DEBUGGER: A MUMPS PROGRAM THAT DETECTS ERRORS IN MUMPS PROGRAMS

Charles S. Volkstorf

Computing Center
University of Massachusetts Medical Center
55 Lake Ave. North
Worcester, Massachusetts 01605

Introduction

Unfulfilled expectations are common in Data Processing. Human errors occurring in the development and maintenance of computer programs cause disillusionment among new users anticipating automatic, error-free performance from their systems. A study of the nature of the errors occurring in the programming of a Medical Records Tracking System has led us to develop an effective, but time-consuming, procedure for averting these problems during the early stages of program development. Further investigations have led us to develop DEBUGGER, a set of programs that automates the detection and correction of errors in applications programs. A mathematical analysis of this system shows a continuum among Program Verification, Program Synthesis, and formal Programming Language Semantics, and prompts an interesting, as yet unsolved, problem for future research.

I. The Paradox of Data Processing

There is a paradox in Data Processing: To an organization that is currently maintaining a lot of information manually, the possibility of having a computer keep track of this information prompts a number of expectations.

1. Accuracy: There is no longer any chance of human errors in all of the calculations and other manipulations that must be performed. After all, everyone knows that "Computers never make mistakes."

2. Automation: The computer will do everything automatically. Reports that used to take us forever to compile and type by hand will now be printed by the computer, as we just sit back and watch.

But most DP Directors have seen otherwise. We have all heard stories from fellow DP-ers, the worst of which might sound like this:

1. (Accuracy): The reports are wrong. One report says we had 523 orders last month, and the other says we had 724. And under the heading "Type A", it says we had -3 orders.

2. (Automation): We spend all of our time trying to straighten out the system. We have two programmers that don't have any time to write new programs, because it takes all of their time to keep what we already have running.

What went wrong?

The premises are in fact true. Computers essentially never make mistakes in their calculations, and they can be programmed to run with just the touch of a button. Yet, when we go from the theoretical to the practical application, all too often something goes awry.

The problem, as most of you know, is:

"A computer is only as good as the person who programmed it."

If a clerical worker has to compile and type the reports by hand each month, perhaps 1% of the figures will be in error. If a computer performs the chore, the programmer need program it only once (hopefully). The computer will do the work automatically thereafter. But the programmer is liable to have his own 1% error rate in the writing of the programs. And when he tries to fix them, or make any changes at all for the user, all too often he introduces more in the process. Herein lies the resolution of the paradox, and the source of so many unrealized expectations. The computer only multiplies your efforts, errors and all.

II. Why Bugs Occur

What can be done about the realities of the fallibility of programmers? At the University of Massachusetts Medical Center, we have made a concerted effort to tackle this problem, with very satisfying results.

The hardware consists of two DEC PDP 11/70s plus an 11/34, all linked via DMCs, running Standard Mumps. The applications software is primarily a Patient Information System, which includes bed census and billing.

As a first step in the analysis of the problem of assuring accuracy in programs, we made an informal study of the kinds of human errors ("bugs") that occur during program development. Three situations were most prevalent within our sample.

1. Misunderstanding of a file design.

2. Changing one part of a program to satisfy a user's request, but inadvertently disturbing other parts of that program or its variables.

3. Not considering every condition that can occur when processing pieces of data entered from a terminal or extracted from a file.

As a first approach, stringent rules were imposed in the development and maintenance of software. Specifically, whenever a new program is written, or an existing program changed, a copy of the system is made and thoroughly tested. The files in this copy are initialized, so that the programmer knows exactly what data is in the system. Then the programmer sits at a CRT terminal and repeatedly enters pseudo-random data into the system, checking the validity of the files after each transaction. Furthermore, data is entered which satisfies every type of condition that might effect the internal processing within that program.

This procedure was first applied by this author in the programming of a Medical Records Tracking System (MRS). The purpose of this system is to keep track of deficiencies that cause the Medical Records of discharged inpatients to be incomplete, and to notify the physicians responsible when the deficiency becomes overly late.

For example, if a program requests that the user identify a patient, the programmer successively enters each of the following possible inputs:

1. A valid Medical Record Number.

2. A number that is in the right format as a Medical Record Number, but which isn't assigned to any patient.

3. A number that has too many digits to be a Medical Record Number.

4. An existing patient's entire name.

5. The first few letters of a patient's name, when there is exactly one patient whose name begins with those letters. (The system looks up all patients whose name matches the input letters.)

6. Letters with two or more matches.

7. Letters with no matches.

Furthermore, consideration is given to how the patient will be used in the program. For example, if we are adding a deficiency to the patient's record, we should test the following types of input:

8. Add a delinquent deficiency to a delinquent record.

9. Add a non-delinquent deficiency to a delinquent record.

10. Add a delinquent deficiency to a non-delinquent record.

11. Add a non-delinquent deficiency to a non-delinquent record.

After each completed transaction (such as Add or Delete a Deficiency), all of the files are listed and their validity is visually checked by the programmer. If an error is detected, the programmer knows that it was caused by the last transaction entered, and checks the appropriate programs (figure 1).

There are drawbacks to this approach, however. The main problem is the excessive amount of time required to test every combination of every transaction. While it takes just a few seconds to enter one transaction, manually checking the files takes an order of magnitude more time. Another problem is in making sure that every condition is covered (as is the concern when the program is originally written).

In light of its repetitive nature and the fact that it is fairly well-defined, this author embarked to automate this process. The result of this effort is a system that I call DEBUGGER.

III. DEBUGGER

DEBUGGER replaces the programmer by detecting, and, to a certain extent, correcting bugs in Mumps applications. There were four goals in the design of DEBUGGER.

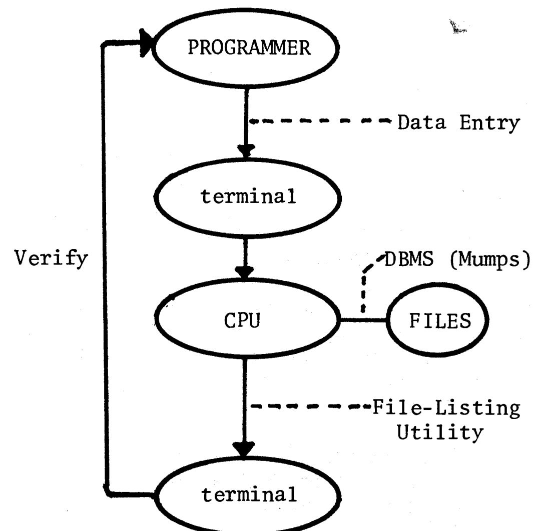

FIG. 1. Manual Debugging Procedure

1. It should be generalized so that it can be applied to any Mumps system.

2. It should be easy to use.

3. It should do the testing quickly.

4. The tests should be thorough.

DEBUGGER performs the above Manual Debugging Procedure, but without human intervention. This involves three main tasks.

1. Generate pseudo-random input.

2. Call the appropriate filing subroutines being tested in the system.

3. Check the validity of the files.

In addition, various utility programs help the programmer determine the exact cause of the bug, and to correct it.

In order to meet the first goal (generality), DEBUGGER provides both a Transaction Definition facility, and a File Definition facility. The Transaction Definition allows the programmer to tell DEBUGGER how to generate the pseudo-random input for this particular application programs. The File Definition allows DEBUGGER to do complete logical checks on the files for individual integrity and mutual consistency. In addition, DEBUGGER allows "escapes" to hand-written programs if the user wishes to add additional definitions to supplement those provided by the system (figure 2).

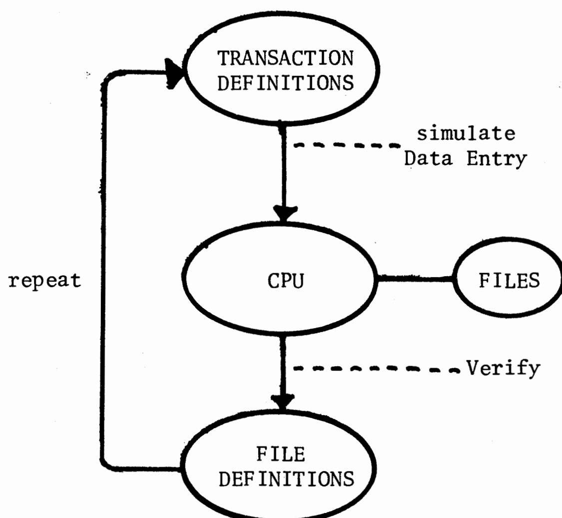

FIG. 2. Automated Debugging

Once DEBUGGER has been interfaced to a particular applications package, the tests can be run anytime. If an error is detected, the processing stops, and the programmer can check and correct the error. There are altogether 8 routines:

1. Input/Delete Transaction Definitions
2. Print Transaction Definitions
3. Input/Delete File Definitions
4. List File Definitions
5. Run Tests
6. Monitor Background Simulator
7. Find the Shortest Test Sequence
8. Re-construct to Right Before Error

Routines 1 through 4 are used to set up the tests (transactions). There is currently no "Edit Transaction Definition" or "Edit File Definition", but rather, the programmer merely deletes the existing one and re-enters it should a change be needed. The complexity of editing has prompted us to rely on this procedure instead.

Routine 5 closes the current device (terminal), and runs the tests in the background. While the tests are being generated randomly, the actual values generated are saved in a file, to be later used by routine 8.

Routine 6 allows the programmer to determine if an error has been found by routine 5. It also allows the user to tell routine 5 to stop, and later resume, its processing.

Routine 7 attempts to find a subset of the error-causing test sequence that also causes an error to occur. This is possible because usually most of the tests actually don't contribute to the error. This way, the programmer can examine a shorter test sequence containing only those tests that are actually involved in creating the error.

Routine 8 re-initializes the files in the application being tested, and re-runs the tests that caused the error, but stops right before the last one. It then sets the variables of this last test into the programmer's local symbol table, and returns to programming mode. At this point, the files are in a legal state, but the transaction in the programmer's partition is one that causes an erroneous file state when executed to completion.

Each of the checks made against the files, for individual integrity and mutual consistency, is assigned a sequential number by DEBUGGER. MRS files have 71 possible errors. DEBUGGER reports the error number and the relevant field values. A typical error message is as follows:

```
ERROR #20: CODE P MRN 723814 VI 2 IN ^MR NOT IN ^MRK
```

This is telling the programmer that medical record number 723814, in which the CODE field is equal to "P" on the second visit, is in file ^MR, but not in file ^MRK. This is an error, because these two files, by their definitions, should contain the same information, although in different orders. This error would have caused reports created from these two files to be inconsistent.

An interesting mathematical aspect of DEBUGGER is the relationship between the pseudo-random Transactions and the File Definition error checks. While a fixed (finite) number of checks are made to the files, there is no limit to the (infinite) number of different combinations of transactions that may be generated. This is because, as DEBUGGER simulates the addition of data to the system being tested, the number of possible values for this data grows. The longer the series of tests that are allowed to run, the larger the number of combinations of values that can occur. Within the files created by the simulation, any number of circumstances might cause the system to fail, for example:

> Input of a non-delinquent deficiency, on a record with a delinquent deficiency but in Code T, causes the record to become delinquent by changing the Code to I and thus activating the existing delinquent deficiency.

This oversight was the first error ever detected by DEBUGGER (9/1/81). While there is an unlimited (infinite) number of possible combinations such as this, all such errors will (theoretically) eventually be detected by the finite number of file checks as creating a logically inconsistent file state.

DEBUGGER also detects when the programs being tested "bomb" (abort) on an error. In order to protect DEBUGGER from also aborting due to the program being called, the programs being tested are started up as a separate Mumps job by using the ZJOB command. DEBUGGER communicates with this separate job through files. If the program being tested bombs on an error, DEBUGGER will detect this by virtue of its not responding within a preset time limit, and reports it to the programmer for analysis and correction.

If routine 5 doesn't find an error after performing 20 transactions, the files are re-initialized and the cycle starts over. This keeps the audit trail of tests that are saved (to be later used by routine 8) to a maximum of 20. This minimizes the amount of disc space used, the time required to run routine 8, and the complexity of analyzing the bug when it is found. (Routine 7 also contributes to minimizing the complexity of the error-causing test sequence.) In practice, however, we have almost always been able to correct the error based on the error-causing transaction plus the current files, without regard to the transactions prior to it. This is to be expected, as the prior transactions merely put the files into a legal state which becomes illegal when the final transaction is run. It is also our experience that bugs can be exposed well within the 20 transactions, although of course the cycle of 20 typically must be repeated many times before each bug is uncovered (depending on how well the system was originally written).

If a system that is in use is to be tested (as opposed to a new system under development), all tests must be run against a test copy of that system. This is because DEBUGGER modifies the files in the system being tested. To aid in this effort, once the users began using MRS, a program was written to make a copy of all of the programs in MRS, but to precede each program and file name with the letter "X" to distinguish it from the original system. It is also possible to merely file a copy of the original programs under a different UCI in order to accomplish this.

Once an error is detected by DEBUGGER (routine 5), and the programmer reconstructs the files (routine 8), it is usually a fairly simple matter to determine the cause of the error. The programmer merely loads the program containing the offending filing subroutines, and follows through the code, having the actual values for the variables in his symbol table as his guide. This is in contrast to more conventional situations in which the bug is missed by the programmer using manual methods, and is later reported by the user when a report or interactive routine gives erroneous results. A number of difficulties are avoided by DEBUGGER.

1. The programmer knows which file is in error, and what aspect of it is wrong. There is no need to trace through programs in order to determine which file contains erroneous data.

2. DEBUGGER provides the files in the exact state that led to the error. In conventional environments, the programmer attempts to trace backwards from the erroneous file state into an originally-valid state that can lead to the current error.

3. The programmer knows what transaction caused the error to occur. There is no need to consider all of the programs that effect the contents of the erroneous files and determine which one might have caused the error to occur.

4. The symbol table of the offending transaction provides the programmer with a single execution path that leads to the error. The programmer merely follows through the logic of the code, until the files become erroneous. In conventional environments, the programmer must consider each execution path, and each possible result from logical checks such as IF commands and conditional commands and arguments, in an attempt to imagine a situation that would lead to the error.

In practice, the programmer using DEBUGGER can often go directly to the subroutine that is in error, based on the error message provided by DEBUGGER. This is more often possible when the programs are highly structured, as is the case with MRS. When DEBUGGER reports that particular files are in error after a particular transaction, this author has almost always been able to go directly to the subroutine responsible, and determine the error within it, without more than a cursory examination of the rest of the programs.

After the programmer makes a change to the system in an attempt to correct the bug, routine 8 can also be used to determine if the programmer has been successful. Routine 8 allows the entire test sequence to be repeated, including the final transaction that caused the error. All the while, the file checks are being made, to see if the error still occurs. Once the error is eliminated, the Run Tests routine can be resumed.

With the help of DEBUGGER, MRS, consisting of 31 programs (segments) and 15 routines, was written between Aug. 24 and Sept. 15, 1981. After using DEBUGGER extensively, a programmer begins to wonder how anyone can ever debug a large, complex system using conventional methods.

As an example of the use of DEBUGGER with MRS, the Transaction Definition listed in figure 3 defines the first routine in MRS, "Manually Add a Discharged Visit".

There are 10 "steps" in this definition. Four types of steps are illustrated.

```
TEST# 1   MANUALLY ADD A DISCHARGED VISIT
          PROG.: FILE^MRA ; WT=4

 1. N ($NEXT)   VAR: MRN
    FILE REF.: ^["ADT","AAA"]MRN(
    $N FROM: 1
    # TO CONSIDER: 10

 2. E (EXPR)   VAR: ID
    EXPR: ^["ADT","AAA"]MRN(MRN)

 3. I (IF)
    EXPR: $D(^["ADT"]BI(2,ID,0))

 4. E (EXPR)   VAR: NAME
    EXPR: $P(^["ADT","AAA"]PT(ID,0),"^",1)

 5. F (FOR)   VAR: VI
    START: 1
    INC: 1
    END: ^["ADT"]BI(2,ID,0)-1

 6. I (IF)
    EXPR: $D(^["ADT"]BI(2,ID,VI*100))

 7. I (IF)
    EXPR: '$D(^MR(MRN,VI))

 8. E (EXPR)   VAR: ADM
    EXPR: +^["ADT"]BI(2,ID,VI*100+3)

 9. E (EXPR)   VAR: DIS
    EXPR: $P(^["ADT"]BI(2,ID,VI*100),"^",5)

10. E (EXPR)   VAR: NS
    EXPR: $P(^["ADT"]BI(2,ID,VI*100+1),"^",4)
```

FIG. 3. Transaction Definition for Routine 1 of MRS

1. N ($NEXT): Randomly selects a subscript at a given file level, starting at a given subscript, and choosing from a fixed number of successive subscripts.

2. E (EXPR): Set a given variable to the value of a given expression.

3. I (IF): Return to the previous F (FOR) if a given expression isn't true.

4. F (FOR): Choose a random value from a given interval of numeric values with a given increment.

The WT (weight) determines the frequency with which each test is randomly selected. It is our experience that "edit" routines are the most error-prone, and consequently they are usually given the highest weight. Also, "add" routines are given higher weights than "delete" routines so that DEBUGGER will tend to build up rich sets of data (visits, deficiencies, doctors, etc.) during each sequence of tests.

The above Transaction Definition chooses a random value (the Medical Record Number) from the first 10 subscripts in ^["ADT","AAA"]MRN(...), makes sure it is an inpatient who hasn't been added to MRS already, and sets a few variables from the patient visit file. These values are passed to the test copy of MRS.

DEBUGGER also sets a limit on the number of times it will attempt to complete a given test, after which another test is chosen. This is needed to avoid infinite loops in which no "FOR" value satisfies the logical constraints imposed by the Transaction Definition. For example, if DEBUGGER were to attempt to "Edit a Deficiency" before any deficiencies were actually added, the "FOR" randomly looking for a patient with a deficiency to edit would never run to completion.

Running on a DEC PDP 11/70 with a light load (10 to 15 simultaneous non-intensive users), the Run Tests routine has averaged 6.7 seconds per transaction.

Since MRS has been in daily use, DEBUGGER has been useful for determining file inconsistencies that are caused by hardware failures. Machine outages that occur in the midst of programs that alter files (for example, filing away a new deficiency) can leave files inconsistent due to the fact that the filing sequence was interrupted midstream and never ran to completion. After any such outage, unless it can be determined that no one was using MRS at the time, we run DEBUGGER to check the files. Any inconsistencies are manually corrected, by either completing a partially-made transaction if enough information is available, or by removing it altogether and notifying the user. We hope to someday eliminate the manual aspects of this procedure and completely automate the recovery from machine outages.

This procedure of checking files also alerted us to a flaw in the overall design of MRS. DEBUGGER

indicated that three particular fields, the Admit Date, Discharge Date, and Patient Name, were sometimes mysteriously inconsistent between MRS and files in other, related, systems. We were surprised to discover that users were making edits to these files on a regular basis, and MRS didn't assure that the edit was reflected in its files. Consequently, MRS was modified to reflect edits made by users of these other systems.

A final unexpected benefit from DEBUGGER has been its ability to "instruct" us as to how to make a given change to a system. It is often the case that users will need to change definitions, add new files, and make other modifications to a system as their needs evolve. When this involves a change to a file design (or the addition of a new file), DEBUGGER can be used to determine which file-maintenance programs need to be altered to accomodate the change. The appropriate definitions in DEBUGGER are changed, and the Run Tests routine is started up. One by one DEBUGGER tells us which subroutines must be altered.

Whenever such a change is made to MRS, we start up DEBUGGER to make sure that the change has been implemented both correctly and that no aspects of the system that need to be modified have been overlooked.

We are currently in the process of applying DEBUGGER to other, existing, systems in order to determine and correct bugs that have long eluded the efforts of our programming staff.

IV. Future Research and an Open Problem

DEBUGGER is an evolving system. Experience with its use (primarily with MRS) has suggested areas in which its power can be expanded, and its use simplified.

The Transaction Definitions and File Definitions can be more "command-oriented". Rather than indicating the type of step and the value of each parameter (such as VAR or FILE REF), the programmer should be able to type in a single line containing all of these values. For example, the example given above might be coded as follows:

```
 1. N MRN=^["ADT","AAA"]MRN(1::10)
 2. ID=^["ADT","AAA"]MRN(MRN)
 3. I $D(^["ADT"]BI(2,ID,0))
 4. NAME=$P(^["ADT","AAA"]PT(ID,0),"^",1)
 5. F VI=1:1:^["ADT"]BI(2,ID,0)-1
 6. I $D(^["ADT"]BI(2,ID,VI*100))
 7. I '$D(^MR(MRN,VI))
 8. ADM=+^["ADT"]BI(2,ID,VI*100+3)
 9. DIS=$P(^["ADT"]BI(2,ID,VI*100),"^",5)
10. NS=$P(^["ADT"]BI(2,ID,VI*100)+1),"^",4)
```

In order to further automate the use of DEBUGGER, the Transaction Definitions and File Definitions should be created by DEBUGGER itself. The user should only be required to type in the name of the first program in the routine, plus the values of any variables that are set by the "Option Reader" (the program that asks the user which routine is desired, and calls the appropriate program). DEBUGGER could follow through the Mumps code, see how variables are created that are passed to filing subroutines, and create the appropriate Transaction Definitions to simulate that routine. Alternately, the name of the Option Reader might be indicated, and DEBUGGER would set up definitions for all appropriate routines.

It is not possible to determine the appropriate File Definitions, since this is one aspect of the application that DEBUGGER attempts to correct. DEBUGGER is basically determining if the given programs match the File Definitions indicated by the user. Alternately, DEBUGGER could determine the actual File Definitions implicit in the file-maintenance subroutines, and report that to the user for verification. This would transcend the current version of DEBUGGER by replacing simulation with analysis of the Mumps code. Even without an indication from the user as to their intent, DEBUGGER should be able to detect inconsistencies between various Add/Edit/Delete subroutines that manipulate the same files. Thus DEBUGGER would detect inherent inconsistencies in applications programs, although, without additional knowledge provided by the user, it might be unable to determine which subroutines are the ones in error. However, a "majority" rule might be adopted, in which agreement among, say, all but one subroutine would prompt DEBUGGER to suspect that the single subroutine is in error. The notion of "consistency" has long been basic to mathematical systems. It is interesting to apply this principle to computer programs.

Ideally, DEBUGGER would then correct the program itself. The logical extreme is for DEBUGGER to construct the entire program based on the user-supplied definitions. That is, given an empty program (a single QUIT command) and the desired program definitions, DEBUGGER's "correction" would consist of generating the entire correct program. Thus we see a continuum from debugging, which has classically been called Program Verification (Manna, 1974), and Program Synthesis (Manna and Waldinger, 1971).

This can be taken a step further, as illustrated in figure 4. Thus a system, such as DEBUGGER, in which the user inputs both the Property (definitions) and Program (to be tested), performs Program Verification. Likewise with Program Synthesis and

	PROPERTY	
PROGRAM	INPUT	OUTPUT
INPUT	PROGRAM VERIFICATION	PROGRAM ANALYSIS
OUTPUT	PROGRAM SYNTHESIS	PROGRAMMING LANGUAGE SEMANTICS

FIG. 4. The continuum in the study of computer programs

Program Analysis (determining the properties of a given program). A system which outputs both a list of possible programs and their corresponding properties constitutes a definition of the Programming Language Semantics. For example, (Mumps Development Committee, 1977), as any programming language reference book, details a means of constructing all possible Mumps programs, and the properties of each (based on the given rules governing Mumps).

This continuum can be expressed in more formal terms. The PROGRAM and PROPERTY form a two-place relation, call it R, defined as follows:

R(PROGRAM,PROPERTY) is TRUE iff computer program PROGRAM has property PROPERTY.

In this paper, the value of PROGRAM has been the 15 routines constituting MRS, while the PROPERTY has been the perpetual maintenance of the files in accordance with the File Definitions. (The Transaction Definitions serve as a means of verifying this assertion.) When each of these two components is either input or output, the four combinations formalize the processes illustrated. We can make the notions of input and output more precise by representing components that are input, by the letters I,J,.. and those components that are output, by the letters X,Y,.. Then the four processes discussed are represented by the following WFFs:

R(I,J) Program Verification
R(I,X) Program Analysis
R(X,I) Program Synthesis
R(X,Y) Programming Language Semantics

The conventional means of expressing manipulations on relations is the Predicate Calculus (for example, Manna, 1974). The use of input and output variables merges well with the Predicate Calculus quantified variables, to form what we will call the Extended Predicate Calculus, or EPC for short.

EPC is useful for expressing and analyzing a wide range of mathematical ideas. We note that, as a Query Language, it is simpler and shorter, yet actually more powerful, than the Relational Calculus (Codd, 1971). EPC may also be used to formalize, and subsequently eliminate through logical consequence, many "undefined primitives" of classical mathematics, such as Peano's Postulates (Dedekind, 1888) that are commonly used to axiomize Arithmetic (for example, Gödel, 1931 and Stoll, 1974). See (Volkstorf, 1982) for details (available from the author).

We now examine the internal structure of DEBUGGER from a mathematical point of view. In terms of mathematical concepts, the basic components are as follows:

1. constant: The program that initializes the files in the system being tested is setting these files into a fixed state. If we treat these files as a single number, say, their Gödel number (Gödel, 1931), then initializing the files sets in a constant. (It is a simple matter to apply Gödel's functions to Mumps files, for example, treating the entire data base as a single proof, and each node in these files, in collating sequence, as a theorem within that proof.)

2. function: The Transaction Definitions, plus the applications programs being tested, apply a function to these files, transforming them into a new state (number). Even though $R is used to generate pseudo-random transactions, the Operating System is actually using a mathematical function (typically A*X+B#C for constants A, B, and C) to generate the necessary random numbers. Thus we need only include the hidden current value of X along with the files to indicate the current state.

3. predicate: The File Definitions apply a predicate to the current file state, to determine if they pass or fail the checks for logical consistency.

If we let the constant, function, and predicate be k, f, and P, respectively, then the tuple (k,f,P) defines the system being tested by DEBUGGER. The effect of DEBUGGER is to determine if the following program HALTs or not (figure 5).

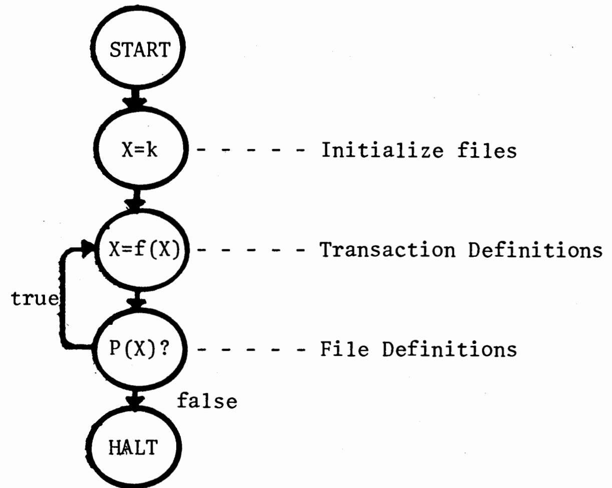

FIG. 5. Mathematical model of DEBUGGER

The assignment statement X=k represents initializing the files, X=f(X) represents running a transaction, and P(X) represents checking the files with the File Definitions. If this program ever HALTs, then the system (k,f,P) has a bug in it, which occurs when the system (files) enters state X. DEBUGGER attempts to determine if this program HALTs, by simulating its execution, reporting to the programmer if and when it does detect an error.

Let us call the problem of mathematically determining if a given (k,f,P) has a bug in it, that is, whether or not its corresponding program ever HALTs, the Debugging Problem. While it is in general

impossible to determine if any program whatsoever will HALT (Turing, 1937), how about the above, particularly simple, program? This will be left as an open problem.

This author welcomes personal correspondence, and may be reached at the above address.

Acknowledgements

I would like to thank my sister, Nina Bug, for proofreading this paper for me, the Computing Center of the University of Massachusetts Medical Center, and Dennis L'Heureux in particular, for their help and encouragement, and especially John Sloan, Astronomer/Physicist par excellence, for diligently critiquing and suggesting numerous improvements in its presentation.

References

1. Codd, E. F. 1971 Relational Completeness of Data Base Sublanguages IBM Research, San Jose, California, in Rustin, Randall editor Data Base Systems Prentice-Hall, Inc.

2. Dedekind, Richard 1888 Was sind und was sollen die Zahlen? ("What are the numbers, and what should they be?") Brunswick. English translation in Beman, W. W. 1901 Essays on the Theory of Numbers Chicago: Open Court, 31-105

3. Gödel, K. (1931) On Formally Undecidable Propositions of Principia Mathematica and Related Systems I in From Frege to Gödel 1967 Harvard University Press

4. Manna, Zohar 1974 Mathematical Theory of Computation McGraw Hill

5. Manna, Zohar and Waldinger, Richard March 1971 Toward Automatic Program Synthesis Communications of the ACM

6. Mumps Development Committee 1977 American National Standard Mumps Language Standard

7. Stoll, Robert R. 1974 Sets, Logic, and Axiomatic Theories W. H. Freeman and Company

8. Turing, A. M. (1937) On Computable Numbers, with an Application to the Entscheidungsproblem Proc. London Mathematical Society p. 230

9. Volkstorf, Charles S. April 1982 Query Languages, Predicate Calculus, and Primitives of Mathematics Computing Center, University of Massachusetts Medical Center (preprint submitted for publication)