RELIABILITY IN TERMS OF PREDICTABILITY

M. Hamilton and S. Zeldin

Higher Order Software, Inc.
Cambridge, Massachusetts

ABSTRACT

Issues of reliability include identification of errors in relation to a particular object system. Experience in large system developments indicates that objective error identification is not a simple procedure. Particular aspects of error identification and analysis for a large, complex, real-time system are addressed. In addition, some measures taken, as a result of this experience, to address these issues of reliability with respect to system definition are discussed.

I. INTRODUCTION

In order to talk about reliability, we need to talk about systems and errors and systems which have errors or systems which do not have errors. For a reliable system is a predictable system; it does exactly what it is intended to do. Just how reliable a system is depends on the extent to which that system is error-free. But, what is a system? What is an error? And what does it mean for a system to have an error?

The struggle for answers to these questions is not just a mere exercise in a philosophical discourse, but rather the outcome of such an exercise could have far-reaching practical implications in a typical large system development process. This paper discusses the implications we experienced with a large system development as a result of having gone through such an exercise.

Consider a system as an assemblage of objects united by some form of regular interaction or interdependence; a system itself could be an object within another system. (For example, a software system, a hardware system, and a man-in-the-loop are not only systems, as individuals; but they could also be objects with respect to an overall system within which the individual systems all reside.) The very mechanism by which these objects are united could determine just how reliable a particular system is.

A particular system may be considered from many possible viewpoints (or development layers). If, for example, one is concerned with a definition of a system, it is viewed with respect to what it is supposed to do. If one is concerned with a description of a system, it is viewed with respect to whether or not the definition is effectively conveyed. If one is concerned with the implementation of a system, it is viewed with respect to whether or not the system is constructed to do what it is supposed to do. If one is concerned with the execution of a system, it is viewed with respect to whether or not the system does what is is supposed to do. Whereas the description and implementation layers of a system represent static views, the definition and execution layers of a system represent dynamic views.

Just as software, as a system, could be considered as an object in a larger system, each viewpoint of the software system, as a system, could be considered as an object with respect to the overall development process of the software system as a system. The important point to emphasize is that a correct focus is necessary with respect to both a component within a given system and a particular developmental viewpoint of that component before that object as a system can be discussed in terms of its reliability. If it is not possible to properly identify the object in question, we could be discussing different objects when we think we are talking about the same ones, or discussing a particular object with an incorrect assessment of the information concerning that object. Thus, well-intentioned changes for the sake of reliability could not only do nothing, but, in fact, they could make things worse. A case in point is the often misunderstood set of objects called software systems.

Basically, software has to be defined before the reliability of a particular software system is determined. If, for example, software is considered to include inputs from everything in the system within which a computer program is to reside, the reliability will no doubt be different than if the software is considered to be only the computer program itself (i.e., a set of instructions encoded in some manner for a particular computer). Or, if software is considered to include the specification of the computer program, the reliability of the software will be, no doubt, different than if the software is considered to include only the implementation of the specification (i.e., the computer program). Once having determined the object in question, from a software standpoint, it is then necessary to understand conceptually the concepts of a software error and the software having an error. Such was the case on the Apollo project, where the issues of reliability were of uppermost concern[1].

The questions and corresponding answers which follow summarize our attempt to understand how reliable the on-board "software" was after it had flown on all of its "real simulations."

II. RELIABILITY ISSUES: QUESTIONS AND ANSWERS

Question: What is a software error?
Answer: A software error was an unintended phenomenon in either the specification for a computer program or the computer program itself. There were

two major kinds of errors. One kind was determined by analyzing a system (or a set of sub-systems) on a stand-alone basis. For example, if the specification had an inconsistency between its functions or if the computer program had a data conflict, such errors could be found by analyzing only the system in question. Certain checks were made for completeness, consistency, and non-redundancy with respect to a self-contained system. At that time, there was no method to guarantee this type of interface correctness. The other kind of error was determined by checking one development layer with the layer from which it evolved. Thus, for example, a computer program was compared to its specification; the specification was compared to the designer's intent. It was possible to check for inconsistencies and incompleteness between, say, a specification and a computer program although we did not have methods which guaranteed traceability between layers. And, of course, we could not guarantee that a specification was consistent or complete with respect to the original intent of the designer. (Thirteen percent of the errors during development fell into this category, while 75 percent of the development errors were due to logical incompleteness or inconsistency[2].) Today, we support the solution to the "13 percent problem" by providing techniques which are intended to eliminate other sources of problems.

How do you determine if an unintended phenomenon is an error from a practical standpoint, and once having done so, how do you weigh the seriousness of the error in the context of overall reliability?
There were catastrophic errors which could have aborted a flight, errors which were worrisome to the success of a flight, and errors which were merely an annoyance. Other problems were officially recorded as "funny little things." They weren't problems per se, but yet, no one quite understood why they happened, since they didn't make sense in the context of known facts about the specification or the computer program. Sometimes there were borderline cases in that the difference in definition or interpretation of the software could determine whether or not a potential problem was a real error. The determination of these categories was often left up to engineering judgement; others, however, were more obvious and could be explicitly determined.

If the computer program "doesn't work" because the specification is wrong, is the computer program unreliable?
The software was unreliable if the software specification was unreliable. With respect to the specification, however, the computer program was determined to be reliable.

If an error is found in the computer program during development and a decision is made not to fix the error but to use a "workaround" (i.e., something that either prevented its happening or eased or removed its effect) to compensate for the error, is there an error in the specification with the new recorded anomaly and workaround, or does the recording change the specification? Is there an error in the computer program? If the computer program is in execution and the error in question takes place due to the error of ignoring the workaround, what system does this error reside in?
Even though officially errors were recorded as anomalies, unofficially the recording of an error with a workaround, rather than fixing the error, was an update to the specification. Such an update removed, in essence, the anomaly in question. Thus, if the error took place in flight with a workaround, this was not an error to be recorded, since not only was it officially recognized before flight, but it was also sanctioned to fly with special provision. If, however, the error took place and the workaround was ignored, the software system itself was not considered to have an error, but rather the user, who ignored the workaround, was considered to be in error.

If an error is officially known before flight (that is, it is recorded with the specification), but an official decision is made not to fix it or not to provide a workaround, and it occurs during flight, is it a software error?
Such a phenomenon would not have been recorded as an error, since a decision was made to officially sanction its existence in the software. Thus, it became part of the specification.

If an algorithm in a specification is incorrect, but the algorithm in the corresponding computer program is correct, is the computer program in error? Is the specification in error? Or is only the designer, who created the specification in the first place, in error?
Technically speaking, the computer program would have been in error. Practically speaking, if it was feasible, the specification would have been updated to conform with the computer program. The specification was in error if the algorithm could be determined to be incorrect, on a stand-alone basis, within the specification. Otherwise, the error was the designer's, alone.

If an error is officially known before flight and a decision is made not to fix it, but the computer program is fixed anyway, is this an error in the computer program?
To be consistent, the computer program was in error, because it was inconsistent with the specification. But an after-the-fact "fix" was sometimes considered better than an already sanctioned anomaly. Other times, fixes were frowned upon since last-minute attempts to fix an error often either didn't fix the program or made it worse. The resolution of the dilemma of a "good" fix was left to management, who either happily adjusted the specification by cancelling the anomaly or became angry at the program designer for his misguided good intentions, and forced the designer to remove the illegal fix.

Sometimes specifications are provided in the form of official documentation. Often, however, an implementation is based upon well-known assumptions that cannot be found in writing anywhere. Is it an error if the implicit information is followed? What if it is not followed?
Early in the project, implicit information made up the major part of a specification; it was considered an error if it was not followed. Later, if there was both implicit information and written information on the same subject, the written information overrode the word-of-mouth information, usually.

If there is an error in the input to the computer program, is the data considered part of the computer program when the reliability of that program is being measured?
If it was assumed that the final responsibility of loading correct inputs to the computer program was designated to either the mission-control engineers, the astronauts, or the hardware, then the problems resulting from wrong inputs (either instructions or data) were not software errors. The exception to this rule was if the computer program was required to allow and filter bad inputs.

If certain areas of the computer program are secure from the user and a lock mechanism prevents him from changing the program to fix an error during flight, is the philosophy of having a lock mechanism in error?
Certainly, the error that was locked out from being fixed was in error.

Conversely, if secure mechanisms are not implemented and the user inadvertently causes an error because he is not locked out, is the philosophy of having a non-lock mechanism in error or is the user in error--or both?
An extreme case of bad inputs was when an operator (like an astronaut) of the computer program inadvertently changed erasable* instructions incorrectly. Certainly, in this case, the user was responsible for making an error. It was not clear if the non-lock mechanism approach was in error. As long as there was an available mechanism to change erasables and to send control to these erasables, catastrophies were possible. Yet, the computer program could not be held responsible. One could draw the conclusion that such dangerous practices should never be allowed. Yet, if there had been a real error in the software, this mechanism could have saved a flight. (And, in fact, this very mechanism did save a flight in the case of an error from a system interfacing with the software.)

How can reliability be defined until the philosophy of error detection and recovery is defined? What is the relationship between reliability and error detection and recovery? Should the specification determine whether or not error detection and recovery should exist at all or is this the responsibility of the computer program? If the specification is responsible, should the specification include approaches for error detection and recovery?
In the early days of the project, many software engineers lobbied to incorporate more error checking procedures into the software. For example, it was considered by some that all input data should be checked and refused if it was not within predetermined limits. The astronauts and other users considered such measures unnecessary, since they assumed that the astronauts would know the procedures so well that there was zero chance for error. Unfortunately, understanding the software was not enough to prevent human errors. The more conservative philosophy was later vindicated when, on actual flights, not only the astronauts but others made real-time mistakes. The "no-error checking" philosophy accounted for the majority of recorded system errors. Of course, there were tradeoffs to consider; the more error logic incorporated in the software, the more chance for errors from the additional software. Thus, a careful analysis was made to determine just where the critical error detection logic should be incorporated. (Later we devised techniques which addressed problems on a system-wide basis and a lot of prima donna logic was no longer necessary.) Input errors, which caused problems in the software, were considered as software errors only if the specification provided for the software to protect itself against these errors.

If two errors cancel each other, is there an error?
No. However, if a subsystem was active with only one of the errors, that error was counted.

If an error is not detected during flight, but it is determined that it should have occurred, is it to be recorded as part of the errors which occurred on that particular flight?
Unresolved.

Is "better the enemy of good" in providing for protection against errors?
When adding redundancy or backup configurations, there were more tradeoffs to consider. There was a time when we were considering systems for the astronauts that were so error-proof that we had finally created the perfect system for preventing bad inputs. There was one flaw in the system however. We had not only prevented the astronaut from making any errors, we had prevented him from doing anything. There were also backup systems that could not prevent errors any better than the primary system (i.e., generic errors). A wrong specification for a primary and secondary system, for example, could result in two incorrect computer programs. (Resources are often compromised in the development of a primary system for the development of a backup system. This could jeopardize the reliability of the primary system. Given a choice, it is better, for example, to have a primary system that is 99 percent reliable over a primary system that is 50 percent reliable and a secondary system that is 50 percent reliable; for in the latter case, safety could be compromised for safety's sake.)

If there is a "problem" in the computer program and it is not clear if it originated in the specification, to which system is the error attributed?
Here, the specific source of the error would be recorded as unknown (i.e., with respect to either the specification or the computer program). However, an error would have been recorded for the software.

When more than one specification exists and they conflict with each other, which is in error?
Once, there was a question as to whether the computer program interface specification or the simulator program specification was correct. The flight program aborted during a simulation by one of the independent verification contractors. It was determined by several experts that the simulator should be fixed so that the flight program would not abort. The simulator was fixed, the flight program then worked beautifully during the next simulation, but it aborted in a real unmanned flight.

*Most of the instructions in the computer program were hard-wired. But there was a small set of instructions that was not. This set was called erasable memory (as opposed to fixed memory).

This is a good example of a generic error being introduced into a system which had preflight developmental backup capability. As to which specification was in error, when there is a conflict, there is no easy answer until careful analysis is performed; and even then, the existence of an error must be decided upon in terms of the initial intent of the designer.

If several errors take place and they are later discovered to be caused by a root source, are all of these errors recorded in the determination of the reliability of the system?

No. Errors, other than the root source errors, were only recorded for descriptive purposes in the analysis of the root error. There were times, however, that it was not clear, until much later, that certain errors were connected. It is also no doubt true that there were errors which were connected that were never found.

If an error took place in the computer program during flight, and a system error and detection mechanism took care of the effect of that error, would this have been a recorded error?

Yes. Sometimes the same mechanism that would have taken care of a software problem would also resolve problems from other systems. In one case where a human error was found and recovered by the software, the software was blamed because it reported the bad news![3] Although there were no recorded software errors during flight in the on-board flight software, there were several overall system errors within which the software resided. Many of these errors affected the software, but, fortunately, the software was able to recover or to be recovered in every case.

III. OBSERVATIONS

Of these overall system errors, 73 percent were real-time human errors, 92 percent were recoverable by using software, 40 percent were known about ahead of time but the workaround was inadvertently not used, and 100 percent could have been prevented by a more encompassing philosophy and tools to support it.

The analysis of the Apollo system problems during actual flights pointed out that the philosophy of how one goes about defining system relationships cannot be overemphasized. The type of relationships can determine not only how reliable the interfaces are, but also how flexible a system is in order to make new decisions with fast response times, both during development and in real time. The need for this type of flexibility was often crucial during off-nominal mission conditions.

IV. QUALITY OF SPECIFICATION

The difficulty in measuring the reliability of software is, in part, because of an inability to adequately define what it is a system is intended to do. To measure something is to determine the quantity of that something by comparing it to some standard or unit. The problem of measuring a software system is well stated by Manna and Waldinger[4]:

> To determine whether a program is correct, we must have some way of specifying what it is intended to do; we cannot speak of the correctness of a program in isolation, but only of its correctness with respect to some specification.

With respect to their own work, Avizienis and Chen[5] stress the importance of a reliable software specification as a standard unit of measure for developing equivalent algorithms.

If to measure the reliability of a software system we need an adequate specification, we are faced with the problem of measuring the reliability of the specification itself. In this case, it is impossible to completely measure the intent of a designer; we can at least measure logical completeness and consistenty, i.e., quality assurance.

Reliability, in the sense of quality assurance, is addressed in the specification language, AXES[6], by providing mechanisms to define data types (in order to identify objects); functions (in order to relate objects of types); and structures (in order to relate functions). The aim is to be able to define a system so that we can automatically check interface specifications statically. The foundations of AXES are based on a set of control axioms derived from empirical data of large systems[2]. (Fig. 1 illustrates this evolvement by the connecting dashed lines.) Each axiom describes a relation of immediate domination with respect to a functional system. We call the union of these relations control. From these axioms, a set of three primitive control structures have been derived[7]. These three control structures identify control schemas on sets of objects. From the assumption that we can identify an object or a set of objects, a mechanism for defining an algebra for each distinct set of objects is provided in AXES. Each algebra takes the form of a set of axioms that relate operations applied to objects of a type.

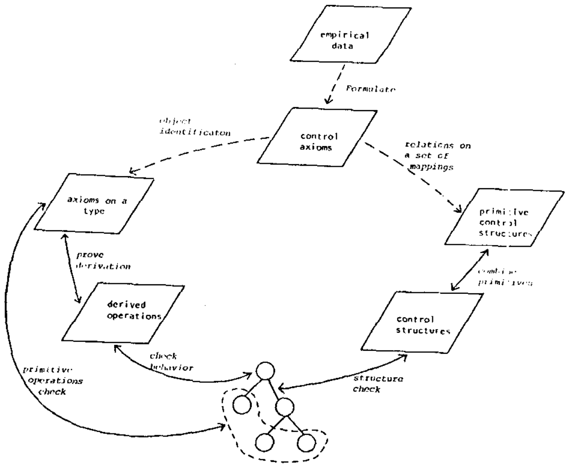

Fig. 1: Quality Assurance Checks with Respect to AXES

To form a system, defined as a function, control structures are defined in terms of the primitive structures; operations are defined either implicitly by deriving them mathematically from the axioms on a type or explicitly in terms of control structures using already defined operations on a type. When an operation is defined both implicitly and explicitly, we can crosscheck the intent of the specification for correctness (Fig. 1).

AXES uses the functional notation

$$y = f(x) \qquad (1)$$

where x is the input, y is the output, and f is the operation applied to x to produce y.

In attempting to define a system as a function, we already have incorporated an element of reliability in that we assert that for every value of "x" we expect to produce one and only one value for "y". That is, we expect the system to predictably produce the same result each time we apply f to a particular value.

Now, we must incorporate into our definition a means to identify all of the acceptable inputs and outputs and a means to describe the relationship between the inputs and outputs. In AXES, each input and output value is associated with a particular set of values. Each particular set of values, called a data type, is defined by means of an algebra. The syntax for defining each algebra is similar to that used by Guttag[8], but the semantics associated with each algebra is similar to the concepts described by Hoare[9]. The semantics for our algebras assumes the existence of objects. That is, when we define a system, as in (1), we assume the values of x and y to exist, and that when f is applied to x, y corresponds to the value x.

In many systems, especially large ones, it is often not readily apparent which input values correspond to the system's intended function until the system is decomposed into smaller pieces. Although we start with a large set of "seemingly" acceptable values, a predictive system must be able to identify "truly" acceptable inputs or to produce an indication that a particular function will not be able to perform its intended function. To identify a system's intended function, we make use of a distinguished value which we call Reject. This distinguished value is a member of each data type (Fig.2). If an input value corresponds to the value Reject, as an output, then the function applied to that input is said to have detected an error. A function applied to an input value of which Reject is a component (e.g., the value (1, 3, Reject)) may either assign Reject as an output value, or may "recover" from the error by assigning an output value other than Reject.

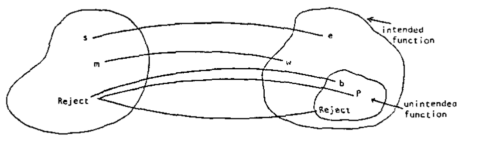

Fig. 2: Acceptable vs Intended Values of a Function

Once we have identified all acceptable inputs and outputs of our system, we need a means to describe the relationship between the input and output, sometimes called the performance of the function. We can relate input to output by asserting relationships about our system in terms of already defined operations and already defined relationships on a set of operations. Relations on a set of operations give rise to a hierarchical system structure. This means that when our definition is complete, the structure of our system will look something like the structure appearing in Fig. 3.

At each node in our hierarchy we shall put a function with the intent that at any level of our hierarchy (a level is a set of immediate dominated nodes with respect to a particular node, sometimes called a step of refinement), we can relate the functions at that level to the function at the node immediately dominating them.

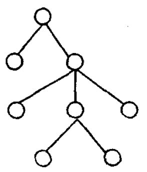

Fig. 3: Hierarchical System Structure

We need a set of rules to determine a level, and a set of rules to determine whether we want to create a level (or stop decomposing).

To determine a level, we want all the functions at the nodes of a level to be necessary and sufficient to replace the function at the node directly controlling these functions (Fig. 4). This will assure us that we will get no more or no less than we want, i.e., that our level is logically complete.

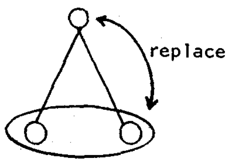

Fig. 4: Level Completeness

As we continue to build our hierarchy, each level completely replacing the function at the node directly above it, we must be able to define each point at which we want to stop. We stop when we reach a function whose behavior, i.e., its input and output relation, has been defined in terms of other operations on a defined type, and our specification is complete when we determine each stopping point. Now, if we know the behavior of each function at a bottom level and how it relates to the other functions at that same level, we know the behavior of the node directly above it. And with the same reasoning, we then know the behavior of the functions at each level successively closer to the root, or top node. And with the same reasoning, we end up with knowing the behavior of the root function itself. Thus, the behavior of the top node is ultimately determined by the behavior of the collective set of bottom nodes (Fig. 5).

Now we also want to assure logical consistency for a level. Since our intent, in the end, is to understand the behavior of the function at the top node, every time we talk about a value of that function we want to assure outselves that we are talking about the same value at the level directly dominated by that function; that is, we want to be able to

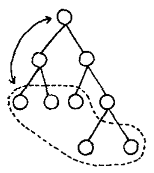
Fig. 5: Endpoint Completeness

determine which values match up with which functions. To talk about a value we use its name, or variable. We want to be consistent about input variables (Fig. 6) and output variables (Fig. 7).

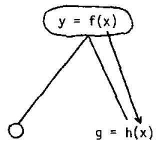

Fig. 6: Tracing Input Names

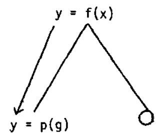

Fig. 7: Tracing Output Names

To avoid specification errors in naming values, a particular name is always associated with the same value as we travel down the hierarchy.

If our function is intended to be executed on a computer, we want to be able to determine which functions are more important than others. As we travel down the hierarchy, a function is always more important than the functions at the level dominated by that function, and at a particular level each function is assigned an importance with respect to each other function at that level (Fig. 8). Among other things, we can use this information to implement specific timing relationships, both relative and absolute, without conflict.

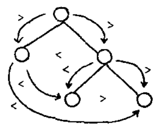
Fig. 8: Complete Ordering Relationships
*(">" means "more important than"
"<" means "less important than")*

When a system is defined incorporating the aspects of reliability, illustrated in Figures 1 to 8, we can limit the complexity of interface definition among systems. Furthermore, interface consistency and completeness can be checked statically by comparing the use of certain system structures with their definitions.

V. SUMMARY

A system is defined in terms of its overall environment and with respect to its position within its overall development process. An error is an unintended phenomenon. A system can have an error on either a self-contained basis (i.e., with respect to its overall environment) or with respect to its position in the development process. Once we understand the system in question and what it means for that system to be predictable, we are then able to concentrate on methods which will address both the errors on a self-contained basis or those which result from a developmental evolvement process. It is our contention that once a system is able to be defined consistently and completely on a self-contained basis, then we are able to evolve from such a definition to a next layer which is also logically consistent and complete.

REFERENCES

1. Hamilton, M. "First draft of a report on the analysis of APOLLO system problems during flight," Shuttle Management Note 14. Charles Stark Draper Laboratory, Inc., Cambridge, MA, Oct. 1972.

2. Hamilton, M. and S. Zeldin. "Higher Order Software--A Methodology for Defining Software." IEEE Trans. on Software Engineering, Vol. SE-2, No. 1, Sept. 1975.

3. Hamilton, M. Letter to the Editor. Datamation, March 1, 1971.

4. Manna, Z. and R. Waldinger. "The Logic of Computer Programming," IEEE Trans. on Software Engineering, Vol. SE-4, No. 3, May 1978.

5. Avizienis, A. and L. Chen. "On the Implementation of N-Version Programming for Software Fault-Tolerance during Program Execution." Proceedings, Computer Software and Applications Conference (COMPSAC), Chicago, Nov. 8-11, 1977.

6. Hamilton, M. and S. Zeldin. "AXES Syntax Description," TR-4. Higher Order Software, Inc., Cambridge, MA, Dec. 1976.

7. Hamilton, M. and S. Zeldin. "The Foundations for AXES: A Specification Languaged Based on Completeness of Control," Doc. R-964. Charles Stark Draper Laboratory, Inc., Cambridge, MA, March 1976.

8. Guttag, J. "The Specification and Application to Programming of Abstract Data Types." University of Toronto Technical Report CSRG-59, Sept. 1975.

9. Hoare, C.A.R. "An Axiomatic Approach to Computer Programming." CACM 12, Oct. 1969.