

RESPONSE TO UNDESIRED EVENTS  
IN SOFTWARE SYSTEMS

D. L. Parnas  
H. Würges

Research Group on  
Operating Systems I  
Technische Hochschule Darmstadt  
Federal Republic of Germany

KEYWORDS

Run-Time Errors, Error Recovery, Reliable Systems, Information Hiding, "Uses"-Hierarchy.

ABSTRACT

This paper discusses an approach to handling run-time errors in software systems. It is often assumed that in programs which can be proven correct, errors will not be a problem. This paper is predicated on the assumption that, even with correct programs, undesired events at run-time will continue to be a problem. Routines to respond to these undesired events (UEs) must be provided in reliable systems.

This paper describes a program organization which aims at satisfying the following criteria:

- (1) UE response routines are written by each programmer in terms of the abstract machine which he uses for his normal case code. UEs are reported in those terms. He is never forced to use information about the implementation of other modules in the system.
- (2) Programs can be written so that the code for UE detection, UE correction, and normal case, are lexically separate and can be modified independently.
- (3) The system can evolve from an initial version that does little recovery to one which uses sophisticated recovery techniques without a change in the structure of the system.
- (4) Even with unsophisticated recovery procedures, the task of locating the module containing a bug discovered at run-time does not require internal knowledge of many modules.
- (5) Costs incurred because of the recovery techniques are low as long as no UE occurs.

1. Introduction

Perhaps because structured programming is advocated 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.

Four justifications for this assumption are:

- (1) even the best of "structured programmers" occasionally err;
- (2) the machines which we use occasionally fail 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;
- (4) incorrect or inconsistent data may be supplied to the system.

If this assumption is valid, then it would be foolhardy to make early design decisions on the assumption that errors or other undesired events will not occur. As explained in /10/, the early decision (e.g. the interfaces between the various independently developed components) will be the hardest to change. In real world situations certain undesired events occur frequently, they are expected by the user, and he can define programs to take corrective action when they occur. Often such programs can only be added after a period of use, but the structure of the system must allow for such a likely change or addition to the program. The overall reliability of a system can be significantly increased by the addition of programs which respond to or "handle" UEs.

The term "undesired event" (UE) is introduced, because (1) we want to include all events that result in a deviation from normal behavior, and (2) the term "error" often led to the objection that "errors" should not be handled, but "corrected".

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

This paper does not present an algorithm for recovery from UEs. 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 UEs which might occur.

The concept proposed in section 9 aids in specifying reaction to UEs and enables the user of an abstract machine to state explicitly which UEs he is prepared to "handle" and what he regards as "correct" UE handling.

## 2. Difficulties Introduced by a "Leveled Structure"

To understand the proposal of this paper, one must understand the concept of a hierarchically structured system /8,9/. 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 about the higher levels. However, recovery usually requires the combined action of several levels. An UE will be detected by a lower level, but information available elsewhere determines the appropriate action.

This is a special case of the observation that structured systems compartmentalize knowledge and any action which requires knowledge from several compartments may require more effort as a result of the extra communication needed.

## 3. The Effect of Undesired Events on Code Complexity

A straightforward machine language program to write on a tape is usually naive. The probability of an UE in peripherals is high; the code needed for error detection and correction makes the programs quite complex. As a result, a change in the normal case procedure is difficult.

Such complications can occur in all parts of a system. They are most apparent in the I/O modules because the probability of UEs is higher there, but the problems are not essentially different for other modules.

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

In the examples given in /1/, the modules are specified to transfer control to user-provided routines under conditions which we interpret as UEs. In fact, this is the only way that restrictions on the use of these functions are specified! The user of those functions may write his code without checks for violations of the "applicability conditions". The code concerned with recovery from UEs is called by means of a trap. This organization achieves lexical separation of normal use, detection, and correction procedures, thereby easing changes.

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

Every group of programmers, writing programs using an abstract machine, provides additional programs that will be executed if the abstract machine fails to execute that program correctly. Only these programmers know what their program was intended to do and what should be done in case of an UE. They also know which message should be given to the user of their program, if recovery is not successful or if the UE was caused by a forbidden application of their program.

Routines for UE handling use the same abstract operations and operands as the normal case program, i.e. they use the same abstract machine. This guarantees that no knowledge about lower level programs and about the implementation of the abstract machine is used in UE handling. Otherwise a change in those programs may lead to "incorrect UE handling".

## 4. Impossible Abstractions

In this paper we are assuming that systems are structured according to the recommendations of /2,8/ and /9,10/. Each program is written in terms of abstractions of the code that it calls upon. This section illustrates that the need to make appropriate responses to UEs often severely limits the abstractions that we may use.

The structure of a program will be less clear if the user of an abstract machine cannot write all of his code in terms of the abstract model /4/. Consequently, we cannot abstract from facts which should be used to recover from (or diagnose) an UE.

As an example, consider an abstract 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 ap-

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

pearance that all lines are processed simultaneously).

Execution of the machine "instruction" will extend over a measurable period of time and might be interrupted by an UE. If the file is partially processed, recovery will depend on 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 "abstract machine", information sufficient to restore the file to its original state /11/. 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 UEs and provide information to assist in recovery. The set of "degraded" designs includes designs which specify the sequencing as well as designs that mark unchanged and erroneous parts of the file. Unless we abandon the idea of abstraction completely, no design always presents all of the information usable for recovery. We can, however, prepare for the most frequent UEs by defining a set of abstract UEs.

The above brings out the second suggestion of this paper: Do not specify a module to have properties which UEs will frequently violate. One must include in the interface the necessary operations to communicate about the occurrence of UEs. An example can be found in Appendix 1.

## 5. Error Types and Direction of Propagation

An UE detected at any given level in a system may be propagating either downward (violating the specified restrictions on the virtual machine) or upward. The upward propagating UE represents either failure of a mechanism which has been used correctly, or it represents "reflection" of an UE which had previously propagated downward. We shall deal with these cases in turn.

When detected, a downward propagating UE should be returned to the level above. Responsibility for diagnosis and possible recovery must be assigned to 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 transfer of control to a routine designated by the last caller. Thus, when a downward propagating UE is detected, it is reflected to higher levels.

An upward propagating UE is reported by "trapping" to an UE handling routine. If the routine responds to a reflected error it should first determine whether or not the original error occurred at its own level. If it determines that the "error of usage" occurred at a higher level, it must adjust its external state\*, and report the UE to its user. If it is determined that the UE has returned to its original level, the program may either attempt recovery or inform the next higher level of an "error of mechanism" (by use of a trap).

\* We elaborate on this later.

When a level is informed of a failure of the machine below it, it may either attempt recovery or adjust its external state\* and report the UE still higher. Any of these routines may also produce diagnostics for programmers as side-effects.

The lower levels of a system should never abort the job in the event of a failure of mechanism. Recovery or loss minimization procedures may be available elsewhere. Job abortion occurs only as a last resort (e.g. when the trap mechanism fails).

To summarize, upon detecting an UE in a hierarchically structured piece of software, the UE is first reflected and control passed to the level where it originated. At this point it is either corrected or reflected still higher as a "defective virtual machine". At every level, either recovery is attempted or the UE is reported still higher. At each level, the UE 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.\*\*

## 6. Continuation After UE "Handling"

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

(1) It allows each UE handling procedure to be written at the level where the necessary knowledge exists and in terms of the virtual machine. This does not violate the "information hiding" principle /4/.

(2) "uses" still defines a hierarchy allowing usable subsets /8,9/.

(3) It provides for the evolution of a system towards increased reliability without major revisions. Usually, when a system is first assembled, the UE handling 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 and loss-minimization routines.

(4) The use of even the trivial versions of the trap routines 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 UE policies approximate those suggested in this paper, UE routines which do no more than print out their own name usually indicate which module (and

\*\* 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".

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 UE routines become very useful.

## 8. Specifying the Error Indications

When a module is designed and specified, we specify all the limitations of the program and all the UEs which will occur in the event that those applicability conditions are violated. We also specify routines to be called in the event of certain classes of internal 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 UE anticipation.

8.1 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 can be omitted if it is impossible to violate them at run-time, because "compile time" checks are feasible.

8.2 Capacity Limitations. Since any module which stores information will have 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).

8.3 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.

8.4 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 useful to add functions to a module in order to simplify the specification of the conditions under which such traps occur. In the file example a predicate "OPENED" would be appropriate. (See also Appendix 2.)

8.5 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 UE routine. This particular recommendation is a question of taste. Modules designed in this way often have restrictions that some find annoying. Some people prefer executing OPEN for an OPEN file to checking to see whether or not it is already opened.

8.6 Sufficiency. The above list of applicability conditions could be summarized as follows:

The set of 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".

8.7 Priority of Traps. A single erroneous call may violate several of the applicability conditions mentioned above. It is usually not useful to trap to several UE routines. Instead, we assign a priority to each trap and specify that only the highest priority "enabled" trap will occur. (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 applicability conditions might be undefined in an erroneous call. The priority of the traps must guarantee that there will be an enabled trap with a higher priority than any UE condition which mentions undefined functions (see Appendix 1).

8.8 Size of the "Trap Vector". The structure and efficiency of the individual trap routines can be improved by restricting the class of UEs they handle. The analysis done by the routine to determine the exact UE often computes information which was known to the "trapping" program. However, one must also avoid specifying a very large number of distinct UE 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 UE 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 providing additional UE information.

8.9 State After the Trap. Programming is simplest when the module did not change state if an UE occurred. 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 users.

8.10 Errors of Mechanism. Reporting a failure by the module is inherently more difficult than reporting the violation of applicability conditions. The actual UE 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 (i.e. to classify the UE) 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 software-implemented "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" trap 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 the value of any function?
- (2) Is it possible that a retry would work?
- (3) Were functions other than that which was 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?

Information of this sort can be mentioned in the specifications where a means for communicating supplementary information can be defined. 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 problems.

## 9. 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 UEs are quite rare, we can eliminate some of the redundant checks.

Here one can discern two extreme approaches.

- (1) Retain the upper level checks, eliminate the lower level checks assuming that no UE will affect the variable on its way down.
- (2) Retain the lower level checks, use the trap routines at the intermediate levels to pass the UE back up to the point where it originated. The second is usually preferable, but

there are exceptions. It may be difficult to effect the "backing up" which is sometimes needed in the second approach; the first approach can detect UEs before irreversible changes are made.

## 10. Degrees of Undesired Events

In /12/ Krakowiak and Kaiser distinguished two types of errors: incidents and crashes. Incidents are events which, although undesired, were expected and where recovery attempts were successful.

All other errors are called crashes. This distinction may be refined to allow several degrees of UEs. Each degree corresponds to a set of predicates which must be satisfied if recovery is to be considered successful. Degree 0 describes normal behavior of the program. If the requirements for degree  $i$  cannot be met, the system attempts to satisfy degree  $i+1$ . An UE that is successfully handled by satisfying the requirements of degree  $i$  is termed "an UE of degree  $i$ ".

Distinguishing several degrees of UEs enables a programmer to exactly define (1) what he expects his program to do, (2) what he wants to treat as an incident which he is prepared to handle, and (3) what he means by "correct UE-handling". By this means system specifications which normally define desired behavior only, may be extended to include response to those undesired events which are expected to happen and which the system should be prepared to handle.

### Examples:

- (1) If a deadlock occurs in an operating system (e.g. because of a defect in an external device) this is usually regarded as a normal undesired event. (Note that such a deadlock is possible in spite of a mechanism for deadlock prevention.) Degree 1 may require delaying the processes involved in the hope that continuation will be possible within a short (and fixed) period of time. If this is not possible, degree 2 requires storing the blocked processes on secondary storage so that they may be restarted at a later point of time. If this cannot be achieved (because of a lack of free storage), degree 3 may allow deleting one or more processes.

- (2) Assume that a program for matrix inversion fails due to "division by zero". A program on degree 1 that works with greater precision may succeed in case of near singularity. Degree 2 may require an error message.

- (3) Assume that the alphabetical order of a sorted textfile is destroyed. Degree 1 may try to re-establish alphabetical ordering. If this is not possible, e.g. because the separately stored keys are destroyed too, degree 2 may use an old copy and a list of operations that were executed since the copy was made. If this too fails, e.g. because such a list does not exist or is no longer accessible, degree 3 may make the old copy available to the user (together with an apology).

10.1 Why Different Degrees? There are two basic factors which determine the degree of an UE.

- (1) The degree of an UE is determined by

its basic cause. As an example, consider a read error by a card reader. If this UE was caused by a transient error in the power supply, it is usually sufficient to re-read the card. With a permanent defect, however, this does not work. One has to use another card reader or wait until the first one is repaired. Another cause might be a damaged card or an invalid card code.

In many cases the easiest way to determine the cause of the UE is to try several recovery actions. The order of "tries" usually depends on their "cost". One starts with the simplest or cheapest recovery action and only when it fails, one tries the next one.

(2) The degree of an UE depends on the situation at the time the UE occurs. This can be illustrated using the example given above (device failure): In a situation where not all devices are allocated, a failure of one device need not cause a deadlock. One may switch to another device and retry the interrupted operation. Under heavy load, however, this is not possible and a deadlock may occur. Similarly, success or failure of degree 2 (storing the processes involved on secondary storage) depends on the availability of storage.

10.2 Order of Degrees. When we tried to solve the question whether we should try all degrees of one level before we go to the next level or execute the first degree of all levels before trying the second degree, we faced the problem of defining an ordering between degrees. What does it mean that degree  $i$  is lower (i.e. has a lower number) than degree  $j$ ? In the example shown above we stated that some actions are "simpler" or "less costly" than others. But what does that mean and how can simplicity and costliness be determined? In the following some criteria for determining the ordering of degrees are considered.

10.3 Order of Aims. By definition recovery actions associated with degree  $i$  are tried only if it is impossible to satisfy the requirements associated with degree  $i-1, i-2$ , etc. From this one can conclude that the situation achieved by degree  $i$  is less desirable than the aims of lower degrees. Clearly, "less desirable" depends upon the goal or purpose the user wants to achieve. A young boy traveling by bike would always fix a flat tire before continuing his journey. That is the cheapest way for him to "handle" this undesired event. If the same happens to a racing cyclist during the Tour de France, he replaces the bike. This is the "cheapest way" for him to handle such an event. Similarly, it must be left to the user of an operating system to decide whether he wants to wait until a defective device (e.g. disc) comes into operation again or to use another device (e.g. tape). The order of recovery actions and thus the order of degrees may be different for different users according to their goals and purposes.

Sometimes not only the ordering may be different, but some situations are not desirable at all. For example, storing a process on secondary storage and restarting it at a later point in time may not be sensible for some processes, either because the results produced by that process are useless after a certain point in time or because restarting that process may be cheaper.

10.4 Order of Actions. Even if all degrees lead to the same situation but use different methods with different costs to achieve this situation, the order of degrees may be different for each user. For some applications (weather prediction, air-traffic control) time is the most critical resource, and all other resources (storage, paper, etc.) are regarded as less expensive. A program for editing text files will view costs differently. Therefore the decision which degree should be tried first must be left to the user. He is the only one who knows which situations are less desirable and which actions less costly with respect to his task.

From this we may conclude that recovery from an UE requires cooperation of both levels: the user knows which situations are useful and which costs are acceptable; the programs of the lower level know how these situations may be achieved (e.g. how to store processes) and which costs will arise.

10.5 Possible Solutions. To achieve this cooperation, two basic concepts may be used: (1) Several different versions of a module /2/ are provided; the difference between these versions lies in their preparation for and recovery from UEs. For example, one version of a module for file management provides additional operations for copying a file and contains a mechanism for recording operations that change the contents of this file. A second version works without this mechanism and thus offers less security. It is up to the user to decide which version he wants to use. This possibility may be applied to modules that are used within large (operating) systems where a unique interface is not necessary.

(2) Another solution is to provide the recovery actions as operation of the abstract machine. The specification of these operations describes the requirements these operations try to fulfill and the costs that result from executing them. These specifications can be used by higher level programs to determine a response suitable for their purposes. This procedure does not exclude the alternative: some recovery operations are tried before the higher level is informed. This may be useful if costs for reporting an UE are higher than costs for trying that degree (e.g. repeating an I/O operation).

## 11. 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 UE 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 are 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.

## 12. Conclusions

We find it unfortunate that our conclusions are based on small scale systems. This limited experience supports the following conclusions:

1. Proper handling of UEs requires that a systematic approach to UE handling be taken in every part of the system. Most of the difficulties with UEs that we have experienced occurred because our "lowest" level, the commercial system that we were using, did not follow our approach.

2. Our ability to use abstract interfaces does not appear excessively restricted by the necessity of considering UEs in designing the abstraction.

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

4. The consideration of UE possibilities requires half and sometimes more than half of a designer's effort in writing specifications in our present efforts. In our own evaluation this is a reasonable price for the potentially increased reliability of the system.

5. The TRAP-function and the functions for resuming normal execution should form a separate module, the details of communication between levels are hidden from both levels. This module may be implemented differently for different levels. Thus additional facilities available at higher levels may be used.

6. Information determined by UE routines of one level and given to the user of this level support user's reaction to UEs. It is important, however, that this information is consistent with the abstraction defined by that level. Otherwise it will only be meaningful to users that know "hidden" information.

7. The "uses" hierarchy can be maintained even in case of UEs /8/.

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

### 13. Acknowledgement

We are grateful to P.J. Courtois, H.D. Wactlar, Dr. James S. Miller, A. Newell, A. Jones, J. King, and the other members of the Research Group Betriebssysteme I 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.

### 14. References

- /1/ Parnas, D.L., "A Technique for Software Modules Specification With Examples", CACM, May 1972.
- /2/ Parnas, D.L., "On the Criteria for Decomposing Systems into Modules", Carnegie-Mellon University Technical Report, 1971, CACM vol.15,12 (Dec. 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", Proc. of IFIP Congress 71, 1971.
- /5/ Wirth, N., "Programming by Stepwise Refinement", CACM vol.14,4 (April 1971).
- /6/ Parnas, D.L., "Some Conclusions from an Experiment in Software Engineering Techniques", Carnegie-Mellon University Technical Report, Proc. FJCC 1972, AFIPS Press.
- /7/ Jones, A., Private Communication
- /8/ Parnas, D.L., "Some Hypotheses About the "Uses" Hierarchy for Operating Systems", Forschungsbericht BS I 76/1, Technische Hochschule Darmstadt (Germany), March 76.
- /9/ Parnas, D.L., "On a Buzzword: Hierarchical Structure", Proc. of IFIP Congress 74, 1974, North-Holland Publ.Co.
- /10/ Parnas, D.L., "On the Design and Development of Program Families", Forschungsbericht BS I 75/2, Technische Hochschule Darmstadt (Germany), IEEE Transactions on Software Engineering, March 1976.
- /11/ Randell, B., "System Structure for Software Fault Tolerance", Proc. of 1975 Int. Conf. on Reliable Software, Los Angeles CA, April 1975.
- /12/ Kaiser, C., Krakowiak, S., "An Analysis of Some Run-Time Errors in an Operating System", Colloques IRIA, Rocquencourt, April 23-25, 1974.

### 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 EC41 or EC46 will occur. 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 does not occur, the value of EC5 should be defined. Trap EC6 only makes sense, if EC4 or EC5 need not be issued.

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 probable 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 reaction to 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 UEs, but it can force inefficiency when it is desired to delete a whole subtree. Were this to become a problem, we would add a distinct function to delete a whole subtree.

8. The three points marked "(8)" illustrate a difficulty in trying to make the traps completely predictable, yet not reveal the implementation. Our manipulation on SPSLFT makes the assumption that space for storing VAL is allocated when the node is created. Further, we assume that the space is limited by the number of nodes. In some implementations that would not be so. For example, the real limitation might be maximum depth. For those implementations the specifications will require traps 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 SVVAL, 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 SVVAL. 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 traps for the situations 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 traps 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 gave it as a reasonable but not necessarily optimal design.

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:

- (3) call EC4 if i<0 or i>p1  
(11) call EC5 if 'Exists'(i)=false  
call EC6(k) if failure

Function VALDEFD

possible values: true,false  
parameters: integer i  
initial values: VALDEFD(0)=false;  
all others undefined  
effect:

- (4) 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  
effect:

- (11) call EC10 if i<0 or i>p1  
call EC11 if 'Exists'(i)=false  
call EC12 if 'VALDEFD'(i)=false  
call EC13(k) if failure

Function ELS

possible values: true,false  
parameters: integer i  
initial values: ELS(0)=false;  
all others undefined  
effect:

- (11) call EC48 if i<0 or i>p1  
call EC14 if 'Exists'(i)=false  
call EC15(k) if failure

Function ERS

possible values: true,false  
parameters: integer i  
initial values: ERS(0)=false;  
all others undefined  
effect:

- (11) call EC49 if i<0 or i>p1  
call EC16 if 'Exists'(i)=false  
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

(11) call EC20 if 'ELS'(i)=false  
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

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  
'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 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  
'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 possible.
- 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.
6. If not stated otherwise, "continue" is always possible.

## Examples of Possible Error Degree Specifications

When the user of this tree module specifies just exactly what he wants implemented, he must specify not just the desired behavior when everything goes well, he must provide information about his preferences in case of mishap. He can do this by describing various degrees of damage, from his viewpoint. He might, for example, define 4 degrees as described below.

Degree 0: Through the use of extra data maintained by the module, the tree is reconstructed. The user is only aware of a delay.

Degree 1: The user is informed of the root of the deleted subtree. All functions are updated to indicate that the lost subtree does not exist.

Degree 2: The user is informed that some data has been lost, but not which subtree. Functions are updated as in degree 1.

Degree 3: The user is informed that some data has been lost. Functions are not updated and the user may find the tree in an "impossible" state. He may, however, continue to insert data.

Degree 4: The user is informed that he may make no further changes to the tree, that some data may have been lost, and that the tree may be in an impossible or inconsistent state.

Although this set of degrees appears reasonable, it certainly does not correspond to the wishes of every possible user. For some users with strict real time demands, the delay involved in degree 0 may be unacceptable, it may be preferable to lose the data and proceed as in degree 1. For other users, the cost of maintaining the redundant data necessary to satisfy the requirements stated in degree 1 may not be worth it, they may prefer degree 2.

The very fact that such differences in goals can exist emphasizes the need for a specification of the degrees. The behavior of the module when something goes wrong should not be left as a random result of implementation decisions made by a programmer. The programmer should be informed of the way that purchaser or user ranks damage so that the programmer may be guided by that information.

## APPENDIX 2

### Examples in Which UE Messages Must Be Passed Between Levels

It may not be obvious to some readers that UEs 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 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 UE 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 UE due to some hardware violation. If the UE 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 UE 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 UEs at the incorrect level would have failed due to lack of proper knowledge, the system would have a poorer reliability than necessary. The alternative solution is to introduce the necessary knowledge to the lower level. This clearly would reduce the advantage we have gained from the hierarchical structure and from decomposition into "information hiding" modules.