Exception Handling in CLU

BARBARA H. LISKOV AND ALAN SNYDER

Abstract—For programs to be reliable and fault tolerant, each program module must be defined to behave reasonably under a wide variety of circumstances. An exception handling mechanism supports the construction of such modules. This paper describes an exception handling mechanism developed as part of the CLU programming language. The CLU mechanism is based on a simple model of exception handling that leads to well-structured programs. It is engineered for ease of use and enhanced program readability. This paper discusses the various models of exception handling, the syntax and semantics of the CLU mechanism, and methods of implementing the mechanism and integrating it in debugging and production environments.

Index Terms—Exception handling, exit mechanisms, procedural abstractions, programming languages, structured programming.

I. Introduction

RECENTLY, there has been considerable emphasis on the development of programming language features that enhance the verifiability of programs [5]. While it is desirable that the task of developing correct programs be simplified as much as possible, another important goal of program construction is that programs behave "reasonably" under a wide range of circumstances. Such programs have been variously termed as reliable, robust, or fault tolerant.

In a reliable program, each procedure must be designed to behave as generally as possible. Its specifications should require a well-defined response to all possible combinations of legal inputs (inputs satisfying the type constraints), even when lower level modules on which this procedure is depending fail. Of course, different responses will be appropriate in the different cases. Note that even if the software has been verified, the possibility of hardware failure implies that software modules may fail, as does the presence of resource constraints.

This paper describes a linguistic mechanism that supports the construction of reliable software. The mechanism, called an *exception handling mechanism*, facilitates communication of certain information among procedures at different levels. The mechanism supports the view that different responses are appropriate in different situations. We assume that for each procedure there is a set of circumstances in which it will terminate "normally"; in general, this happens when the input arguments satisfy certain constraints and the lower level modules (implemented in both hardware and software) on which the procedure depends are all working properly. In other circumstances, the procedure is unable to perform any action that would lead to normal termination, but instead must notify some other procedure (for example, the invoking

Manuscript received March 8, 1979; revised June 25, 1979. This work was supported in part by the Advance Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under Contract N00014-75-C-0661, and in part by the National Science Foundation under Grants DCR74-21892 and MCS 74-21892.

B. H. Liskov is with the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

A. Snyder is with the Hewlett-Packard Corporation, Palo Alto, CA 94304.

0098-5589/79/1100-0546$00.75

procedure) that an *exceptional condition* (or *exception*) has occurred.

For example, suppose *search* is a procedure that retrieves information associated with a given identifier in a symbol table. *Search* can return this information only if the identifier is present in the symbol table. The absence of the identifier constitutes an exceptional condition. Other exceptional conditions might also occur, for example, if the symbol table is implemented using a stack and the module implementing stacks is not working properly.

In referring to the condition as exceptions rather than errors we are following Goodenough [2]. The term "exception" is chosen because, unlike the term "error," it does not imply that anything is wrong; this connotation is appropriate because an event that is viewed as an error by one procedure may not be viewed that way by another. In fact, the term "exception" indicates that something unusual has occurred, and even this may be misleading: if the exception handling mechanism were efficient enough, exceptions might be used to convey information about normal and usual situations. For example, the *search* procedure might terminate normally only if the identifier were a local variable of the current block and use the exception handling mechanism to convey extra information about nonlocal variables.

Exception handling mechanisms have been largely ignored in programming languages. For a discussion of existing mechanisms, the reader is referred to [2] and [3]. In our opinion, the existing mechanisms are overly powerful and ill-structured. For example, in the on-condition mechanism of PL/I, on-units are associated with invocations dynamically rather than statically, and global variables must be used to communicate data between the procedure performing the signal and the on-unit. Goodenough [2] proposes a new mechanism that is more constrained and better structured. The mechanism presented in this paper is still more constrained. We also believe it to be more conducive to the development of well-structured programs.

The mechanism we describe facilitates communication of information that can be used to recover from faults such as erroneous data and failures of lower level modules. We do not discuss the methods, e.g., redundancy, that are used for fault detection and recovery. Mechanisms that are designed to facilitate fault detection and recovery, e.g., recovery blocks [8], are complementary to ours, as was noted in [7].

The mechanism we describe has been defined as part of the CLU programming language [4]. The mechanism is of general interest because it is constrained and simple. Its design was based on a tradeoff between simplicity and expressive power; major design goals were ease of use and program readability. The mechanism was designed for a sequential language (without coroutines or parallel processes). Otherwise, however, the mechanism is not dependent on CLU semantics, and could be incorporated in any procedure oriented language.

In the next section we discuss the main decisions that must be made in designing an exception handling mechanism and the exception handling models that result from these decisions; we also discuss our decisions and our reasons for making them. In Section III we describe the syntax and semantics of the CLU exception handling mechanism. In Section IV we discuss some methods of implementing the mechanism and also how the mechanism can enhance programmer effectiveness in a debugging and a production environment. In Section V, we discuss the expressive power of our mechanism and compare it with some other mechanisms of greater power. Finally, in Section VI we summarize and evaluate what we have done.

II. The Model

To discuss exception handling we must first introduce some terminology about programs. The term *procedure* will be used to mean program text, either in a higher level language or in machine language. A procedure implements a *procedural abstraction*, which is a mapping from a set of argument objects to a set of result objects, possibly modifying some of the argument objects. A procedure may be *invoked* (or called) by an *invocation*, which is textually part of some procedure; that procedure is referred to as the *caller*. Invocation results in *activation* of the invoked procedure. An activation may *signal* an exception; the invocation that caused the activation *raises* that exception. The program text intended to be executed when an exception is raised is called the *handler*.

Our model of exception handling involves the communication of information from the procedure activation that detects an exceptional condition (the *signaler*) to some other procedure activation that is prepared to handle an occurrence of that condition (the *catcher*). In designing this model, we faced two major questions: 1) which procedure activations may catch an exception signaled by a procedure activation and 2) does the signaler continue to exist after signaling. These two questions are independent and may be addressed separately.

A. Single Versus Multilevel Mechanisms

The obvious candidates[1] for handling an exception signaled by some procedure activation are the activations in existence at the time the signal occurs. We can rule out the signaler itself, as exceptions are, by definition, conditions that the signaling procedure is unable to handle. The remaining question is whether to allow activations other than the immediate caller of the signaler to handle the exception.

Our answer to this question is based on the hierarchical program design methodology that CLU is intended to support [4]. As was explained above, each procedure implements a mapping. The caller of a procedure invokes the procedure to have the mapping performed; the caller need know only what the mapping is, and not how the procedure implements the mapping. Thus, while it is appropriate for the caller to know about the exceptions signaled by the procedure (and these are part of the abstraction implemented by that procedure), the caller should know nothing about the exceptions signaled by procedures used in the implementation of the invoked procedure.

The above considerations lead us to allow only the immediate caller of a procedure to handle exceptions signaled

[1] Levin [3] proposes an additional set of candidates. We will discuss Levin's work in Section V.

by that procedure. Of course, the handler in the caller can itself signal an exception, but that exception will then be part of the caller's abstraction.

We believe that the decision to limit handling of exceptions to the immediate caller is necessary for any well-structured exception handling mechanism. To maintain intellectual manageability of software, program structures that support understanding and verification through local code examination are needed. In particular, to understand how a procedure is implemented, one should not have to examine implementations of any other procedures. An understanding of the mappings performed by invoked procedures is needed, but this understanding should be obtained by reading specifications of those procedures and not their code. This requirement implies that specifications must describe all exceptions arising from invoking a procedure, including information about exceptions arising from procedures called at a lower level if the mechanism does not limit the handling of these exceptions. The point is that all exceptions that may be raised by a procedure, whether explicitly or implicitly, must be considered part of that procedural abstraction. Limiting the handling of exceptions to just the caller simply ensures that the linguistic constructs match the proper conceptual view. Note, however, that this constraint does *not* prevent the language designer from providing simplified ways of passing exceptions from one level to the next where appropriate.

The exception handling mechanism proposed by Goodenough [2] does impose our constraint on handling exceptions. The PL/I mechanism does not, nor does the mechanism in Mesa [6].

B. Resumption Versus Termination Model

The second question, whether the signaler should continue to exist after the exception is signaled, involves a tradeoff between expressive power and the complexity of the semantics. If the signaler can continue to exist after signaling, then it is possible that a catcher may fix up the exceptional condition so that processing of the signaler may be resumed. For this reason, we refer to this model as the *resumption model*. The model in which the signaling activation ceases to exist we refer to as the *termination model*. In this section we assume that the decision to support a one-level mechanism has been made, and we therefore limit our analysis to this case.

A one-level resumption model works as follows. Suppose that there are three procedures P, Q, and R, and that P invokes Q and Q invokes R. If R signals an exception r, then Q must handle it. Let H_r denote the statements in Q that handle r (H_r is the handler for r). In the course of handling r, H_r may signal an exception q, which must be handled by P (since P is the caller of Q).[2] Let H_q denote the statements in P that handle q (H_q is the handler for q). When H_q terminates, then

[2]If dynamic binding for exception names is used (as in PL/I), then R would be required to handle q. Making this assumption leads to a model at least as complex as the one we are considering. Furthermore, it is impossible under this assumption for H_r to raise exception in P without resorting to a multilevel mechanism.

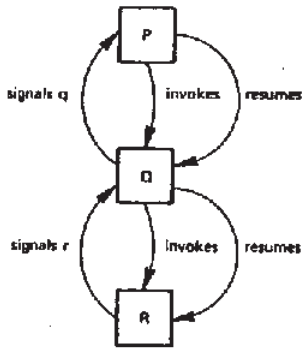

Fig. 1. Flow of control in the resumption model.

Q is resumed in the middle of H_r; only when H_r terminates is the execution of R continued. This situation is illustrated in Fig. 1. Note that information about signals flows upward one level at a time, while resumption flows downward one level at a time; multilevel flow is not permitted in either direction.

The resumption model is most easily understood by viewing the handler as an implicit procedure parameter of the signaler. The handler is called by the signaler when the exception it handles is signaled. The handler procedure is declared in the calling procedure, and its free variables get their meaning in the caller's environment, as do any exceptions it signals.[3]

In the termination model, occurrence of an exception causes the signaler to terminate. However, different kinds of behavior are expected of the called procedure under different conditions. The view taken is that a procedure may terminate in one of a number of conditions. One of these is the *normal condition*, while others are *exception conditions*. In each condition, it may be convenient to return a number of result objects; these will differ in number and type in the different conditions.

The resumption model is more complex than the termination model. This can be appreciated by considering how resumption affects the interrelationships among procedures, specifications of procedures, and linguistic mechanisms for exception handling.

The ordinary view of procedures is that, in the absence of recursion, the calling procedure is dependent on the called procedure but not vice versa. This view is upheld in the termination model. However, in the resumption model, the signaler and caller are mutually dependent: the caller invokes the signaler to perform some mapping, or satisfy some input/output relation, but the signaler depends on (the handler in) the caller to satisfy a similar relation when an exception is signaled.

Specifications of procedural abstractions in the termination model consist of a number of clauses, one specifying the behavior for the normal case and one for each exception case. Such clauses also exist in the resumption model, since it is still possible that the signaler is unable to terminate normally, for example, because a handler is unable to clear up the prob-

[3]That is, exception names have static scope.

lem that led to the exception. In general, specifications have a termination model form (several termination states are defined) even when the resumption model is in use.

The interdependence between procedures in the resumption model show up in specifications as extra information. In addition to the clauses describing different termination states, it is also necessary to include descriptions of the behavior expected from the handlers when exceptions are signaled. Such descriptions are analogous to what must be given for a procedure taking procedure parameters, since handlers are implicit procedure parameters, as was discussed earlier.

The complexity of a linguistic mechanism supporting resumption is illustrated by Goodenough's proposal [2], which is a carefully considered design of a complete mechanism. Goodenough's design recognizes that to be really useful, termination must be supported as well as resumption. Three types of signals are recognized, corresponding to cases where the signaler may not be resumed, must be resumed, or where resumption is optional. In case the caller does not resume a signaler that must or could be resumed, a special ability is provided to permit the signaler to clean up (i.e., restore some nonlocal variables to a consistent state) before its activation is terminated. In addition, a default mechanism is provided to permit the signaler to handle its own exception in case the caller does not.

The termination model requires a simpler linguistic mechanism for its support than does the resumption model. Since a signal terminates the signaler, there is no need for multiple kinds of signals. Also, special mechanisms for cleaning up are not needed (the signaler must always clean up before signaling).

Since the termination model is simpler, it is preferable to the resumption model, provided it supplies adequate expressive power. We conjecture that the expressive power is adequate: that situations handled awkwardly by the termination model and simply by the resumption model are not frequent. We will discuss this conjecture further in Section V. In the next section we discuss the design of an exception handling mechanism based on the termination model.

III. Syntax and Semantics of the CLU Exception Mechanism

In Section II we explained the rationale for our major decisions.

1) The exceptions signaled by a procedure must be caught by the immediate caller.

2) Signaling an exception terminates the signaling procedure.

These two decisions lead to a single-level termination model of computation in which a procedure may terminate in one of a number of conditions. Thus, instead of a single return path, each procedure has several return paths. One of these is considered the normal path, while others are considered exceptional. In each case, result objects may be returned; the result objects may differ in number and type in the different cases.

An exception handling semantics that terminates execution of the signaling procedure could be incorporated in a programming language with no additional mechanism. The signaling procedure could simply return, passing back in addition to the real result objects a tag that identifies the reason for termination. Indeed, such a convention is often adopted as a way of dealing with exceptions in a language that has no exception handling mechanism. However, this approach has a major defect: every invocation must be followed by a conditional test to determine what the outcome was. This requirement leads to programs that are difficult to read, and probably inefficient as well, thus discouraging programmers from signaling and handling exceptions.

To aid programmers in building reliable software, an exception handling mechanism must be devised that can be implemented efficiently and that enhances program readability. In the remainder of this section we describe the CLU exception handling mechanism, which was developed to satisfy these goals. The discussion identifies some problems that arise in designing any such mechanism; the CLU mechanism provides a possible set of solutions to these problems.

A. Signaling

To provide a convenient method of signaling information about exceptions, we included directly in CLU the model of a procedure having many kinds of returns. A CLU procedure, therefore, can terminate in the normal way by returning and can terminate in an exceptional condition by signaling. In each case, result objects, differing in number and type, can be returned.

The information about the ways in which a procedure may terminate must be included in its heading. For example, the procedure performing integer division has the following heading:

```
div = proc (x, y: int) returns (int) signals (zero_divide)
```

which indicates that *div* may terminate by returning a single integer (the quotient of the two input arguments) or by signaling *zero_divide* (which indicates that the second argument was zero) and returning no results.

A CLU procedure terminates its execution by performing a *return statement* or a *signal statement*. The return statement terminates execution normally, while the signal statement terminates execution in the named exceptional condition. For example, the following (fairly useless) procedure determines the sign of an integer:

```
sign = proc (x: int) returns (int) signals (zero, neg (int))
       if x < 0 then signal neg (x)
          elseif x = 0 then signal zero
          else return (x)
          end
       end sign
```

The information in the procedure heading is used to check that the exception names actually signaled are the correct ones and that the correct number and types of result objects are returned in both the normal and exceptional cases. This information is also used to determine that the exceptions handled by a calling procedure are named in the heading of the called procedure, and that, again, the number and types of result objects are correct in both the normal and exceptional cases.

B. Handling Exceptions

In CLU, exceptions arise only from invocations.[4] In particular, all uses of infix and prefix operators in CLU are considered to be "syntactic sugar" for invocations. For example, the expression

```
x + y
```

is syntactic sugar for the invocation

```
t$add(x, y)
```

where t is the type of x. Thus, if x is an integer, $x + y$ is an invocation of the integer addition operation. This viewpoint permits exceptions arising from built-in operations and user-defined procedures to be treated uniformly.[5]

In this section we discuss how handlers are associated with invocations. For usability and program readability, it is necessary to permit considerable flexibility in the placement of handlers. For example, requiring that the text of a handler be attached to the invocation that raises the exception would lead to unreadable programs in which expressions were broken up with handlers. Furthermore, the control flow of a program is often affected by the occurrence of an exception (for example, an *end_of_file* exception will terminate a loop). Therefore, our mechanism was designed to permit placement of a handler where the programmer deemed convenient, out of the main flow when possible to enhance readability, and altering the control flow when this was desired.

Two major decisions determined the form of CLU exception handling statements.

1) Handlers are statically associated with invocations.

2) Handlers may be attached only to statements, not to expressions.

Static association means that the handler associated with a particular exception condition that may be raised by a particular invocation can be determined by static analysis of the program text. This decision not only enhances program readability, but makes possible a more efficient implementation of the exception handling mechanism.

The decision to attach handlers only to statements and not expressions was made to simplify the mechanism. When a handler attached to an expression terminates, unless an explicit return, signal, or exit (see Section III-C) is performed, it must provide a value to be used as the value of the expression. By allowing handlers to be attached only to statements, we avoid providing a mechanism for substituting new values for expressions. We believe that the need to substitute a value for an expression is not great. In any case, the effect of attaching handlers to expressions can be obtained by breaking up complex expressions into sequences of assignment statements.

Handlers are placed in CLU programs by means of the *except statement*, which has the form

statement **except** *handler list* **end**

This statement has the following interpretation: the *statement* raises all the exceptions raised by the invocations it textually contains, excluding those handled by embedded except statements. The *handler list* will handle some subset (possibly all) of these exceptions. The except statement as a whole raises all the exceptions of the *statement* that are not handled by the *handler list* plus any exceptions raised by the *handler list*. Thus, when an exception is raised by an invocation, control goes to the innermost handler that handles that exception and is part of an except statement containing the invocation in its *statement* part.

Each handler in the *handler list* names one or more exceptions to be handled, followed by a list of statements (called the *handler body*) describing what to do. Permitting several exceptions to be named in the same handler avoids code duplication when the exceptions are all handled in the same way.

Several different forms are available for handlers depending on whether the named exceptions have associated result objects and whether those objects are used in the handler body. To handle one or more exceptions with no associated objects, the exception names are simply listed. For example,

when underflow, zero_divide: *body*

will handle exceptions named *underflow* and *zero_divide*, neither of which has any associated result objects.

To handle exceptions with result objects that are to be used in the handler body, names must be associated with the objects. Again a list of exception names is given, but it is followed by declarations of local variables to name the result objects, for example,

when e1, e2 (s: string, i: int): *body*

The scope of the declarations is the handler body. All of the named exceptions must return objects of the types listed in the declaration, in the order stated. When the handler is executed, these objects are bound to the declared variables and the body is executed. (This binding is similar to the binding of actual arguments to formal arguments that occurs when procedures are invoked. However, a return or signal in the handler body, rather than terminating just the handler, will instead terminate the entire enclosing procedure.)

To handle exceptions with result objects when the objects are not used in the handler body, the list of exception names is followed by (*) as shown below:

when neg, underflow (*): *body*

There need be no agreement between the number and types of result objects associated with the exceptions in this form; for example, the *neg* exception had a single argument, while *underflow* had none. This form encourages a programming style in which a procedure returns all possibly useful information when signaling; if this information is not needed in the calling procedure, it can easily be ignored.

If the programmer wishes to handle all remaining exceptions without listing their names, one of the following two forms

[4] Except for the special exception *failure* (described in Section III-D), which may be signaled at any point by the underlying implementation of CLU.

[5] The viewpoint does *not* require that a built-in operation be implemented by a closed routine; in-line code is perfectly permissible and consistent.

can be used as the last handler in an except statement. The form

others: *body*

is used when information about exception names and result objects is not important. If information about the exception name is desired, the form

others (e_name: string): *body*

may be used. Here the name of the exception is given to the handler body as a string.

The handler body may contain any legal CLU statement. If the handler body returns or signals, then the containing procedure will be terminated as discussed in Section III-A. The handler body may also be terminated by an exit (see next section) or because an invocation within it raises an exception that is not handled within the handler body. Otherwise, when the handler body is finished, the next statement following the except statement in the normal flow will be executed.

The example below illustrates the association of handlers with exceptions:

```
begin % start of inner block
   S1 except
         when zero: S2
         end
   . . .
   end % end of inner block
   except
        when zero: S3
        others: S4
        end
```

If *zero* is raised by an invocation in *S1*, it will be handled by *S2*, not *S3*. However, if *zero* is raised by an invocation in *S2*, it will be handled by *S3*. All other exceptions raised in *S1* and *S2* will be handled by *S4*.

C. Exits and the Placement of Handlers

Our intention in defining the except statement is to permit the programmer to position handlers as is convenient. There are two constraints on the placement of handlers.

1) The handler must be placed on the statement whose execution is to be terminated if the handler body terminates without returning or signaling.

2) Suppose that an exception named *e* is raised by two invocations, and we wish to handle the occurrences of *e* differently. We do not permit multiple handlers to be provided for *e* in a single except statement. (This rule holds even if the invocations raising *e* provide different numbers or types of result objects; we do not allow such information to be used in selecting a handler.) Therefore, the two handlers must be in two except statements, each situated such that only one of the invocations raising *e* is in its scope.

These two constraints may conflict. For example, suppose that within a statement, *S*, the procedure *sign*, mentioned earlier, is invoked at two different points. Suppose also that the programmer wishes to handle the *neg* exception signaled by *sign* in a different manner for each of the two invocations, but in each case wishes execution to then continue with the statement following *S*. The first constraint would require that both handlers be placed on *S*, so that the execution of *S* would be terminated when the exceptions are raised. However, the second constraint requires that at least one handler be placed within *S* to resolve the ambiguous association between the invocations and the handlers.

```
begin  % beginning of S
 a := sign(x)
  except when neg(i: int):
           S1
           exit done
  end
 b := sign(y)
  except when neg(i: int):
           S2
           exit done
  end
 -
end  % end of S
  except when done:
   -
   end
```

Fig. 2. Example illustrating use of the exit mechanism.

We resolve this conflict in CLU by the addition of an exit mechanism, similar to those proposed by Zahn [9] and Bochmann [1]. The handlers are placed near the invocations. They terminate by exiting to a handler attached to the statement *S*. For example, one could handle the *neg* exceptions as shown in Fig. 2.

The exit statement can be used anywhere within a CLU procedure; its use is not restricted to handler bodies. The exit statement is similar to the signal statement, except that while the signal statement signals the condition to the calling procedure activation, the exit statement directly raises the condition so that it can be handled in the same procedure activation. The exit statement can specify a number of result objects to be passed to the handler.

We chose to have separate mechanisms for exits and exceptions (rather than using the signal statement for both exits and exceptions) because the two mechanisms capture different programmer intentions and thus naturally have different restrictions on their use. The intent of an exit is a local transfer of control. Thus, we require that exits be handled in the same procedure activation where they are raised. Furthermore, we require that exits be handled by a when arm (not an others arm), and if there are result objects, these must be accepted as arguments by the handler. The justification for these requirements is that exit names and result objects (unlike exception names and result objects) are under the control of the programmer of the procedure, and therefore should be chosen to mean something within that procedure.

The exit mechanism meshes nicely with the exception handling mechanism. In fact, the signal statement can be viewed simply as terminating a procedure invocation and exiting to the appropriate handler in the caller.

D. Uncaught Exceptions

Now we address the question of what happens if a procedure provides no handler for an exception raised by some contained invocation. One possibility is to consider the procedure to be illegal; checking for unhandled exceptions can be performed at compile-time. This approach is taken by Goodenough [2].

We have taken another approach. We felt it was unrealistic to require the programmer to provide handlers in situations where no meaningful action can be taken. Such situations will occur when a used abstraction is not working properly. For example, consider the statement

```
if ~stack$empty(s) then
    ...
    x := stack$pop(s)
    ...
    end
```

Here the programmer invokes the *pop* operation for stacks only when the stack is nonempty. Now suppose that nevertheless stack underflow occurs. This situation is unlikely to arise in a debugged or verified program (but see Section IV). If it does arise, it indicates that the stack abstraction is not behaving correctly. Often there is no appropriate action for this procedure to take other than to report the fact to its caller. Since almost every abstraction can potentially behave incorrectly or in a way not expected by its caller, procedures must always be prepared to handle such cases. However, the action taken is almost always the same, and to require explicit handling of such cases would load every procedure with uninteresting code.

To facilitate reporting of failures and to relieve the programmer of the burden of handling such errors, CLU has one language-defined exception, named *failure*. *Failure* has one associated result object, a string that may contain some information about the cause of the failure. Every procedure can potentially signal *failure*; therefore *failure* is implicitly an exception of every procedure and may not be listed in the procedure heading explicitly. *Failure* may be signaled explicitly, however, in the usual way:

```
signal failure ("reason is . . . ")
```

The most common way that *failure* is signaled, however, is by an uncaught exception being automatically turned into a *failure* exception. For example, procedure *nonzero*

```
nonzero = proc (x: int) returns (int)
          return (sign (x))
              except
                  when neg (y: int): return (y)
                  end
          end nonzero
```

does not catch exception *zero* signaled by *sign*. If this exception is signaled, the invocation of *nonzero* will be terminated with the exception

```
failure ("unhandled exception: zero")
```

The effect is equivalent to attaching a handler to the procedure body, e.g.,

```
nonzero = . . .
          . . .
          except
              others(s: string): signal failure (
                      "unhandled exception: "|| s)
              end
          end nonzero
```

Here the symbol || is string concatenation.

A common case in which an exception will not be handled is when the unhandled exception is *failure*. Note that in this case it is the string argument of *failure* (rather than the string "failure") that is of interest. Therefore, this string is retained when *failure* is passed up to the next level. This effect is equivalent to attaching to the procedure body the handler

```
except
    when failure (s: string): signal failure (s)
    end
```

Sometimes before signaling *failure* some cleaning up is needed. In this case, the others or when form is used explicitly, and after cleaning up, *failure* is signaled explicitly.

E. Example

We now present an example demonstrating the use of exception handlers. We will write a procedure, *sum_stream*, which reads in a sequence of signed decimal integers from a character stream and returns the sum of those integers. The input stream is viewed as containing a sequence of fields separated by spaces and newlines; each field must consist of a nonempty sequence of digits, optionally preceded by a single minus sign. *Sum_stream* has the form

```
sum_stream = proc (s: stream) returns (int)
             signals (overflow,
                      unrepresentable_integer (string),
                      bad_format (string))
    . . .
    end sum_stream
```

Sum_stream will signal *overflow* if the sum of the numbers or an intermediate sum is outside the implemented range of integers. *Unrepresentable_integer* will be signaled if the stream contains an individual number that is outside the implemented range of integers. *Bad_format* will be signaled if the stream contains a field that is not an integer.

An implementation of *sum_stream* is presented in Fig. 3. It consists of a simple loop that accumulates the sum, using a procedure *get_number* to remove the next integer from the stream. *Get_number* will signal *end_of_file* if the stream contains no more fields, in which case *sum_stream* will return the accumulated sum. *Get_number* will also signal *bad_format* or *unrepresentable_integer* if an invalid field is encountered; these exceptions are passed upward by *sum_stream*. The *overflow* handler in *sum_stream* catches exceptions signaled by the *int$add* procedure, which is invoked using the infix + notation. We have placed the exception handlers on

```
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
```

Fig. 3. The sum_stream procedure.

```
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
```

Fig. 4. The get_number procedure.

the while statement for readability; they could also have been placed directly on the assignment statement.

The procedure *get_number* is presented in Fig. 4. It calls a procedure *get_field* to obtain the next field in the stream and then uses *s2i* to convert the returned string to an integer. *S2i* has the following form:

```
s2i = proc (s: string) returns (int)
        signals (invalid_character (char),
                bad_format,
                unrepresentable_integer)
        ...
        end s2i
```

S2i will signal *invalid_character* if the string *s* contains a character other than a digit or a minus sign. *Bad_format* will be signaled if *s* contains a minus sign following a digit, more than one minus sign, or no digits. *Unrepresentable_integer* will be signaled if *s* represents an integer that is outside the implemented range of integers. *Get_number* handles the exceptions signaled by *get_field* and *s2i* and signals them upward in terms that are meaningful to its callers. Although some of the names may be unchanged, the meanings of the exceptions (and even the number of arguments) are different in the two levels. Note the use of the (*) form in the handler for the *bad_format* and *invalid_character* exceptions since the signal arguments are not used.

The *get_field* procedure is presented in Fig. 5. It uses the following operation of the *stream* data type:

```
getc = proc (s: stream) returns (char) signals (end_of_file)
        ...
        end getc
```

The *stream$getc* operation returns the next character from the stream and signals *end_of_file* if the stream is empty. Note that if *end_of_file* is signaled when a field is being accumulated, then that field is returned. Otherwise, *get_field* signals *end_of_file*.

Programming of the procedures in Figs. 3-5 would be

```
get_field = proc (s: stream) returns (string) signals (end_of_file)
        field: string := ""
        begin  % delimits scope of outermost end_of_file handler
               c: char := stream$getc (s)
               % search for field
               while c = ' ' cor c = '\n' do
                       c := stream$getc (s)
                       end
               % accumulate field
               while c ~= ' ' cand c ~= '\n' do
                       field := string$append (field, c)
                       c := stream$getc (s)
                          except when end_of_file:
                                return (field)
                                end
                       end
               end
                  except when end_of_file:
                       signal end_of_file
                      end
        return (field)
        end get_field
```

Fig. 5. The get_field procedure.

simplified if the mechanism permitted implicit upward propagation of exceptions. This would permit arms of the form

```
when unrepresentable_integer (f: string):
        signal unrepresentable_integer (f)
```

to be omitted from the program text. As we gain experience in using the mechanism, we will learn how to modify it to enhance its convenience.

F. On Disabling Exceptions

One question that naturally arises about an exception handling mechanism is whether exceptions can be disabled. By disabling exceptions two kinds of savings can (potentially) be realized: the time spent detecting the occurrence of the exception can be saved, and the space used for the handlers and the information used to find the handlers can be saved. However, it is unacceptable if the result of disabling exceptions is that errors still occur, but are simply not recognized. Therefore, we do not believe that providing a means for programmer disabling of exceptions is consistent with encouraging good programming practice, and no such mechanism has been provided in CLU.

The situation still arises, however, in which it is possible to *guarantee* that the exception cannot occur, and it is desirable to take advantage of that guarantee to generate more efficient code. Looked at in this way, disabling of exceptions is seen as a kind of program optimization technique, since program optimization makes use of properties detected from program analysis to control the generation of code. There are two ways in which such properties can be detected. First, the combination of in-line substitution followed by analysis across module boundaries can result in more efficient code. For example, consider

```
if ~stack$empty(s) then x := stack$pop(s) · · ·
```

where s is a *stack*. If both *empty* and *pop* are expanded in-line, the result will be code roughly like

```
if s.size > 0 % body of empty
   then % body of pop
      if s.size > 0 then · · ·
```

Conventional techniques like redundant expression elimination and dead code removal can then be used to improve the code.

Alternatively, it would be fruitful to integrate the activities of a program verification system with the compiler. Then, for example, a verifier might prove of the user of s that *pop* is never called if s is empty. This assertion could then be used later to control the compilation of both the program using s, and the program implementing the *stack* module.

IV. Implementation, Debugging, and Diagnostics

In this section we discuss some implementation issues. First we sketch some methods for implementing the exception handling mechanism. Then we discuss how the mechanism can be incorporated in a debugging environment and in a production environment.

A. Implementation

There are several possible methods of implementing the exception handling mechanism. As usual, tradeoffs must be made between efficiency of space and time. We believe the following are appropriate criteria for an implementation:

1) normal case execution efficiency should not be impaired at all;

2) exceptions should be handled reasonably quickly, but not necessarily as fast as possible;

3) use of space should be reasonably efficient.

The tradeoff to be made is the speed with which exceptions are handled versus the space required for code or data used to locate handlers.

The implementation of signaling an exception involves the following actions:

1) discarding the activation record of the signaling activation (but saving the result objects associated with the exception),

2) locating the appropriate handler in the calling procedure,

3) adjusting the caller's activation record to reflect the possible termination of execution of expressions and statements,

4) copying the result objects into the caller's activation record,

5) transferring control to the handler.

Actions 3) and 5) are equivalent to a **goto** from the invocation to the handler. Actions 1) and 4) are similar to those occurring in normal procedure returns. Because the association between invocations and handlers is static, the compiler can provide the information needed to perform actions 2) and 3). Below we sketch two methods of providing this information; these methods differ considerably in their performance characteristics.

The first method, called the *branch table method*, is to follow each invocation with a branch table containing one entry for each exception that can be raised by the invocation. The

```
Code for invocation p( ) of p = proc ( ) returns ( ) signals (e1, e2):

                call   p
                e1_handler          ; branch table
                e2_handler
                failure_handler
                ...                 ; normal return here
-----------------------------------------------------------
                size1               ; new activation record size
                ...                 ; other information about the handler
e1_handler:     ...                 ; code for e1 handler
```

Fig. 6. Sketch of code generated by the branch table method.

invocation of a procedure whose heading lists n exceptions will have a branch table of $n + 1$ entries; the first n entries correspond to the exceptions listed in the heading, while the last entry is for *failure*. Each entry contains the location of a handler for the corresponding exception.

Using this method, return and signal are easy to implement: return transfers control to the location following the branch table, while signal transfers control to the location stored in the branch table entry for the exception being signaled. The information needed to adjust the caller's activation record could be stored with the handler, as could information about whether to discard the returned objects and whether this is an others handler; for example, this information could be stored in a table placed just before the first instruction of the handler. An example is given in Fig. 6 of the code generated by this method.

The branch table method provides for efficient signaling of exceptions, but at a considerable cost in space, since every invocation must be followed by a branch table (all invocations may at least signal *failure*). A second method, the *handler table method*, is the one used by the current CLU implementation. This method trades off some speed for space, and was designed under the assumption that there are many fewer handlers than invocations, which is consistent with our experience in using the mechanism.

The handler table method works as follows. Rather than build a branch table per invocation, the compiler builds a single table for each procedure. This table contains an entry for each handler in the procedure. An entry contains the following information: 1) a list of the exceptions handled by the handler (a null list can be used to indicate an others handler), 2) a pair of values defining the scope of the handler, that is, the object code corresponding to the statement to which the handler is attached, 3) the location of the code of the handler, 4) the new activation record size, and 5) an indicator of whether the returned objects are used in the handler. The scope and exceptions list together permit candidate handlers to be located: only an invocation occurring within the scope and raising an exception named in the exception list can possibly be handled by the handler (for an others handler, only the scope matters).

In this method, a return statement is implemented just as it would be in a language without exception handling. A signal statement requires searching the handler table to find entries for candidate handlers; if several candidates exist, the one with the smallest scope is selected. Placing the entries in the table in the (linear) order in which the corresponding handlers appear in the source text guarantees that the first candidate found is the handler to use. Unhandled exceptions can be recognized either by the absence of candidates or by storing one additional entry at the end of the handler table for this case.

B. Debugging and Diagnostics

Our exception handling mechanism is designed explicitly to provide information that programs, not programmers, can use to recover from exceptional conditions. However, the mechanism can also mesh smoothly with mechanisms intended to collect information of interest to programmers. The kind of behavior desired will differ, however, from a debugging environment to a production environment.

In an interactive debugging environment it is likely that a programmer would wish to be informed about the occurrence of some or all exceptions as they are signaled and be given a chance to handle them himself or take some other corrective action. Two possible modes might be useful here. The programmer may be interested only in signals of *failure* (especially those resulting from unhandled exceptions), or he may in addition name some particular exceptions of interest.

An exception handling mechanism running in such an environment, before locating a handler, would consult some debugging system information to determine if the current exception is one that the programmer wishes to know about. If the exception is of interest to the programmer, then system routines can be invoked to initiate a dialogue with the programmer. This dialogue may result in the program being continued or terminated.

It is worth noting that one argument in favor of the resumption model has been that it integrates debugging with program execution. The programmer (or actually the system as his representative) is thought of as the highest level activation, which will handle all exceptions not otherwise handled and which may later resume execution of some lower level activation. Note that this viewpoint allows the programmer to examine only unhandled exceptions. At any rate, we believe that it is not productive to try to merge debugging with ordinary processing, since the requirements in the two cases are quite different.

In a production environment, there is no programmer

available to interact with the program. Of course, there may be an operator present, and a program may attempt to recover by requesting some operator action (e.g., mounting a tape). This action can be accomplished by ordinary program structures (e.g., invoking a procedure to print a message on the operator's console).

When *failure* occurs in a production environment, there is still a good chance that program error is responsible. Therefore, it would be helpful if information about the failing program were collected for later examination by a programmer. This capability can easily be provided. Whenever *failure* is signaled, the exception handling mechanism can output information about each activation before terminating it. In the case of the first implicit signal of the "unhandled exception" failure, the mechanism should also provide information about the activation that signaled the unhandled exception. The information collected as *failure* propagates upwards will provide a trace of the failing program, which should be helpful for the programmer who determines later what the problem was. Debugging in a batch environment can be facilitated similarly, except that information about more exceptions than just *failure* may be of interest. Note that in either case the information being collected is *not* useful to programs (since it describes the states of implementations of other procedures) and therefore need not be made available to them.

V. Expressive Power

As we stated earlier, the decision to choose a termination model instead of a resumption model involves a tradeoff between the expressive power of the exception handling mechanism and its complexity. In our opinion, a more complex mechanism can be justified only if the additional expressive power it provides is frequently needed. In this section we explore this issue by considering examples of problems often put forth as justifying a resumption model.

The first problem concerns exceptions such as *underflow* that are generated by numeric operations. Often when an operation like *multiply* signals *underflow*, the desired action is to substitute a particular value (e.g., zero) for the result of the operation and continue the computation. In a resumption model, this behavior can be obtained by resuming the operation and passing it the value to be returned.

This behavior is equally easily obtained using a termination model. Because the *multiply* operation is not performing any computation after being resumed (it is merely returning the value provided), it is acceptable to terminate its activation. The only problem is for the handler to somehow substitute the new value for the result of the operation. For simple examples like

```
z := x * y
```

"substituting" for the result of the invocation of *multiply* can be done simply by assigning to z. For more complicated examples, e.g.,

```
z := x * y + z
```

using our mechanism it is necessary to introduce additional statements and temporary variables. However, such awkwardness is not a defect of the termination model but rather a result of our decision not to allow handlers to be attached to expressions. If such examples turned out to be frequent, our mechanism could be changed to accommodate them.

In fact, resumption is truly useful only in the following situation: when the exception is signaled, the signaler is in the middle of a computation that can be completed by performing additional computation upon receipt of a value from the handler. Resumption permits completion of the computation in this situation without redoing work already performed.

We can imagine that such a situation could arise during a numeric computation. If it did, and resumption were not available, then a default value (or, in the most general case, a procedure to compute a default value) could be passed as an extra input of the numeric routine.

This method is clearly not as convenient as using resumption; it becomes unacceptable if there are many default values or if there is deep nesting of procedures within the numeric routine, so that even a single default value must be passed down through many invocations. In our experience, neither of these characteristics hold for the routines in numeric libraries; on the contrary, default values are almost never of use, and the nesting is shallow.

The other example often used to support the choice of a resumption model is that of a storage pool that performs storage allocation for a number of objects in a program. If the amount of free storage in the storage pool becomes too low to satisfy a particular allocation request, it may still be possible to satisfy the request if some of the objects stored in the pool can be reorganized to use less storage. Many objects can be implemented in a number of ways, some that permit fast execution but use a lot of space and others that are slower but use less space. The idea would be to start out using fast representations but switch to more compact representations if free storage became too low. Note that this example is an instance of the general situation, described above, in which resumption is truly useful.

Levin [3] has designed an exception handling mechanism that directly supports the desired behavior. In Levin's mechanism, an exception can be associated with an object (the mechanisms discussed previously associate exceptions only with invocations). Thus, if the storage pool were unable to satisfy a request, it could signal an exception associated with the storage pool object. The mechanism would then allow all users of the object (in this case, modules that have objects allocated in the storage pool) to handle the exception. The handlers would attempt to free storage by reorganizing their associated objects.

Note that Levin's mechanism is strictly more powerful (in terms of expressive power) than the resumption models we discussed in Section II, since the users of the storage pool do not necessarily have any outstanding procedure activations at the time the exception is signaled. Furthermore, those objects that are in the middle of being operated upon are likely to be in an inconsistent state and thus not prepared for reorganization. Levin's mechanism makes it easy to inhibit the handling of an exception for objects in an inconsistent state.

In CLU, this recovery algorithm could be programmed by

having the storage pool explicitly maintain a collection of handler procedures to be invoked whenever free storage became too low.[6] The storage pool abstraction provides operations *alloc*, to add an object to a pool, and *delete*, to remove an object from a pool. *Alloc* would have an additional argument: the handler procedure to invoke if it becomes necessary to shrink the object being added to the pool. *Alloc* would add this procedure to the collection, while *delete* would remove from the collection the handler procedure associated with the object being deleted from the pool.

There is no doubt that the method sketched above is more complicated and more error prone than what could be done using Levin's mechanism. However, we believe that the storage pool example is both unusual and a special case. We doubt the existence of a large number of cases where the amount of storage freed would make the difference between successful and unsuccessful execution of a program.

In selecting examples for discussion, we examined those presented in papers favoring the resumption model [2], [3], and chose the ones that made the strongest case for resumption. In both examples, the solutions achieved using resumption were more natural than those possible without resumption. However, unless it is shown that such cases arise frequently, they do not justify the more complex mechanism.

VI. Discussion

In this paper we have discussed exception handling and described an exception handling mechanism. An exception handling mechanism is a tool for enhancing program reliability and fault tolerance. To enhance reliability procedures should be defined as generally as possible, that is, they should respond "reasonably" in as many situations as possible. An exception handling mechanism simplifies the writing of such procedures; it is primarily a mechanism for generalizing the behavior of procedures.

In Section II we discussed major decisions that must be made in designing an exception handling mechanism and the exception handling models that result from these decisions. We argued that any well-structured mechanism should be one-level: only the caller should handle exceptions raised by the invoked procedure. We further argued that the termination model, in which the signaling activation terminates, is better than the resumption model, in which the signaling activation continues to exist. The termination model is clearly simpler than the resumption model; we also believe that it has sufficient expressive power. Note that in our termination model, a procedure may terminate in one of a number of conditions (one of which is the so-called "normal" condition) and may return result objects differing in number and type for each condition. The ability to return objects provides a kind of expressive power not found in most other exception handling mechanisms.

Section III described the syntax and semantics of the CLU exception handling mechanism, which supports the termination model. While in Section II we were concerned primarily with interprocedure control and data flow, in Section III, we were concerned primarily with intraprocedure control and data flow. Our goal was to permit the programmer to place handlers where they are needed, without constraints due to conflict of exception names. This goal led to the introduction of an exit mechanism similar to those described by Zahn [9] and Bochmann [1]. Our design also acknowledged that many exceptions cannot be handled. These exceptions may not occur often, but they can potentially occur almost anywhere. The special exception named *failure*, which is signaled implicitly for all uncaught exceptions, was introduced to accommodate this situation. We also discussed why disabling exceptions is not a good idea, and suggested that research in program optimization techniques may be fruitful in avoiding the cost of checking for errors that are known not to occur.

In Section IV, we discussed two methods of implementing the exception handling mechanism, the branch table method and the handler table method. Both methods process normal returns as fast as possible; the branch table method also processes exceptions as fast as possible, while the handler table method is somewhat slower, but more space efficient. We also discussed the integration of the mechanism in debugging and production environments. The mechanism is defined to communicate information that can be used by *programs*, but this does not preclude an implementation that produces additional information for use by *programmers*.

In Section V, we discussed the expressive power of our exception handling model. We described two examples commonly put forward to justify the resumption model and discussed how they could be programmed in the termination model. The termination model solutions were inferior to the resumption model solutions. However, we believe that the examples under discussion occur very rarely, so a mechanism like the resumption model, which eases their programming at the cost of extra complexity, is not justified.

The CLU exception handling mechanism has been implemented by the handler table method. We have used the mechanism in writing many CLU programs (for example, most of the CLU compiler is written in CLU). We are convinced that our programs are better structured than they would be in the absence of the mechanism. Furthermore, we have not encountered any situations where a more powerful exception handling mechanism (e.g., resumption) was desired. Thus, our experience so far supports our belief that the mechanism is a good compromise between expressive power and simplicity. However, we have not written programs that attempt to handle the problem of resource constraints, a situation where resumption is most likely to be needed. Further experimentation is needed to reach a final conclusion on the wisdom of our choices.

Acknowledgment

The design of our exception handling mechanism was the work of the CLU design team, including R. Atkinson, T. Bloom, E. Moss, C. Schaffert, and R. Scheifler. This paper was improved by the comments of the referees and many others.

[6] Each procedure would have to be bound to the environment in which reorganization should be done. Since CLU procedures do not have free variables, the storage pool would have to maintain these environment objects also.

REFERENCES

[1] G. V. Bochmann, "Multiple exits from a loop without the GOTO," *Commun. Ass. Comput. Mach.*, vol. 16, pp. 443–444, July 1973.
[2] J. B. Goodenough, "Exception handling: Issues and a proposed notation," *Commun. Ass. Comput. Mach.*, vol. 18, pp. 683–696, Dec. 1975.
[3] R. Levin, "Program structures for exceptional condition handling," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, June 1977.
[4] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in CLU," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 564–576, Aug. 1977.
[5] *Proc. ACM Conf. on Language Design for Reliable Software, SIGPLAN Notices*, vol. 12, Mar. 1977.
[6] J. G. Mitchell, W. Maybury, and R. Sweet, "Mesa language manual," Xerox Res. Cent., Palo Alto, CA, Rep. CSL-78-1, Feb. 1978.
[7] P. M. Melliar-Smith and B. Randell, "Software reliability: The role of programmed exception handling," in *Proc. ACM Conf. on Language Design for Reliable Software, SIGPLAN Notices*, vol. 12, pp. 95–100, Mar. 1977.
[8] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 220–232, June 1975.
[9] C. T. Zahn, Jr., "A control statement for natural top-down structured programming," *Programming Symposium, Lecture Notes in Computer Science*, vol. 19, B. Robinet, Ed. New York: Springer-Verlag, 1974, pp. 170–180.

Barbara H. Liskov received the B.A. degree in mathematics from the University of California, Berkeley, and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA.

From 1968 to 1972, she was associated w the Mitre Corporation, Bedford, MA, wh she participated in the design and implemer tion of the Venus Machine and the Venus erating System. She is presently Associ Professor of Electrical Engineering and C puter Science at the Massachusetts Institute Technology, Cambridge. Her research inter include programming methodology, distribu systems, and the design of languages and s tems to support structured programming.

Alan Snyder received the S.B., S.M., a Ph.D. degrees in computer science fro the Massachusetts Institute of Technolog Cambridge.

He is currently a member of the Technic Staff in the Computer Research Laborato at Hewlett-Packard Laboratories, Palo Alt CA, working primarily in the area of integrate circuit design automation. His other interes include programming languages and machi architecture.

Dr. Snyder is a member of the Association for Computing Machinery

Proving Total Correctness of Parallel Programs

ALAN F. BABICH, MEMBER, IEEE

Abstract—An approach to proving parallel programs correct is presented. The steps are 1) model the parallel program, 2) prove partial correctness (proper synchronization), and 3) prove the absence of deadlock, livelock, and infinite loops. The parallel program model is based on Keller's model. The main contributions of the paper are techniques for proving the absence of deadlock and livelock. A connection is made between Keller's work and Dijkstra's work with serial nondeterministic programs. It is shown how a variant function may be used to prove finite termination, even if the variant function is not strictly decreasing, and how finite termination can be used to prove the absence of livelock. Handling of the finite delay assumption is also discussed. The illustrative examples include one which occurred in a commercial environment and a classic synchronization problem solved without the aid of special synchronization primitives.

Manuscript received April 12, 1978; revised April 30, 1979.
The author is with the Basic Four Corporation, Santa Ana, CA 92711.

Index Terms—Concurrent program, correctness, deadlock, finite delay, finite termination, infinite loops, livelock, mutual exclusion, parallel program, termination, variant function, verification.

INTRODUCTION

An abstract model general enough to capture most notions of parallel computation is highly desirable. Three crucial parts of such a model seem to be as follows:

1) The state must factor into a control part and a data part, so that such topics as "the number of processes at a given point in the program" may conveniently be discussed.

2) The atomic actions must be specifiable, for no coarser level of detail will, in general, suffice for rigorously proving the correctness of parallel programs.

3) It must be possible to ignore irrelevant details of the computation including absolute and relative execution timings,

0098-5589/79/1100-0558$00.75