M.I.T. Laboratory for Computer Science
Computer Systems Research Division

June 22, 1977
Request for Comments No. 144

RESPONDING TO ERRORS IN A COMPUTER SYSTEM

by Harry C. Forsdick

The attached paper reflects thoughts I had in January 1976 about error recovery. Since that time, I have changed my mind on the feasibility of some of these ideas. For historical accuracy, we are publishing this paper now, even though my feelings about the subject have changed.

Responding to Errors in a Computer System

January 9, 1976

by Harry C. Forsdick

I. Introduction

The subject of this paper is errors in a computer system and the proper response by the system to them. The purpose of this work is to develop insights about the difficulties encountered in dealing with errors. Although it is not achieved here, the ultimate goal is to develop a system where normal, error-prone operations are augmented by the system's response to errors so that the combination equals the idealized operation of the system.

II. A Model of Errors in a Computer System

The action of a computer system can be viewed as a very complex function which operates by calling smaller, less complex functions. The arguments to the called function must obey certain input assertions. The calling function depends on the called function to perform the tasks associated with it -- the output assertion of the called function. Output assertions then become a set of axioms upon which the calling function is based.

With this view, an error in a function is a departure from either the set of axioms upon which the function is based or the assertions about the inputs to the function. If we assume that all parts of a system can be characterized as a function then this statement applies to both hardware and software.

For example, the machine instruction load register has a precise set of input and output assertions. Executions of the load register instruction can be viewed as calls on a function which alters the state of the machine. An error can occur in any function of a system, hardware or software. Thus it is consistent to view a parity error in the execution of a load register instriction in the same light as an access violation error in the execution of, say, the create segment function.

Besides the uniform view of operations that functions yield, a consistent view of data is necessary to lay the groundwork for a treatment for errors. Object oriented systems provide such a view: all data items in the system are assumed to possess certain common attributes like unique identification and type. The uniform treatment of data items in an object oriented system provides two desirable qualities with respect to error processing: accountability of data items and a common mechanism for naming data items. Accountability is necessary to determine the authority that should be informed when an error is detected in an object. A common mechanism for naming is needed for referring to objects in reporting errors.

So far, I have introduced two elements into an evolving model of errors: functions and data objects. Functions can call other functions supplying data objects as arguments. From the viewpoint of one function in a sequence of calls, there are five basic ways to describe how errors occur:

1. Its caller makes an error by passing it an inconsistent set of parameters. Input assertions are not satisfied.

2. An error has occurred which the callee cannot correct. This situation must be signalled to its caller. Output assertions are partially satisfied.

3. The called routine causes an error but does not detect it. Output assertions are not satisfied.

4. A routine called by the callee signals an error. Axioms are partially satisfied.

5. A routine called by the callee causes an error, does not detect it, but the error is detected by the callee. Axioms are not satisfied.

These five points interact as illustrated in figure 1. Function A calls function B supplying input arguments. Errors in the input argument (1) from A are detected, this effect cannot in general be corrected, so the occurrence of an error is signalled (2) by B to the calling routine A (4). Alternatively someone could have signalled to B that an error has occurred (4) but B could do nothing to correct the error and must signal an error (2). Similarly someone could have returned to B without performing its function correctly (5), B detects this but cannot correct the error and signals an error (2). Finally, B could have caused an error but not detected it (3) and return to A at which point A detects B's failure (4).

The model of system operation has been stated in terms of a single process executing function calls and returns, accepting input arguments and returning computed results. A generalization is to expand a function call into an interprocess signal or message, the input argument into the contents of the message and the return into an acknowledgement message. Many of the other ideas about errors carry over into this context, however error reporting seems to present a problem. Error reporting follows the routine dependencies induced by a call-return pattern. A different characterization of dependencies that the call-return model misses is asynchronous interprocess communication and references to shared objects. With these dependencies, reports of errors cannot necessarily be associated with returns. The essence of the difference is that in a call, the calling routine supplies the arguments and waits for the called routine to return and report on the outcome of the operation. In a message oriented scheme, the calling routine supplies the arguments (the message) to the called routine which runs in a separate process. The calling routine proceeds with its computation. Thus there is no implicit combination of convenient path and relevant point to return a report of an error. An additional departure from the call-return image of error reporting occurs when the error report is to be directed to a third party, an error handling routine. Both of these additions tend to force error reporting to be a much more explicit operation. Rather than being part of the call-return mechanism, the report of an error must a distinct operation.

Finally, a single error can take on multiple appearances, one appearance for each level that is concerned with the erroneous object. For example, consider a bank transaction system running as an application program on a Multics-like system. The data bases might be the ACCOUNTS file and the

AUDIT_TRIAL file. The ACCOUNTS file is composed of multiple ACCOUNT records and the AUDIT_TRAIL file is composed of multiple TRANSACTION records. Similarly, each ACCOUNT record and TRANSACTION record is composed of smaller parts. At the lowest level, an error occurring in an atomic object, like a single word in a paged virtual memory system, might be viewed as an erroneous bit in a word by the load register instruction. In the routine that is searching the ACCOUNTS file for accounts with negative balances, the same error would be viewed as an erroneous BALANCE record. By the routine that audits the ACCOUNTS file, the error would be viewed as a partially completed check for consistent accounts. Recovery from the error in the page might be possible by accessing another copy of the page. If this is impossible, one or more atomic objects might have to be declared to be in error -- for example, a BALANCE record which is part of an ACCOUNT record. If the balance record is not reconstructable, (perhaps from an AUDIT_TRAIL record) the ACCOUNT record might have to be declared to be in error, and so on.

The multiple views of a single error suggest two requirements of error processing: First, the semantics of detecting, reporting and correcting errors must be matched with the level of the view of the error. Second, a crucial task for an error processing scheme is to find the proper authorities for dealing with an error. These two points will receive further attention in the sections that follow.

The error recovery problem in its most general form is probably unsolvable. This is because the operations that are performed in processing errors are also based on axioms and input assertions which can become invalid. While it is appropriate to minimize the basis on which the fault tolerance mechanisms run, it is impossible to design a mechanism that is completely immune to errors. A uniform approach to error recovery can provide two services that come close to a complete solution: it can limit the probability of catastrophic failure and it can provide a means for accepting aid from outside the system when such catastrophic errors do occur.

III. Detecting Errors

Error detection is the act of discovering that part of the state of a system does not conform to one of the assertions about the system. The design of error detection algorithms should be based on the assertions associated with the objects in question. Using this approach, a balance can be achieved between the interpretation put on the object and the tests performed on the object. This idea supports the notion that a parity check on a single word of memory, a check sum on a collection of words in memory and a consistency check on a directory in a file system are just three points in a continuum of error detection algorithms; each check attaches a different level of semantics to logically different objects which are built out of the same raw data.

For any single function of a system there are many assertions that can be made about its operation. Consider the add_name function of a Multics-like system and possible Input and Output assertions:

add_name(dir_name, old_entry_name, new_entry_name)

Input Assertions:

1. dir_name is a legal path name and refers to an existing directory in the file system.

2. The principal of this process has modify access to the directory dir_name.

3. old_entry_name is a legal entry name and is the name of an object a segment or directory in dir_name.

4. new_entry_name is a legal entry name and is not the name of any existing object in dir_name.

5. There exists no entry name in dir_name equal to new_entry_name.

Output Assertions:

1. Referring to "dir_name">"new_entry_name" is identical to referring to "dir_name">"old_entry_name."

2. The directory dir_name is at most one page longer than it was before the operation.

3. No other logical changes are made to the directory dir_name.

4. The add_name operation will return to its call in f(dir_name) seconds.

There are a number of different algorithms that can be used to test the partial correctness of each of the input and output assertions. These algorithms vary in the degree to which they are coupled with the semantics of the assertion. An error detection algorithm that is highly coupled with the assertion it is testing would exactly ask whether or not the assertion is true. A loosely coupled algorithm would ask a question which is an approximation of the assertion. Of course the desired goal would be to have every assertion checked by a highly coupled algorithm. Quite often however, there is a direct relationship between the complexity of an assertion and the cost of the corresponding highly coupled check. Thus a goal is to design a system so that it is inexpensive to make highly coupled checks. For example, if path names are converted to unique identifiers (UIDs) and UIDs are used throughout to refer to segments, then converting both path names to UIDs and comparing yields an inexpensive highly coupled check of the first output assertion above. (1)

As examples of highly and loosely coupled checks, consider the last output assertion above: A loosely coupled check might be to put an upper limit of 10 seconds on the time to be taken by any single function in the entire system; after a function has been running for 10 seconds without returning, an error would be signalled by some time-out mechanism in the calling routine. This is a loosely coupled check because 10 seconds is being used an upper limit on all values of f(dir_name). A more highly coupled check might be to set the limit for add_name to 100 msecs (perhaps the actual upper limit of f(dir_name)) or even the

(1) I suspect that in any inherently cheap error detection algorithm, use is made of an unverified assumption. In this example, the unverified assumption is that the routine that converts path names to UIDs works correctly and will not be sensitive to possible mistakes committed by add_name.

value of some function g(dir_name) that only considered the number of pages in the directory when estimating the computation time for add_name. There are two points that come out of this discussion: error detection algorithms should be highly coupled with the assertions they are attempting to verify and systems should be designed so that at every level, highly coupled error detection algorithms are inexpensive to run.

The placement of error detection algorithms is critical to the efficient operation of a system. From the standpoint of the number of times an error check is performed, it is desirable to concentrate error checks in the higher levels of a system. This way, erroneous arguments are caught early and needless computations are avoided. In addition, the lower level routines that are vigorously exercised by high level routines are not incumbered by error detection logic. However, quite often it is more expensive to verify that an argument to a low level routine is correct at a high level than to perform the actual low level operation. In this situation it is more appropriate to pass an unverified argument to a lower level routine and then respond to an error report than to check and respond in the higher level routine. The various arithemetic data exceptions in most processors are examples of this reasoning. There is an essential conflict here between wanting to perform checks at high levels and not normally having the information available at these levels to perform the check.

Finally, there is the problem of undetected errors. This can happen in at least two ways. As with normal computations, an error detection computation can fail because of a failure in one of the axioms on which it is based. Failing in this manner is another reason for keeping error checks as simple as possible. In the second mode of failure, the error goes undetected because one consistent state of the system gets transferred erroneously into a second consistent state and the failure goes undetected. Here the problem is due to an insufficiency in the detection algorithm. For example, the assertion about add_name that "no other logical changes are made to the directory dir_name" may be very difficult to test. A routine which added not only the new name but an additional name of its own would leave the directory in a consistent state, but would not satisfy this assertion. This mode of failure is more likely than the first if function computation and error detection are independent since only one error must occur for the second mistake to occur while two must occur for the first. Undetected errors contribute to the unsolvability of the general error recovery problem.

IV. Reporting Errors

An error report is a communication from one function to another that an assertion has been violated and in addition that the function issuing the report is incapable of correcting the error. An error report causes a shift in the manner in which an error is viewed. When a function chooses to report an error, it does so because from its perspective, it cannot correct the error. If the calling function has a sufficiently wider view of the environment it may be able to correct the error. Otherwise, it passes on the error report, perhaps transforming the meaning of the report at the same time. There are five aspects of error reporting:

* The decision to produce the message -- error detection.
* The contents of the message.
* The method of transferring the message back to the calling program.
* The means by which the calling program receives the message.
* The manner of responding to the message -- error correction.

The middle three points are the subject of this section.

The nature of the mechanisms classified under these three topics is greatly influenced by the overall structures of the system. For example, the interconnectivity of the functions of the system can present restrictions on the method of transferring messages between functions: It is most natural to trace back the call chain, however it may be more appropriate to notify some authority that is not in the call chain.

In current practice, there is a wide range of techniques for expressing the contents of a message: one bit success-failure return values (true, false), error codes (an integer), named error codes or alternative exits (error_table$no_access), conditions (parity_error) or alternative exits which return a general information structure providing additional error description. Typical contents of an error report include: a complete description of the nature of the error, a list of the offending objects, a description of the authority claiming that the error exists, a description of what the reporter would like the report fielder to do in response to the error: verify, correct, log, reinvoke, etc. Efficiency issues will dictate that some elements of the report be more abbreviated and encoded than others. Vestiges of each of the points above should be discernable in any error message.

There are several alternative methods for transferring the message to the fielder: via a return to the calling routine, via a call to an error processing routine which is part of the set of routines that handle all objects of a given type or via a message between the reporting routine in one process and a handling routine in another. There are several attributes required of the mechanism to transfer error reports:

* The report should go to the proper authority, not always the caller of the routine that is making the error report.
* The transfer of the report should be integrated with, but distinguished from, the normal mode of transferring control from one function to another.

The techniques for receiving error reports and the methods for transferring them are closely related. Essential requirements of routines that handle error reports are:

* The routine must have a means of declaring the types of reports it is prepared to field and have meaningful default responses to reports that it does not anticipate.
* The presence of a handler should not interfere with normal computations.

There are a number of hindrances to error reporting that are due to the structure of a system. An unfortunate distinction is made between intraprocess and interprocess error reporting --

unfortunate because additional complexity is introduced by the differentiation. The distinction is made because interprocess boundaries put constraints on the address space of the discourse as well as the interaction of the reporter and the fielder. The requirement that reports be machine readable (i.e. possess a regular format and refer to objects by easily processed names) introduces message codings which tend to lose information and constrict the expression of messages.

V. Correcting Errors

Correcting an error is an attempt to remove the effect of the error and to resume the computation as if the error had never occurred. Error correction forces the objects of a system back into a state so that the assertions upon which the proof of correctness of the system is based are true again. In existing systems, where error recovery is not planned for, too much information must be fabricated to force the system back into a consistent state. When an error occurs, much of the system state is correct, but is ignored in favor of taking the system back to a minimal correct state and then rebuilding to a totally correct state. Examples of this are the Salvager system's role in recovering from Multics system crashes and the similar function of the Checker program in the Tenex system. Because so much correct information is ignored, these programs must check the entire file system for consistency rather than just the area affected by the error.

In a strictly layered system, the higher the level of a function, the higher the level of semantics associated with the data structures of that function. There is a direct correspondence between the level of a function and the complexity of the interpretation put upon the bits in its data structures. Similarly, the higher the level of a function, the easier it is to nullify the effect of an error. For example, consider several levels of a banking transaction system: the banking transaction application program, the virtual memory and the hardware memory. If an error occurs in one bit of the hardware memory system it seems easier for the virtual memory to recover a redundant copy of the page which holds the correct value of the erroneous word than for the memory to be able to correct all single errors. Similarly, it seems easier for the banking transaction program to attempt an alternative or reduced computation on a value that was inaccessible than it is for the virtual memory to recover any erroneous word in a writable page. This example illustrates an additional point: An essential characteristic of different levels cooperating in error recovery is that they should all demonstrate the same diligence in continuing the communication about errors that they are unable to repair. This also suggests that low level functions should only attempt correspondingly low level error recovery -- using techniques that are balanced with the interpretation they put on data structures. With higher level functions, higher level output assertions offer more latitude in the ways of satisfying the same computational need.

Redundancy, independence and an alternative computation are all integral aspects of error correction:

* Redundancy, because error correction essentially adds information that has been lost to a system. The added information cannot be created out of a vacuum -- it comes from redundant encodings of the state of the system.
* Independence, because for redundant information to be effective, it must be isolated from the same types of errors that effect the state information it is backing up.
* An alternative computation, because the function that was being performed when the error occurred must still be performed after the object with erroneous state has been corrected. In some cases, the same function can be reinvoked, in others, a different set of functions must be used because total correction is not possible.

The three aspects mentioned above and the assertion upon which the recovery is based interact in the following manner:

An error has been detected, either locally or in some other inferior function. If from an inferior function, then the error has been reported to this level through the reporting mechanism. The report indicates the nature of the error -- i.e., which objects are erroneous. Associated with each object there is an assertion of its correct state. From the redundant information stored about the object (in a manner that is isolated from the same error that affected the object), the state of the object is restored and an alternative computation is performed to achieve the same function that was declared to be in error before.

There are wide variations in the actual instances of each of these areas of error correction. To see the whole picture, it is instructive to consider the range of techniques that can be used to achieve redundancy and independence; the techniques used in alternative computations tend to be directly related to redundancy and independence. Redundancy can range from one bit encodings of the state of an object, to a partial replication of an object's state, perhaps in some other organization useful for alternative accessing methods, to complete replication of the state of an object. In addition to replication of state, there is replication of the functions that manipulate state. This distinction recognizes that a single algorithm that causes errors will misperform, no matter how correct its inputs. Redundancy in the state of an object is useless without alternative functions for performing transformations on damaged objects. For example, doubly linked circular lists are quite often cited as robust data structures. If one link is broken, it is always possible to follow links in the other direction around the circle to the desired element. (1) In practice, however, even though these fields are included in the state of objects, rarely are there

(1) Of course this is not the only intended use of doubly linked lists; in general, they offer easier bidirectional accessing to neighboring elements.

alternative functions for making use of this increased robustness and accessibility.

Explicit instances of independent redundant information are rare in current systems; more commonly, independence falls out of a design that is motivated by other goals. The range of instances of independence spans the scope of common susceptibility to errors. Disassociation can be derived from several different techniques:

* Using different storage organizations or algorithms that are vulnerable to different forms of errors (array storage verses list storage).
* Using distributed logical and physical resources that are isolated from common malfunctions (separate memory modules, separate file systems, separate computer systems for applications built out of multiple processes).

While total recovery is desirable, it is not always possible. If it is determined that the effect of an error cannot be repaired by a function, certain minimal output assertions must be insured to be true. In this view, the output assertion of a function is really a conjunction of several separate assertions, some of which must always be true in all circumstances when the function returns. Maintaining the consistency of the internal data bases of a function and isolating or pruning away the effect of an error are all examples of satisfying certain minimal output assertions. For example, the function that moves a logical page of information from a core block to a disk block in a virtual memory system, core_to_disk, might have the following (partial) specification:

core_to_disk(page_ptr)

input assertion:

1. page_ptr points to the first word of the page to be moved.

Output assertions:

1. The contents of the data in the page pointed to by page_ptr will be moved to a disk block.
2. The page table word for the page will be transformed so that the next reference to the page will cause a page fault.
3. The record of the core block will be moved from the used list to the free list.

In addition, there might be the following constraints put on the satisfaction of the output assertions: if it is impossible to satisfy output assertion 1 then the actions described by output assertions 2 and 3 will also not be performed; if output assertion 1 is performed but 2 is not, then 3 will also not be performed. This constraint is motivated by security considerations; there are other system properties that could generate additional minimal assertions.

With any system that is composed of separately programmed modules, there is always the possibility that a calling routine may not have anticipated the report of a certain class of errors. Since there is an interpretation associated with every error report, receiving an unanticipated report must be treated with special care. It might be appropriate to convert the unknown report into a report with the meaning "function not performed, state untransformed." However, this is not always true in situations where the output assertions are partially satisfied and a partial transformation of state has been made. With this consideration in mind, it becomes desirable to have part of an error report specified by convention so that unknown reports can be analyzed for appropriate recovery measures. An unknown report that has the standard attribute "state unchanged" can be translated into a report with the semantics "function not performed" -- a report with reduced, but still useful meaning. (1)

Finally, it should be noted that when attempts are made to recover dynamically from errors, care must be taken in repairing inconsistencies in the system. This is because in repairing after an error, it is quite possible that the tracks of a much more serious error have been covered. This requires that error reports must be well understood before attempts are made to correct a possibly inconsistent state. For example, if a pointer is determined to be illegal, not only must a pointer to a legal object be found, but also the object which the good pointer originally referenced must be found -- a much more difficult task.

(1) In addition, leaving an unchanged state on an error return appears to be a desirable attribute of a function.

VI. A Systematic Approach to Error Processing

So far, I have presented views of three rather distinct aspects of error processing: error detection, error reporting and error correction. It is now appropriate to consider in more detail how these three parts should fit together in a systematic manner. The ability of a system to tolerate errors is very much like the ability of a system to authenticate all accesses to objects: each of the actions (errors and accesses) can occur at any level of the system. At different levels, different semantics are associated with the actions. Thus, at all levels of a system, there must be some logical steps devoted to tolerating errors and authenticating accesses -- each of these tasks must be distributed over the entire system.

Figure 2 illustrates the interaction between two levels of a system. In this view of error processing, each level's functionality besides including the desired logical operations, also includes the error detection, reporting and correction logic. Error processing starts with one level's error detection algorithm discovering that an assertion has been violated. A decision is made in the detection logic whether to correct the error or to report the error to a higher level authority. If the error is corrected, processing will continue at the same level as if the error had not occurred at all. If the error cannot be corrected by the current level, the detection logic causes the error reporting logic to deliver a report of the error to the proper authority. At the next level higher, reports of failures from lower levels are accepted by the error detection logic. Decisions are again made whether to attempt recovery or to send the report up to a higher level.

When an error occurs while the error processing logic is attempting to recover from an error, there are several alternatives that can be taken. It is possible to consider each of the aspects of error processing as a function itself with the ability to detect and correct errors in its own operation. Thus, if an error occurred while reporting the existence of another error, attempts would be made to correct the latter failure so that the original error could be reported. There are problems with the recursive nature of this treatment. A more practical approach is to stop the system when an error occurs while another error is being processed and have an external force correct the second error. Then the system is restarted at the point where the error occurred in the error recovery logic. Stopping the system is actually an error report to a system authority that is responsible for the global state of the system.

Finally, there are a number of design principles that I feel are crucial to systematic error processing. These points have been derived from developments in the rest of the paper.

* Assertion Based -- All efforts at processing errors should be based on well stated assertions about the functions that are being made to tolerate errors. Error processing logic will appear at the part of the system where it has the most relevance. This guideline discourages haphazard placement of

unjustified system interconnections and thus enhances system modularity.

* Semantically Balanced -- Error checking and correcting should be semantically matched. Transitions between levels of semantics should be done through the error reporting mechanism.
* Completeness -- A systematic approach to error processing should apply to all levels and aspects of a system; this includes the error processing mechanism itself. Errors occurring in the error recovery mechanism are viewed as catastrophic errors that need a special form of recovery, for example aid from an outside agent. An outside agent should only fix the most recent problem and then return control to the error recovery mechanism. Most error recovery will occur within the system itself rather than in some alternative system that may not be as diligent in upholding the goals of the main system. Completeness also enhances the ability of the error processing mechanisms to extend out to application programs whose error recovery requirements are not necessarily known to the inner system.
* Accountability of State -- Each part of a system's state belongs to an authority that is responsible for the integrity of that piece of state. The pieces of state can be thought of as objects and the authorities as object managers. The

authority of an object must be easily identifiable. An idea that has been suggested, but not developed here is that error reports must not only be transmitted from level to level, but also from authority to authority.

* Independent Redundancy -- An essential aspect of error correction is redundancy. For a redundant encoding of an object's state to be of use, it must be both stored in memory that is isolated from the memory of the original stored state and encoded in a manner that is functionally orthogonal to the original state.
* Flexible Bindings -- The fragility of objects usually can be traced to breaks in bindings between various generalities of names. Functions should be written so that bindings that are necessary for correct operation are not so tight that they are violated because of the smallest of errors; in addition, for every binding that is made, the information that was used to establish the binding should be saved as redundant versions of the binding.
* Error Reporting Matched to Shape of System -- Detecting and correcting errors, when confined to a single function, is a relatively easily comprehended task. Complexity arises when an error cannot be fixed at one level and must be reported to another function. To handle all of the interrelationships that arise, an error reporting mechanism must be able to back track all forward paths of communication in the system. This not only includes call-return communication channels, but also interprocess messages and general shared data bases.

VII. Further Work

This paper has presented an approach to processing errors in a computer system. A number of ideas have been presented -- some in sketchy detail. Further work needs to be done at least in the following areas:

* A Non-Hierarchic View. Many of the arguments in this paper were presented with the view that the only communication that occurs in a system is hierarchically structured. In real systems, there are several other forms of communication: interprocess communication and shared data bases, for example. Forms of error reporting that reverse the direction of these forms of communication are needed; the strictly hierarchic arguments of this paper need to be rethought.
* Authorities of System State. More development on the idea of framing error reporting around authorities is needed. Just as there is a hierarchic structure induced by the patterns of calls and returns on functions, there might be a structure present between several interrelated authorities that do not explicitly call each other, but rather share a common data base.
* Assertions. I have been vague and loose in my characterization of assertions. Further work on the expression (form) of assertions and techniques for generating

assertions (content) is needed. An assertion must be abstract enough to serve as a brief description of the function of a module as well as detailed enough so that appropriate tests can be constructed to verify that the function has been performed correctly.

* Inexpensive Error Detection Algorithms. The one aspect of error processing that enters into the normal path of computations is the detection of errors. Cheap methods of error detection algorithms that are highly coupled with the assertions they are verifying should be investigated. The other two areas, error reporting and correcting, are used infrequently enough so that less optimized computations will suffice.
* Example Applications. The ideas suggested in this paper should be tried out on several example applications. My hope is that a unified approach to error processing would apply equally well to parts of a centralized system like the Traffic Controller in Multics as to parts of a distributed system like a network file system or communication channel protocol.
* Entendability. The error recovery requirements of application programs should be explored. In addition, there are probably some restrictions that must be enforced to insure that application programs that are unverified with respect to maintaining the integrity of error recovery.

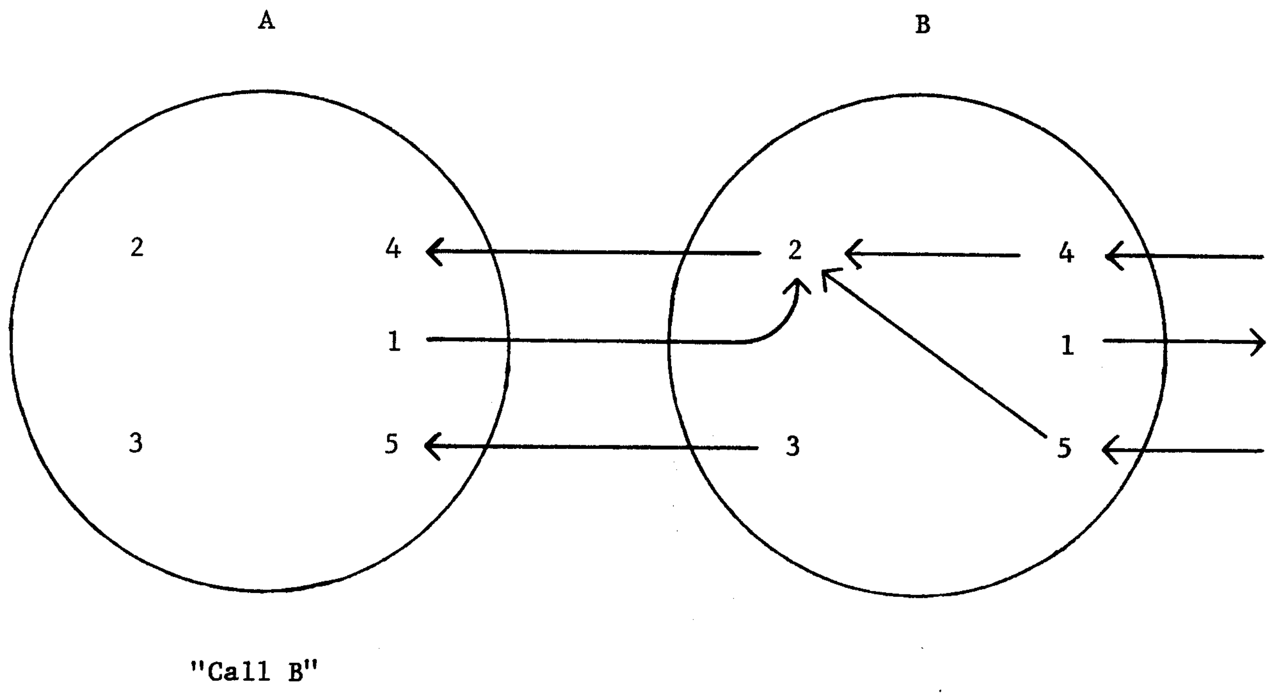

Figure 1
Patterns of Errors

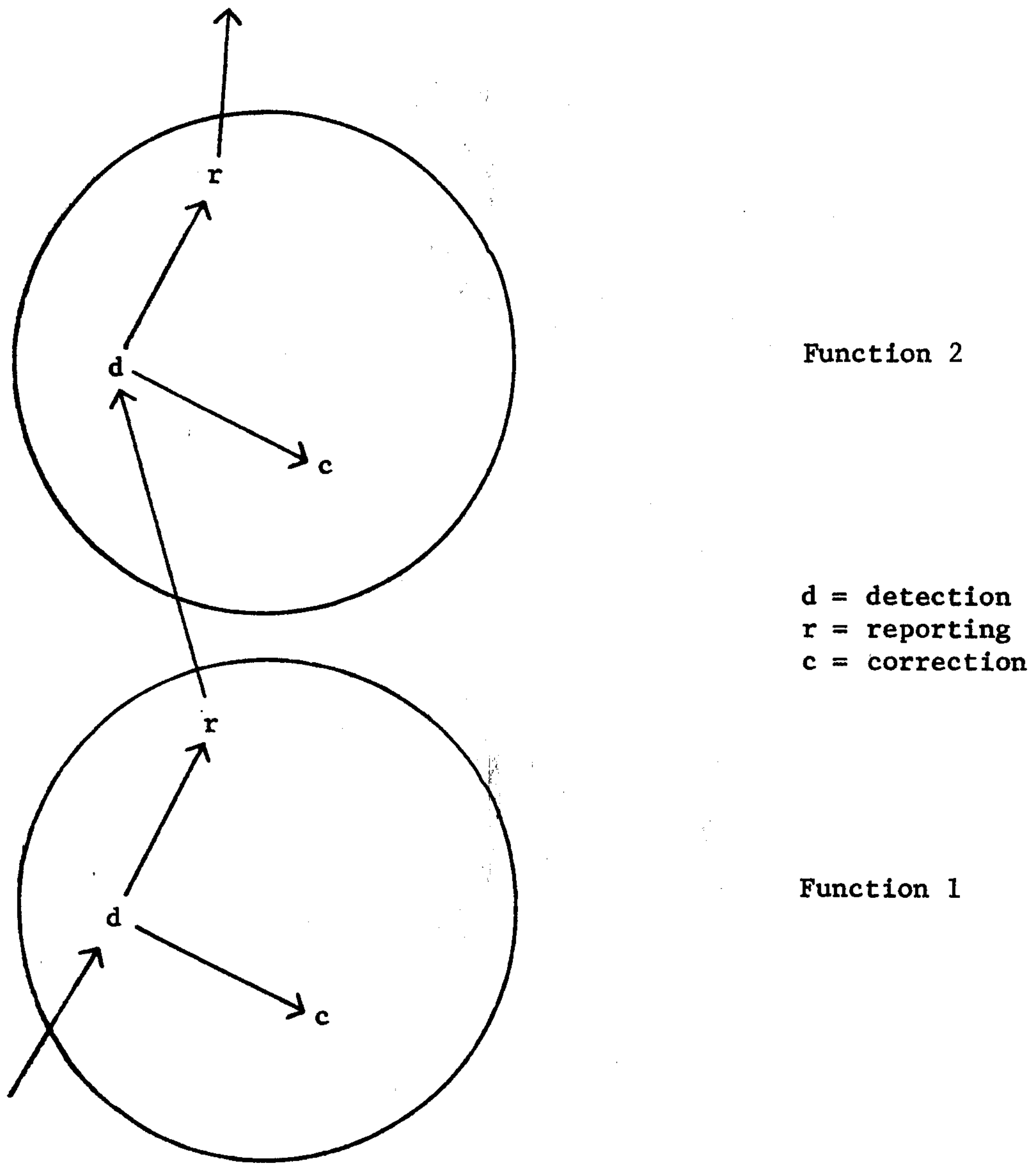

Figure 2: Interaction of Error Detection, Reporting and Correction