. A careful examination of this argument in the context of modularity and verification eliminates this problem. Modularity requires that this category of cleanup actions *always* be performed in the signaller *before* signalling an exception. There are two related reasons for this. First, the exception cannot be externally specified correctly without compromising information hiding if the state is not a consistent state, since the representation at that point does not represent a valid value of the type. Also, the operations of the module, which are the only operations that a handler may apply to the data type provided by the module, depend on the state being consistent for their correct application and therefore should not be allowed to be applied in inconsistent states.

The second category of cleanup actions are those related to *finalization*. Both storage deallocation and closing files are special cases of finalization of abstract data types. Finalization includes all those operations that are required to be performed when an instance of a type ceases to exist. In [24] it is argued that obtaining finalization by having the user program *explicitly* invoke finalization operations, at the point at which the variable is about to cease to exist, does not support program reliability.

"Such an explicit finalization would be vulnerable to users who neglect to finalize their variables, and to blocks that are exited prematurely through an exception, thus tying up scarce resources. Devastating consequences could also result from premature use of the finalization procedure, leaving the variable in existence, with all traces removed from the type generator's data structures."

Their conclusion [24] is that "The finalization operation must be invoked by the underlying mechanism responsible for the deletion of the variable, so that finalization and deletion cannot be separated."

Since separation of orthogonal language features is an important goal in any language design, we believe that supporting finalization is an important but separate issue from exception handling. Finalization has to be done primarily at normal exits from a closed construct or procedure (i.e., in situations which are unrelated to exception handling). The points at which the local objects of a signaller cease to exist remain well defined when replacement model exception handling is introduced. They are at normal exit and at a **replace** in a handler. The actions that should be taken upon reaching a **replace** in a handler are those same actions that are to be performed when the normal end is reached. Therefore, finalization of a signaller should be invoked by an underlying mechanism supporting a general finalization facility, both upon reaching a normal exit and upon reaching a **replace**. This is just another example of the additional protection provided by using closed programming constructs instead of separating the opening and closing actions.

Thus, if the signaller is programmed properly and the language provides the proper finalization facilities, it is never necessary to force resumption, and resumption can therefore be optional on the part of the invoker.

Case 2. The signaller must not be resumed. If the signaller should not be resumed after signalling the exception, the exception should be signalled just before a logical end of the signaller (i.e., the **end**, a Zahn-type **exit** [29], or in a separate branch of a conditional). Thus even if the handler were to resume the signaller, nothing would remain to be done in the signaller's body. The signaller would just return to its invoker normally. Effectively, the signaller has not been resumed. Thus, here too, resumption can be left optional.

Case 3. Resumption is optional. In the replacement model resumption is always optional, and the decision of whether or not to resume is left to the invoker.

The mechanism introduces no additional resource requirements except when exception detection and handling are required. There are no new semantic mechanisms required for supporting exception handling at run time. The mechanisms already in place for dealing with procedures are the only ones used. The only additional resource requirement is the space in the activation records for handlers and for replace labels.

7.2 Replacement vs. Termination

Another issue is which of replacement or termination should be supported. This subsection casts the *sumstream* example of [12,14] in the replacement model in order to demonstrate the differences in capabilities between the termination and the replacement models. *Sumstream* is a procedure that reads in signed decimal integers from a character stream and returns their sum. Liskov and Snyder's example is reproduced in Figure 4.

The input stream is viewed as containing a sequence of number fields separated by spaces. Each number field must consist of a nonempty sequence of digits, optionally preceded by a single minus sign.

Sum_stream signals three exceptions under the following conditions: *overflow*—when the intermediate sum being accumulated is outside the implemented range of integers; *unrepresentable_integer(s)*—when the string *s* contains an individual number that is outside the implemented range of integers; and *bad_format(s)*—when the string *s* contains a field that is not an integer.

Sum_stream uses the procedure *get_number* to get individual numbers from the stream. *Get_number* signals *end_of_file* when it is invoked and there are no more numbers in the stream. This *get_number* makes use of the procedures *get_field* and *s2i*, neither shown here. If *get_field* is invoked when there are no more fields to get, it signals *end_of_file*. *S2i* converts its string argument to an integer if it is possible to do so. *S2i*'s exception conditions are the following: *invalid_character*—when the string contains other than a digit or a minus; *bad_format*—when the string contains a minus after a digit, more than one minus or no digits; and *unrepresentable_integer*—when the string contains an individual number that is outside the implemented range of integers.

Figure 5 contains a rendition of this example using the replacement model. The program structure in this example was chosen to be similar to that in [14] in order to facilitate comparison. It is assumed that *get field* is of type

```
proc(stream)string
   signals(exc(char,char)end of file).
```

S2i is assumed to have the type

```
proc(string)int
   signals(exc(char)(char,int)invalid character,
          exc(string)(int,int)bad format,
          exc(string)(int,int)unrepresentable integer).
```

The resume modes of *s2i*'s exceptions were chosen arbitrarily, since the context in which they are signalled is not provided.

As in the CLU example, the three exceptions of *get number* and the exception *overflow* of + are propagated to the invoker of *sumstream*. It is often the case that when an exception is signalled by one operation used within the implementation of another operation, and this exception is specifiable in terms of the implemented operation, propagating the exception yields the most useful implemented operation, since the decision of how to handle can be made at whatever level is most suitable for the application. In *sumstream*'s case, completing the handling of *get number*'s exceptions within *sumstream* does not lead to a flexible *sumstream* operation. The information of whether an exception should be considered as a fatal error, requiring termination of *sumstream*, or as a recoverable problem, is not available in *sumstream* itself. Therefore, the CLU *sum_stream* also propagates *get_number*'s exceptions.

```
sum_stream =proc (s:stream)returns (int)
        signals (overflow,
            unrepresentable_integer(string),
            bad_format(string))
    sum:int: =0
    while true do
        sum: =sum +get_number(s)
    end
    except
        when end_of_file:
            return (sum)
        when unrepresentable_integer(f:string):
            signal unrepresentable_integer(f)
        when bad_format(f:string):
            signal bad_format(f)
        when overflow:
            signal overflow
    end
end sum_stream

get_number =proc (s:stream)returns (int)
        signals (end_of_file,
            unrepresentable_integer(string),
            bad_format(string))
    field:string: =get_field(s)
    except
        when end_of_file:
            signal end_of_file
    end
    return (s2i(field))
    except
        when unrepresentable_integer:
            signal unrepresentable_integer(field)
        when bad_format, invalid_character(*):
            signal bad_format(field)
    end
end get_number
```

Figure 4

There is however a major difference between propagation in the termination model and propagation in the replacement model. In the termination model, transferring control to a handler causes immediate termination of the signaller. Thus, propagation causes immediate termination of *sum_stream.* There is no choice left to *sum_stream*'s invoker but to record the first bad string, and possibly reinvoke *sum_stream.* For *sum_stream,* reinvocation is feasible. In other cases, it may be infeasible or prohibitively expensive to restart an operation, since all the local state the operation has accumulated up to the time of detecting the exception is lost upon termination. In the replacement model, the invoker can choose between termination and resumption. A resuming handler could, for example, substitute zeros for strings that *get number* cannot translate to integers, so that *sumstream* would deliver the sum of all good strings only.

```
proc sumstream=(stream s)int
    signals(exc(int,int)overflow,
            exc(string)(int,int)unrepresentable integer,
            exc(string)(int,int)bad format):
begin
    int sum:=0;
    do
        sum:=sum+get number(s)
    od
    on
        end file=(char,char):
            begin sum replace end,
        unrepresentable integer=(string num)(void,int):
            begin unrepresentable integer(num) replace end,
        bad format=(string num)(void,int):
            begin bad format(num) replace end
        overflow=(int x,int y)(int,int):
            begin overflow replace end
    no
end

proc get number=(stream s)int
    signals(exc(char,char)end of file,
            exc(string)(void,int)unrepresentable integer,
            exc(string)(void,int)bad format):
begin
    s2i(get field(s))
end
on
    end of file=(char,char):
        begin end of file
replace end,
    unrepresentable integer=(string s)(int,int):
        begin unrepresentable integer(s) replace end,
    bad format=(string s)(int,int):
        begin bad format(s) replace end,
    invalid character=(char c)(char,int):
        begin string s:=c; bad format(s)replace end
no
```

Figure 5

Proponents of the termination model sometimes argue that resumption *can* be obtained using the termination model. This is only partially true. Resumption can be obtained only by local handling of exceptions *inside* an operation. However, this effect can also be obtained without an exception-handling mechanism. There is no way that resumption can be obtained in the termination model once an exception has been signalled by an operation to an external invoker, since the signalling is always followed by termination of the signaller. This is the only case for which an exception-handling mechanism is essential.

Resumption would be possible in the CLU *sum_stream* if

1. the body of the loop were a block, and

2. a handler for an exception of *get_number* were attached to this block *inside* the body of *sum_stream.*

An exception would then transfer control out of this block, and drop down to the end of the loop body to perform the next iteration. However, this version of *sum_stream* would *always* be resuming *sum_stream,* and termination could *not* be obtained at the option of the invoker's handler. The difference is a difference of modularity. In the termination model, all decisions about resumption or termination have to be made *inside* the signaller, and none are left to the invoker. The replacement model allows uncoupling the signaller and invoker in this decision, and exception handling can then be done at whatever level is most appropriate for the application.

Get number, which is invoked within *sumstream* and *sumstream* itself, signals exceptions with identical identifiers but different types. The static scope rules allow determining the intended exception and performing the necessary type checking, since a signalling identifies the declaration of the exception in the **signals** clause of the innermost enclosing signaller declaring an exception with the same identifier. If *sumstream* had not declared the exceptions *overflow, unrepresentable integer, bad format,* it could not propagate the exceptions of *get number* and +.

7.3 Resumption vs. No Resumption

There is still the issue of whether or not to support resumption at all. Supporting resumption is not particularly useful for language-defined operations such as arithmetic operations or array subscripting, which are best considered atomic. However, for the operations of user-defined abstract data types, which may be lengthy and complex, supporting resumption may be essential. The reason for this is that resumption allows handling an exception without losing important state information which may have been accumulated after much arduous computation up to the point at which the exception was detected. Termination, on the other hand, will require reinitialization of the terminated operation and some recomputation when the terminated operation is restarted. In addition, resumption may be necessary in order to avoid undoing side effects that had happened up to the point at which the exception was detected. In many cases, it is necessary to have started a computation before an exception can be feasibly detected, even when the exeption is due to the failure of an input condition to hold. For example, in a merge of two sorted files, the input condition is that the two files are sorted. It is *not* feasible to check for this before beginning to sort. The ability to resume means that the sortedness can be checked on the fly and when an out of order record is found, the handler can dump it to another file and resume the merging with the remaining records.[5]

7.4 Propagation Along Invocation and Use Hierarchies

[5] Having *merge* do the dumping is not the same. In that case, the signaller is deciding what to do for *all* invokers, rather than each invoker deciding for itself via its own designated handler.

Section 3 mentions Levin's alternative proposal for propagation of exceptions. Specifically, Levin's mechanism permits associating exceptions with objects as well as to invocations. This allows all users of an object to handle an exception rather than just the invoker of the call giving rise to the exception condition. Consider a memory management abstraction providing a *memory* object out of which to carve allocations. Among the abstraction's operations is an *allocate* routine. Of course, during any invocation of *allocate* the exception condition *out of memory* can occur. One possibility is to associate the exception with the routine *allocate* so that only the invoker *i* of the invocation of *allocate* that discovers the condition is notified. Another possibility is to associate the exception with the *memory* object so that all users of *memory* can be informed of the condition. The former is propagation of the exception along the *calls* chain while the latter is propagation of the exception along the *uses* chain.

When the exception is propagated along the use chain, all users of *memory* are required to be prepared to handle the *out of memory* exception by having an handler for it. This appears to create high coupling in the system, for totally unrelated processes end up being involved in each other's computations. However, this coupling is still protected, since each process acts through its own view of the *shared* abstraction. It can be argued that since these unrelated computations are sharing the *memory* object, they do have to be involved with each other; in fact, like it or not, they are involved with each other, because if one uses up the last available byte of memory, then none of the others can do any more allocation.

This method of exception propagation has its advantages and disadvantages. On the one hand, with this kind of propagation, any process that can do something about the lack of memory, e.g., release allocated memory or reorganize the free list, can respond. The result is that the process *i* would find its request satisfied. This solution offers more potential than only informing *i* of the exception; after all, if there is no more memory and *i* cannot continue without the piece it has requested, what can it do to create more memory? On the other hand, since the informed processes may be at arbitrary points in their respective programs when they are called upon to release memory, programming the required set of handlers may prove to be complex. In this particular example, it can be very difficult to write handlers that determine for each point of the program which of the visible allocated *memory* objects are dispensible.

This paper does not consider this alternative view of exception handling because we did not fully understand the power of this form of exception propagation and still feel that on balance, the coupling is unacceptably high. However, this alternative is orthogonal to that proposed in this paper. If properly done in the context of an expression-oriented, strongly-typed language with a modularizing construct, this alternative may add power to the language.

Still another alternative is to try to simulate the use of chain propagation of exceptions with the facilities that we already have, as was done by Liskov and Snyder [14]. For example, when declaring the *out of memory* exception inside the *memory* abstraction module, a default handler can be provided which sends a message to the same processes that would respond to the exception in the other method. If a user of the *memory* abstraction does not provide its own handler for *out of memory*, the default handler will be used. Of course, this method is far less convenient than with Levin's scheme, because more prearrangement of the globally visible channels of communication is needed.

8. CONCLUSIONS

Adopting an expression-oriented approach and generalizing the concept of a signaller enables us to support all the structured handler responses that were considered useful in various proposals for exception handling, with minimal additional mechanism. The uniformity of the mechanism contributes to the simplicity of its semantics. In contrast, the only other exception-handling proposals supporting both resumption and termination of the signaller, those of [7] and [17], require a much more complex syntactic and semantic extension, although neither one supports all the handler responses supported in the replacement model. The mechanism can be adapted to any of the block-structured programming languages modulo their specific restrictions, with little loss of expressive power. We are currently adapting the mechanism for parallel programming languages. The preliminary results are encouraging [28].

It is interesting to note that addressing exception handling in the context of modularity and program verification provides insights that contribute to simplifying the mechanism. Modularity requires that both the exception state and the resumption state be *consistent* or *possible* states in the sense of [20], otherwise they cannot be specified externally without compromising modular information hiding. This eliminates the problem of signallers that must be resumed in order to restore the state to a consistent state. Signallers which cannot or must not be resumed simply signal their exceptions just before their logical ends, after which there is nothing left to be done in the signaller even if resumption should be attempted. This eliminates the need for constructs such as the SIGNAL and NOTIFY in [7], SIGNAL and ERROR in Mesa [17], and the main argument of [11] for supporting only resumption.

ACKNOWLEDGMENTS

The authors thank the referees of the first and second drafts of this paper for their highly detailed comments.

REFERENCES

1. Berry, D.M., Kemmerer, R.A., von Staa, A., and Yemini, S. Toward modular verifiable exception handling. *J. Comput. Lang. 5* (1980), 77–101.
2. Cocco, N., and Dulli, S. A mechanism for exception handling and its verification rules. *J. Comput. Lang. 7* (1982), 89–102.
3. Cristian, F. Robust data types. *Acta Inf. 17* (1982), 365–397.
4. Cristian, F. Reasoning about programs with exceptions. In *Proceedings 13th International Symposium on Fault-Tolerant Computing*, (Milano, Italy, June 27–30, 1983).
5. Cristian, F. Correct and robust programs. *IEEE Trans. on Softw. Eng. SE–10*, 2 (Mar. 1984).
6. Dennis, J.B. Modularity. In *Advanced Course in Software Engineering*, F. Bauer, Ed., Springer-Verlag, Berlin, 1973.
7. Goodenough, J.B. Exception handling: issues and a proposed notation. *Commun. ACM 18*, 2 (Dec. 1975).

8. *IBM OS PL/I Checkout and Optimizing Compilers: Language Reference Manual*, SC33-0009-2, IBM Corp., 1973.
9. ICHBIAH, J., *et al* Rationale for the design of the Ada programming language. *ACM SIGPLAN Not. 14*, 6 (June 1979).
10. JOHNSTON, J.B. The contour model of block structured processes. In *Proceedings ACM Conference on Data Structures in Programming Languages. ACM SIGPLAN Not. 6*, 2 (Feb. 1971).
11. LEVIN, R. Program structures for exceptional condition handling. Ph.D. dissertation, Carnegie Mellon Univ., June 1977.
12. LISKOV, B.H., and SNYDER, A. Structured exception handling. Computation Structures Group Memo 155, MIT, Dec. 1977.
13. LISKOV, B.H., SNYDER A., ATKINSON, R., and SHAFFERT, C. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977).
14. LISKOV, B.H., and SNYDER, A. Exception handling in CLU. *IEEE Trans. Softw. Eng. SE−5*, 6 (Nov. 1979).
15. LUCKHAM, D.C., and POLAK, W. Ada exception handling: an axiomatic approach. *ACM Trans. Prog. Lang. Syst. 2*, 2 (Apr. 1980).
16. MACLAREN, M.D. Exception handling in PL/I. In *Proceedings ACM Conference on Language Design for Reliable Software*, Mar. 1977.
17. MITCHELL, J.G., MAYBURY, W., and SWEET, R. *MESA Language Manual.* Xerox Research Center, Palo Alto, Calif., Mar. 1979.
18. MYERS, G.J. *Reliable Software through Composite Design.* Petrocelli/Charter, New York, 1975.
19. PARNAS, D.L. A technique for the specification of software modules. *Commun. ACM 15*, 5 (May 1972).
20. PARNAS, D.L. Response to detected errors in well-structured programs. Computer Science Dept., Carnegie-Mellon Univ., 1972.
21. PARNAS, D.L. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15*, 12 (Dec. 1972).
22. RANDELL, B., and RUSSEL, L.J. *ALGOL 60 Implementation.* Academic Press, New York, 1964.
23. SCHWARTZ, R.L. An axiomatic treatment of Algol 68 routines, In *Proceedings 6th International Conference on Automata, Languages and Programming*, (Gratz Austria, July 1979).
24. SCHWARTZ, R.L. and BERRY, D.M. A semantic view of Algol 68. *J. of Comput. Lang. 4* (1979), 1−15.
25. VAN WIJNGAARDEN, A. *et al* Revised report on the algorithmic language Algol 68. *Acta Inf. 5* (1975).
26. YEMINI, S. The replacement model for modular verifiable exception handling. Ph. D. dissertation, Computer Science Dept., UCLA, 1980.
27. YEMINI, S. An axiomatic treatment of exception handling. In *Proceedings 9th Symposium on Principles of Programming Languages* (Jan. 1982).
28. YEMINI, S. Task termination and exception handling in parallel programs. IBM T.J. Watson Research Laboratory, preprint, 1983.
29. ZAHN, C.T. A control statement for top-down structured programming, In *Programming Symposium*, Lecture Notes in Computer Science 19, J.B. Robinet, Ed., Springer-Verlag, Berlin, 1974.

Received August 1982; revised September 1984; accepted November 1984.