# RESPONSE TO DETECTED ERRORS

# IN WELL-STRUCTURED PROGRAMS

D. L. Parnas

July, 1972

This research was supported by the National Science Foundation under grant GJ 30127 to Carnegie-Mellon University.

ABSTRACT

This paper discusses an approach to handling run time errors in well-structured programs. It is often assumed that in well-structured programs which can be proven correct errors will not be a problem. This paper is predicated on the assumption that run time errors will continue to be a problem.

This paper describes an organization for structured programs which attempts to satisfy the following criteria:

(1) Error response routines are written by each programmer in terms of the abstract machine which he uses for his normal case code. Errors are reported in those terms. He is never forced to use information about the implementation of other levels in the system.

(2) Programs can be written so that the code for error detection, error correction, and normal case, are lexically separate and can be modified independently.

(3) The system can evolve from one which does little error recovery to one which introduces quite sophisticated techniques without a change in structure.

(4) Even with unsophisticated recovery procedures, the task of locating the module containing a bug discovered at run time does not require knowledge of many modules.

## 1. INTRODUCTION

Perhaps because structured programming is advanced as a means of eliminating errors in programs, programs written to demonstrate structured programming (e.g., [3,5]) are written assuming that each subprogram will always perform correctly. Moreover, each program is written on the assumption that it itself will never behave incorrectly.

This paper is written on the assumption that, although structured programming will help, run time errors will be with us for a while.

Three justifications for this assumption are:

(1) even the best of "structured programmers" occasionally err;

(2) the apparatus on which we run occasionally fails and may cause a program to fail (either directly or by causing a change in code or data);

(3) in practice programs are changed and errors appear which had not appeared before.

Given this assumption, a system intended to be reliable must be designed with error handling as a fundamental consideration.

This paper suggests a design approach which we believe can increase reliability. It is not particularly concerned with detecting errors; it is concerned with the response to the detection of an error. We are not primarily concerned with debugging (the programmer's response to a detected error); we are concerned with the program's response to a detected error. Such responses include attempts at self diagnosis, saving of partial results, printing of diagnostic information, etc.

This paper does not present an algorithm for error recovery. The paper does present a scheme for program organization which facilitates the introduction of recovery and diagnostic algorithms. It also presents a list of guidelines to help the designer in anticipating the types of errors which might occur.

## 2. DIFFICULTIES INTRODUCED BY A "LEVELED STRUCTURE"

To understand the proposal of this paper one must understand the concept of a hierarchically structured system [3]. One must recall that the lower levels must function without the presence of the upper levels, and they can be used by a variety of upper level programs. It follows that the lower levels cannot use any knowledge of the higher levels. However, error recovery usually requires the combined action of several levels. An error will be detected by a lower level but information available only at a higher level determines the appropriate action.

This is a special case of the observation that structured programming introduces a compartmentalization of knowledge and may make more difficult any action which requires knowledge from several compartments.*

## 3. THE EFFECT OF ERRORS ON CODE COMPLEXITY

A straightforward machine language program to write on a file is usually naive. Error probabilities are relatively high in peripherals; the code needed for error detection and correction makes the programs quite complex. As a result even a change in the normal case procedure is difficult.

---

*Appendix 2 should be read at this point by readers who do not accept the above paragraphs.

Such complications can occur at all levels in a structured program. They are most apparent at the I/O level because of the probability of error being higher there, but the problems are not essentially different at other levels.

To keep the code for the normal case separate from the code concerned with errors, we propose that modules in a structured system make use of a software analog of a "trap". Most computer hardware is designed to detect common errors and transfer control to a specified location upon detecting such an error. Typical trap conditions are "divide by zero" and "memory bounds violation". Traps allow simpler code because one need not include checks for those errors in the program. Traps also decrease the probability of such errors going undetected.*

In the examples given in [1] the modules are specified to call user provided subroutines under conditions which we interpret as errors. In fact, this is the only way that module restrictions are specified! The subroutine names correspond to the hardware trap locations. The user of those modules may write his code without checks for violations of module restrictions. The code concerned with error recovery is confined to the trap routines. This organization achieves lexical separation of normal use, detection, and correction procedures, thereby easing changes.

We state our first suggestion: Place responsibility for the detection of attempts to violate its specifications in the "abstract machine"; it calls a trap routine upon detection of such an error. Other errors, failures of the virtual machine, will also be reported by traps. The remainder of this paper assumes such an organization.

* It has been suggested that traps provide a convenient mechanism for reporting infrequent events to programs which would otherwize need to make frequent checks. Errors, the subject of this paper, are only special cases of that class of situations [7].

It is intended that the trap handling routines be written by the user of a virtual machine and have access to the data used by that user's "virtual program". It is desirable that the programs written for a "virtual machine" can alter the routine associated with the trap routine name.

## 4. NON-MAINTAINABLE ABSTRACTIONS

In this paper we are assuming that systems are structured according to the recommendations of [2] and [3]. Each program is written in terms of an abstraction of the remaining portions of the system; its correctness depends upon a small subset of the properties of the code that it calls upon. This section illustrates that the need to make appropriate responses to errors often severely limits the abstractions we may use.

The structure of a program will be less clear if the user of a module cannot write all of his code in terms of the abstract model he is given [4]. Consequently we cannot abstract from facts which should be used to recover from (or diagnose) an error.

As an example consider a virtual machine which provides instructions which perform "simultaneous" string substitutions on every line of a file. The substitutions can be irreversible (one cannot tell where the change was made by looking at the file afterwards). Let us further assume that the specification given to the users of this machine completely hides the processing sequence (giving the appearance that all lines are processed simultaneously).

Real difficulties in implementing such a virtual machine arise because execution of the machine "instruction" will extend over a measurable

period of time and might be interrupted by an error. If the file is partially processed, recovery will depend upon the user's ability to know which parts of his file have been operated upon. When this depends upon the sequence of processing, he must know "hidden" information.

One solution would be to keep, within the "virtual machine", information sufficient to restore the file to its original state. This solution usually has a very high cost. If one made a module with such a specification, there would be many situations in which one could not afford to use it.

Often a practical solution is to make the module somewhat less abstract. The specification must admit to the possibility of error and provide information to assist in error recovery. The set of "degraded" designs includes designs which specify the sequencing and designs that mark unchanged and erroneous parts of the file. Unless we abandon the idea of abstraction completely, none of these designs always presents the information necessary for recovery. We can, however, handle the most frequent errors by modeling the errors as a set of abstract errors rather than ignoring them!

The above brings out the second suggestion of this paper: Do not specify a module or level to be an abstraction which errors will frequently deny.

## 5. ERROR TYPES AND DIRECTION OF PROPOGATION

An error detected at any given level in a system may be either propogating downward (violating the specified restrictions on the virtual machine) or propogating upward. The upward propogating errors may represent

either failures of a mechanism which has been used correctly, or they represent "reflections" of an error which had previously propogated downward. We shall deal with these cases in turn.

When detected, a downward propogating error should be returned to the level above. Responsibility for diagnosis and possible recovery must lie at the higher levels because the lower level program does not have sufficient knowledge to determine what was desired. (See Appendix 2 for examples.) With the "trap" mechanism this results in a call to a subroutine written by the last caller. Thus, when a downward propogating error is detected, it is reflected to higher levels.

A level is informed of an upward propogating error when a "trap" handling routine is called. If the routine handles a reflected error of usage, it should first determine whether or not the error originated at its own level. If it determines that the error of usage occurred at a higher level, it must adjust its external state*, and call a trap routine above. If it is determined that the error has returned to its original level, the program may either attempt recovery, or inform the next higher level of an error of mechanism (by calling a trap routine).

When a level is informed of an error by the machine below it, it may either attempt recovery (by means of retry,or an alternative program) or adjust its external state and report the error still higher. Any of these routines may also produce diagnostics for programmers.

Normally, the lower levels of a system do not abort the job in the event of a failure of mechanism. At some higher level recovery or loss minimization procedures may be available. Job abortion occurs only at the highest level (or when the call mechanism fails).

*We elaborate on this later.

To summarize, upon detecting an error in a hierarchically structured piece of software the error is first reflected and control passed to the level where the error originated. At this point it is either corrected or reflected still higher as an error in mechanism. At every level,either recovery is attempted or the error is reported still higher. At each level,the error handling routines have the responsibility of restoring the state of the virtual machine used by the level above to one which is consistent with the specifications. All possible efforts are made to assure that no program is given control with its virtual machine in an "impossible" state.*

The above is only a skeleton into which various error recovery and diagnostic policies may be fit. The meta-structure proposed has three advantages:

(1) It allows each error handling procedure to be written at the level where the necessary knowledge exists and in terms of the virtual machine. This preserves the modular structure [4].

(2) <u>It provides for the evolution of a system towards increased reliability without major revisions</u>. Usually when the system is first assembled the error "trap" routines are primitive. They may do no more than print their name. As the development progresses, increased experience and understanding allows these routines to be replaced with more sophisticated diagnostic, recovery, or loss minimization routines.

(3) The use of even the trivial versions of the trap routines

*With a precisely defined machine (real or virtual) certain relations between the functions may be "proven" by taking the specification as a set of axioms. A state in which those relations do not hold is termed "impossible".

greatly simplifies debugging once the system has been "integrated". When a system has been produced by the cooperation of many programmers, no one knows the complete system well. When a bug appears, it is a difficult job to determine which programmer should study the problem. In our experience in testing systems whose error policies approximate those suggested in this paper, error routines which do no more than print out their own name usually indicate which module (and which programmer) is at fault. We make great efforts to avoid having bugs which show up after the modules are combined, but when we fail, the above becomes useful.

## 6. SPECIFYING THE ERROR INDICATIONS

When a module is designed and specified we specify all the limitations of the program and all the error calls which will be made in the event that those conditions are violated. We also specify routines to be called in the event of certain other failures. The following is a list of considerations which must enter into the construction of the list. It may be viewed as an aid to error anticipation.

6.1 Limitations on the values of parameters. Since any piece of software has a limited range of parameters which it can handle, a trap should occur if these are violated. These should be omitted only if it would be impossible to violate them (e.g., if "compile time" checks are feasible).

6.2 Capacity limitations. Since any module which stores information will have a finite storage capacity, traps should occur when that

capacity is exceeded. The specification must enable users to predict when such a trap will occur (i.e., to determine the capacity).

6.3 Requests for undefined information. Any module which provides a memory function must be designed in the light of the possibility that information will be requested before it has been inserted or after it has been deleted. Traps should be specified for all such conditions.

6.4 Restrictions on the order of operations. Efficiency, ease of implementation, or a desire to detect probable programming errors, may dictate a restriction on the order of calls on a module's functions. For example, most file systems require "opening" a file before one may access it. Traps should be specified for violation of these restrictions. It is sometimes necessary to add functions to a module in order to specify the conditions under which such traps occur. In the file example a predicate "OPENED" would be appropriate. (See also Appendix 2.)

6.5 Detection of actions which are likely to be unintentioned. Experience has shown us a common class of programming errors which result in certain "strange" actions. For example, the opening of a file which is already open is often indicative of an error. Many pieces of software use the unlikely action as a way of encoding some other operation (e.g., the closing of the file). We prefer to specify traps for such occurrences and provide alternative means of performing the other operation. Then a user has the option of specifying the alternative operation as the body of his trap routine. This particular recommendation is a question of taste. Modules designed in this way often have restrictions that some find annoying.

6.6 Sufficiency. The above list of downward propogating error checks could be summarized as follows: The set of error trap conditions specified should be sufficient to guarantee that,if none of them applies, the change specified as the effect of calling the routine could be carried out without violating any module limitations. Further, the fact that no trap occurs, should guarantee that the value of the function (if any) will not be "undefined".

6.7 Priority of traps. A single erroneous call may violate several of the trap conditions mentioned above. It is not usually useful to call several trap routines. Instead we assign a priority to each trap and specify that only the highest priority "enabled" trap to be called. (In [1] the priority was indicated by the sequence of the calls in the text.) Priority assignment becomes essential when the value of some functions in the trap definitions might be undefined in an erroneous call. Then the priority of the traps must guarantee that there will be an enabled trap with a higher priority than any error condition which mentions undefined functions (see Appendix 1).

6.8 Size of the"trap vector". The structure and efficiency of the individual trap routines is improved as the class of errors they handle is restricted. The analysis done by the routine to determine the exact error often computes information which was known to the calling module. However, one must also avoid specifying a very large number of distinct error routines. One can combine several similar conditions to reduce the number of distinct routines. The optimal "trade-off" is a function of: (1) the sophistication of the error diagnosis being attempted

(which determines the number of routines which would actually be different) and (2) a complex space-time tradeoff. A practical compromise is to combine similar conditions and pass a parameter indicating the actual error.

6.9 State after the trap. Programming is simplest when the module had no external changes after an error call. When it is not practical to adhere to such a rule, the trap should not occur until sufficient information to determine the state change is made available to the callers. The trap routine has the option of executing "return" after attempting correction of the error; the module should then continue after ascertaining that the call is now correct. If continuation is impossible, the trap specification should make that clear. A return in such cases can be handled by calling a trap routine used only for such an illegal "return".

6.10 Errors of Mechanism. Reporting a failure by the module is inherently more difficult than reporting the downward passing errors we have been discussing. The actual error can only be accurately described in terms of information which has been hidden from the user. He could not use an accurate report. We want to give him abstract information which may help him in recovering; we are again faced with a trade-off between the simplicity of the design and the accuracy or detail of the abstract report. At one extreme we use a single trap name to report "failure" and require that the user of the module run diagnostic programs on his virtual machine to determine the extent of the damage. Experience with hardware diagnostic programs teaches us that this is quite a difficult

task. In the case of a "virtual machine" there are many types of failures in which the module has the capability of delivering quite a detailed analysis of the damage to the virtual machine. For example, a file system is usually capable of giving its users a list of damaged records and even a list of "commands" which no longer work correctly. However, some failures are so catastrophic that the information is not available. In the example given in Appendix 1 we have chosen a design in which the "failure" error call routines pass a parameter which classifies the type of failure. These classifications allow the user to answer such questions as:

(1) Did the command which failed change any function values?

(2) Is it possible that a retry would work?

(3) Were functions other than the one called affected?

(4) Was the module able to restore functions to a state consistent with the specifications or is the machine in an "impossible" state?

We considered an alternative which was further towards the fully detailed extreme. In this alternative we would have added a predicate associated with each function; the predicate would be true if the failure had affected proper functioning of its associated function. There would also have been a predicate which would be true if the module had been unable to set the value of the previously mentioned predicates properly. This predicate would have been true in catastrophic failures. (There would always be the possibility of a catastrophe so great that even the last predicate could not be properly set.) In an extreme alternative,

the predicates had as many parameters as their associated functions and would provide true or false indications for each possible call.

We rejected these alternatives because:

(1) It seemed unlikely that one would want to make an implementation which was sufficiently redundant that it would be able to provide such detailed information.

(2) It seemed unlikely that a user program would be written to use such information.

Our decision we made was based upon a certain expected set of applications and would be wrong for some. We present it only as an example of one solution to this class of problem.

## 8. REDUNDANCY AND EFFICIENCY

Modules designed as described above can be thought of as highly insulated external programs; the traps can be viewed as a wall protecting the module from damage. In a system constructed with such a view, much of the system resources are applied to maintaining the walls. For example, as a particular value is passed through several modules it will be repeatedly checked against the same limits. Such redundancy is extremely valuable in the early testing stages, but when the system is reliable the inefficiency introduced by the redundant checking becomes significant.

When errors are quite rare, we can eliminate some of the redundant checks.

Here one can discern two distinct approaches. (1) Retain the upper level checks, eliminate the lower level checks, assuming that no error

will be introduced in the variable on its way down. (2) Retain the lower level checks, use the trap routines at the intermediate levels to pass the error back up to the point where it occurred. The second is usually preferable, but there are exceptions. When there are difficulties in the "backing up" which is sometimes needed in the second approach, the first approach can detect errors before changes are made.

## 9. EXAMPLES

Appendix 1 gives an example of a module specified in accordance with this paper. The notes annotating the example indicate which sections of the paper gave rise to particular decisions in the specification. Appendix 2 is a narrative of an error traversing several levels.

Space does not permit us to discuss a whole system in great detail. The reader might wish to look at [2] where all the modules of a small system are presented. In that example we were forced to ignore errors of mechanism because the lowest level was a commercial Fortran implementation which did not permit the fielding of errors by user provided software.

## 10. CONCLUSIONS

We find it unfortunate that our conclusions are based on a small set of experiences on small scale systems with inexperienced programmers. This limited experience supports the following conclusions:

1. Proper handling of errors requires that a systematic approach to error handling be taken in <u>every</u> part of the system. Most of our difficulties with errors occurred because our "lowest" level, the commercial system that we were using, did not follow our approach.

2. The trap approach appears to be enormously helpful, but the use of FORTRAN subroutine calls introduces three important difficulties. The caller's identity is unavailable to the FORTRAN routine; reassigning the contents of the trap locations dynamically was unnecessarily complicated and transfer of control between error routine and main program is unecessarily restricted. These difficulties could and should be corrected in a professional attempt to apply our techniques.

3. Our ability to abstract did not appear excessively restricted by the necessity of considering errors in designing the abstraction.

4. Reflection of downward traveling errors and the passing of failures upward appears, on the basis of very limited trials, to be workable and useful. Reflection provides a basis for eliminating redundant error checks except in the (hopefully) rare case of actual occurrence of an error.

5. The consideration of error possibilities will require half and sometimes more than half of a designer's effort in writing specifications for his modules in our present efforts. In our own evaluation this is a reasonable price for the potentially increased reliability of the system.

6. Our proposal is one which, <u>at first glance</u>, violates a fundamental rule of hierarchically structured systems. In previous examples of such systems it has been presumed that a program at level i calls only programs at level i-1 and lower. We now have a scheme in which programs call error routines written at higher levels. In the previous situation there was a simple formal test which one could make to test whether or not the system was hierarchically structured. It is important that we now find a basis

for mechanical checking of hierarchical structuring. Merely labeling a routine "error routine" does not have any significance.

We propose that the necessary information is contained in the specification of the modules. The correctness of a given module's implementation is dependent upon the lower level routines which it chooses to call. If those lower level routines (which are not named in the specification) fail, causing the module to fail, the module will not be considered to meet its specifications. In contrast, the module is specified only to call the trap routine. Its responsibility ends with the call. The module will be considered correct even if the trap routine is absent. This is analagous to the hardware which is considered correct whether or not trap routines are provided.

We propose then that the test for hierarchical structure (which requires that a program complete all specified actions without calling upon higher level routines) specifically make an exception of the calling of routines named in its specification. Under this definition, the systems we discussed have a hierarchical structure and the concept of "virtual machine" is still valid.

7. Some readers have suggested that instead of trap routines the upper levels leave encoded instructions for use by the lower levels in the event of error. Such a solution only replaces a machine interpreted trap routine by a software interpreted one.

8. We feel that an organization similar to the one proposed is an essential step towards the production of highly reliable systems.

Acknowledgement: I am grateful to P.J. Courtois, H.D. Wactlar, Dr. James S. Miller, A. Newell, A. Jones for helpful comments on versions of this paper. Many of the ideas in this paper were suggested by the work of systems programmers who informally organized parts of this program this way. The assistance of their examples in suggesting the guidelines offered here is acknowledged.

References

[1] Parnas, D. L., A Technique for Software Modules Specification with Examples, Carnegie-Mellon University Technical Report, 1971, Communications of the ACM, May 1972.

[2] Parnas, D. L., On the Criteria for Decomposing Systems into Modules, Carnegie-Mellon University Technical Report, 1971, to be published, Communications of the ACM, 1972.

[3] Dijkstra, E. W., Notes on Structured Programming, T.H.E., Eindhoven, The Netherlands.

[4] Parnas, D. L., "Information Distribution Aspects of Design Methodology", Proceedings of IFIP Congress 71, 1971.

[5] Wirth, N., "Programming by Stepwise Refinement", Communications of the ACM.

[6] Parnas, D. L., Some Conclusions from an Experiment in Software Engineering Techniques, Technical Report, Computer Science Department, Carnegie-Mellon University, to be published in Proceedings 1972 FJCC.

[7] A Jones, Private Communication.

# APPENDIX 1

## ANNOTATED EXAMPLE OF MODULE DESIGN IN THE LIGHT OF ERRORS

### INTRODUCTION

Figure 1 is a module specification using the technique described in [1]. The module specified is a modification of an example from that paper. With one minor exception all changes from the earlier version are a consequence of the considerations in this paper. The notes below refer to markings in Figure 1.

1. This function has no parameters and may always be called. The only trap provided is for the case that the module fails. The function represents the number of nodes which may yet be added to the tree and is included so that the user of the module may predict when the trap routine EC41 or EC46 will be called. See also (8) below.

2. The only limitation on this function call is the size of the parameter (i.e., the maximum integer which may be a node identifier) as discussed in section 6.1 of the paper.

3. Here we have an illustration of the ordering suggested by the priority considerations in section 6.7. If EC4 is not called, the value of EC5 should be defined. It would only make sense to call EC6 if EC4 or EC5 need not be called.

4. The function VALDEFD (Value defined) is included in order to specify a trap if someone attempts to read a value stored at a node in

the tree (by calling VAL) before setting that value (by calling SVAL). This is according to the considerations in section 6.3.

5. Functions ELS and ERS (Exists Left Son and Exists Right Son) are included so that the user can predict the conditions under which EC20 and EC24 would occur.

6. The inclusion of the separate functions SVAL and CVAL (Set VAL and Change VAL) is an example of the attempt to trap probably user errors as discussed in section 6.5. The design makes the assumption that setting a value for a node which already has a value is, in many applications, an error and requires a distinct function CVAL for that case (alternatively we could require deletion of the node, but that would introduce great inefficiency). In most programs this would cause no inconvenience. If it did, the body of EC28 could be a call on CVAL. This of course is an external change of design which is less efficient than the corresponding internal change would have been.

7. We have specified a module in which deletion of a node which still has descendants is illegal. This is certainly a debatable design decision. It might trap some errors, but it can force ineficiency when it is desired to delete a whole subtree. Were this to become a problem, we would add yet another function to delete a whole subtree.

8. The three points marked "(8)" illustrate a difficulty in trying to make the calls on error routines completely predictable yet not reveal the implementation. Our manipulations on SPSLFT make the assumption that

space for storing VAL is allocated when the node is created. In some implementations that would not be so. For those implementations the specifications will require a call to EC41 or EC46 in some cases when space is actually still available. If, however, we took the obvious alternative and made separate changes to SPSLFT for creating a node and SVAL, we would be restricting our implementation to one which made separate allocations. Such implementations would use more space and would be undesirable if SLS or SRS were always followed directly by SVAL. Note that the elimination of space limitations would violate sections 6.2 and 6.6.

9. Note that the specification does not specify the value of LS(i); only some properties of it. Further note that this specification is acceptable only on the very reasonable assumption that p1 (the maximum number of node names) is not less than p2 (the maximum number of nodes). If that assumption were violated, we would have to introduce error calls for the situation where there does not exist a value of k with the properties specified.

10. Note that it would be quite reasonable to reduce the size of the trap vector by combining EC41 and EC46. See section 6.8.

11. The error calls EC1, EC3, EC6, EC9.....report failures of mechanism rather than an incorrect call. We have chosen to have each of these pass a parameter k which will indicate the class of failure which has occurred. The values of k are defined as part of the specifications.

The important thing to note is that the meaning of each particular possible value of k is defined in terms of external properties of the module. If the user had kept redundant records he would be able to

determine which value of k applied by diagnostic testing. We pass the value of k so that he will not need to keep such records and on the assumption that reasonable implementations will be able to determine the proper value under all but the most catastrophic of failures. The last value is an escape for such cases.

As mentioned in section 6.10, this particular design is but one point on a scale which includes many possibilities. We give it as a reasonable but not necessarily optimal design.

# APPENDIX 2

## EXAMPLES IN WHICH ERROR MESSAGES MUST BE PASSED BETWEEN LEVELS

It may not be obvious to some readers that errors in a hierarchically structured system must be handled at levels other than that at which they are detected. We present two examples as a means of showing why the detecting level may not have the information necessary to perform the proper action.

Example 1. Bad Tape Block

An unreadable tape block will be detected at the lowest level because the hardware will signal its presence. The low level program which has been ordered to read a given tape block, has no knowledge of the intended use of the information and can take no corrective action. Some levels higher we have a program providing a simple sequential access method. This program knows that the block was part of a given file but no more. Still higher we might find a program managing a large data base. This program might know that the block in question was part of a summary file which had just been computed from a master file and the record could be recomputed. Alternatively, the system might not be that sophisticated, but the error could be passed higher to the user who is able to give instructions for recovery.

Example 2. Out of Date Directory

Due to a software error a file is changed while a copy of its directory still exists. A program using that old directory attempts to read the file and ultimately receives an error due to some hardware violation. If the

error is passed up to the level which used the incorrect directory, it can check its copy against the master copy and try again. Recovery at the intermediate levels was impossible. If this sophistication were not present, the error could be passed upward to some higher level which would attempt a retry. If the retry involves getting a new copy of the directory, we may well have success.

In these two examples we have tried to show both errors of usage and errors of mechanism. We have also shown that the considerations are important for both hardware and software errors. In all the situations outlined an attempt to handle the error at the incorrect level would have failed due to lack of proper knowledge, the system would have a poorer reliability than necessary. The alternate solution is to introduce the necessary knowledge to the lower level. This clearly would reduce the advantage we have gained from the hierarchical structure.

```
Function SRS
        possible values:  none
        parameters: integer  i
        initial values: not applicable
        effect:
                call EC43 if i < 0 or i > p1
                call EC44 if 'Exists'(i) = false
                call EC45 if 'ERS'(i) = true
(10)            call EC46 if 'SPSLFT' = 0
(11)            call EC47(k) if failure
               there exists k such that [
                  0 < k < p1
(9)               'Exists'(k) = false
                  Exists(k) = true
                  RS(i) = k
               VALDEFD(k) =  false
               ELS(k)=ERS(k) = false
                  ERS(i) =true
                  FA(k) = i ]
(8)            SPSLFT ='SPSLFT'-1
```

Values of k in calls of EC1,EC3,EC6,EC9,EC13,EC15,EC17,EC21,EC25,EC29,EC33, EC37,EC42,EC47

| | |
|---|---|
| k = 0 | value of SPSLFT,Exists,FA,VALDEFD,VAL,ELS,ERS,LS,RS unchanged. successful retry possible. |
| k = 1 | value of SPSLFT,Exists,FA,VALDEFD,VAL,ELS,ERS,LS,RS unchanged. successful retry impossible. |
| k = 2 | value of function called lost or changed, no other changes. "possible state". successful retry possible. |
| k = 3 | value of function called lost or changed, no other changes. "impossible state". continuation impossible. |
| k = 4 | value of function called lost or changed, no other changes. "possible state". successful retry impossible |
| k = 5 | value of functions other than that called changed. "possible state". successful retry possible. |
| k = 6 | value of functions other than that called have been changed. "impossible state". successful retry impossible. |
| k = 7 | value of functions other than that called have been changed. "impossible state". continuation impossible. |

Notes

1. k = 2,3,4 only possible in EC1,EC3 ... EC25
2. k = 0 or k = 1 suggest that information is not lost but growth or change of tree is restricted.
3. "possible state" and "impossible state" are defined in the paper.
4. The design makes the assumption that if the module is unable to restore its external appearance to a "possible state" it cannot continue.
5. "successful retry possible" does not guarantee successful retry. It only means that successful retry is not known to be impossible. This value would be given if the module experienced difficulties which might be resolved externally to the module and suffered no internal damage to its data structures.

```
Function DEL
        possible values: none
        parameters: integer  i
        initial values: not applicable
        effect:
                call EC34 if i<0 or i > p1
                call EC35 if 'Exists'(i) = false
(7)             call EC36 if 'ELS'(i) or 'ERS'(i) = true
(11)             call EC37(k) if failure
               FA(i) is undefined
               VAL(i) is undefined
               ERS(i) is undefined
               ELS(i) is undefined
               VALDEFD(i) is undefined
               Exists(i) = false
               if i = 'LS'('FA'(i))then [
                  LS('FA'(i)) is undefined
                  ELS('FA'(i) ) = false]
               if i = 'RS'('FA'(i))then [
                  RS('FA'(i)) is undefined
                  ERS('FA'(i) ) = false]
(8)            SPSLFT ='SPSLFT' + 1

Function SLS
        possible values:  none
        parameters: integer  i
        initial values: not applicable
        effect:
                call EC38 if i < 0 or i > p1
                call EC39 if 'Exists'(i) = false
                call EC40 if 'ELS'(i) = true
(10)             call EC41 if 'SPSLFT' = 0
(11)             call EC42(k) if failure
               there exists k such that [
                  0 < k < p1
(9)               'Exists'(k) = false
                  Exists(k) = true
                  LS(i) = k
                  ELS(i) =true
                  ELS(k)=ERS(k) = false
               VALDEFD(k) =  false
                  FA(k) = i ]
(8)            SPSLFT ='SPSLFT'-1
```

```
Function SPSLFT
        possible values:   integer
        parameters:none
(1)     initial values: p2
        effect:
(11)            call EC1(k) if failure

Function Exists
        possible values:  true, false
        parameters: integer  i
        initial values: Exists(0) = true;Exists(1:p1)=false; all others undefined
        effect:
(2)             call EC2 if i < 0 or i >  p1
(11)            call EC3(k) if failure

Function FA
        possible values:   integer
        parameters: integer i
        initial values:  FA(0) = 0; all others undefined
        effect:
                call EC4  if i < 0 or i > p1
(3)             call EC5 if 'Exists'(i) = false
(11)            call EC6(k) if failure

Function VALDEFD
        possible values: true,false
        parameters:  integer  I
(4)     initial values: VALDEFD(0) =false; all others undefined
        effect:
                call EC7 if i<0 or i > p1
                call EC8 if 'Exists'(i) = false
(11)            call EC9(k) if failure

Function VAL
        possible values:  integer
        parameters: integer  i
        initial values: undefined
                call EC10 if i<0 or i > p1
                call EC11 if 'Exists'(i) = false
                call EC12 if 'VALDEFD'(i) = false
(11)            call EC13(k) if failure

Function ELS
        possible values: true,false
        parameters: integer  i
        initial values: ELS(0)= false; all others undefined
        effect:
                call EC48 if i < 0 or i > p1
                call EC14 if 'Exists'(i) =  false
(11)            call EC15(k) if failure
```

(5)
Function ERS
possible values: true,false
parameters: integer i
initial values: ERS(0)= false; all others undefined
effect:
call EC49 if i < 0 or i > p1
call EC16 if 'Exists'(i) = false
(11) call EC17(k) if failure

Function LS
possible values: integer
parameters: integer i
initial values: undefined
effect:
call EC18 if i < 0 or i > p1
call EC19 if 'Exists'(i) = false
call EC20 if 'ELS'(i) = false
(11) call EC21(k) if failure

Function RS
possible values: integer
parameters: integer i
initial values: undefined
effect:
call EC22 if i < 0 or i > p1
call EC23 if 'Exists'(i) = false
call EC24 if 'ERS'(i) = false
(11) call EC25(k) if failure

Function SVAL
possible values: none
parameters:integer i,v
initial values: not applicable
effect:
call EC26 if i<0 or i > p1
call EC27 if 'Exists'(i) = false
(6) call EC28 if 'VALDEFD'(i) = true
(11) call EC29(k) if failure
VAL(i) = v
VALDEFD(i) = true

Function CVAL
possible values: none
parameters:integer i,v
initial values: not applicable
effect:
call EC30 if i<0 or i > p1
call EC31 if 'Exists'(i) = false
call EC32 if 'VALDEFD'(i) = false
(11) call EC33(k) if failure
VAL(i) = v