FUNCTIONAL PATH ANALYSIS: AN APPROACH TO SOFTWARE VERIFICATION

John W. Sheppard
William R. Simpson
ARINC Research Corporation
2551 Riva Road
Annapolis, Maryland 21401

ABSTRACT

A widely accepted approach to software development involves successive refinement of design and requirements specifications from a top-level description of the system down to the code level. As the system is refined, it is verified at each phase of development before proceeding to the next phase. In the past, several tools and techniques have been developed to assist in the development and verification process.

Tools have been developed and have been in use for many years to examine the testability and system information flow in hardware systems. These problems are approached as a knowledge base verification and validation problem. Several strong analogues exist between hardware and software systems. However, several fundamental differences exist which affect the approach to modeling and verifying the system.

This paper briefly describes past efforts in verifying hardware and software systems and then presents a preliminary synthesis and extension of these past efforts to the software verification problem. We then conclude with an assessment of our current status and note future directions and recommendations for research in this area.

INTRODUCTION

The structured process of developing software systems (as delineated in MIL-STD-2167 and sometimes called top-down design or the waterfall model) requires a disciplined approach to design, implementation, and testing. In order to ensure compliance with the requirements specification of the system and maintain quality control throughout the software life cycle, each level of the development process needs to be verified before proceeding to the next, more detailed level. This structured approach is seldom followed because the additional cost of the structure imposed may exceed the cost of simply developing executable code.

Software developers are faced with the verification and validation of the system under development as the system progresses through its life cycle and the prevention or early detection of as many errors as possible. What follows is a description of an approach to software verification utilizing artificial intelligence techniques that, when applied at all phases of the software life cycle, should produce a reliable system with detailed documentation, both under the structured approach and in conjunction with "actual" software practices. It should also provide an approach to isolating design and implementation errors in the final product. The approach is intended to be used in conjunction with current approaches and development environments; however, it may also be used as a stand-alone tool for assisting in the development and verification process. The current implementation is limited to the latter phases of the software life cycle, namely, code development and maintenance. However, the authors feel that this can be extended to the entire life cycle.

BACKGROUND

Past Efforts

Several techniques and tools are currently in use for conducting V&V (Verification and Validation) of software systems. The techniques are often classified according to the following four categories:[1]

1. Static and dynamic techniques
2. Formal and informal techniques
3. Automated and manual techniques
4. Functional and structural testing

These are intended to be applied throughout the design and implementation process, yet they are primarily applied in practice at the implementation phase in verifying that the coded system conforms to requirements.

Other more specific techniques are intended to be applied to testing only. However, these techniques offer elements valuable to a more generalized approach to software V&V. The approach we are proposing incorporates elements of graph theory (path analysis) and functional analysis to determine functional interdependencies within software. These functional interdependencies are then analyzed to generate recommendations for improvement to the system throughout design and implementation and a strategy for testing the completed system.

Assumptions

The following assumptions are made:

a. Software V&V ideally occurs throughout the software life cycle. As development proceeds through successive phases, the V&V process is repeated to ensure continued compliance with requirements. Therefore, the following approach to V&V is intended to be applied throughout design and development with final goals being a validated system and a testing strategy in the event of the later uncovering of errors (latent defects). We are currently limited to the latter phases of code development and maintenance; however, we do not see extension to other phases of software development as a problem.

b. The compiler used will verify the syntactic (and, to a limited extent, semantic) correctness of the coded system. The proposed method will be left to analyze the functional interdependencies of the system.

c. It is assumed that we are dealing with higher-level languages that have some code structure breakdown by module. We define a module to be an identifiable subset of the code that has a beginning and an end and may be linked as an entity into one or more locations of the code as a whole.

d. The process of choosing the type of tests to be performed depends on the system under development. It is assumed that a set of tests either exist or may be designed from the specifications as needed to determine the correctness of some function of the software system. The outcome of the test will then be evaluated by some external agent (often referred to as an oracle).[2] Thus, the approach intends to determine a test strategy using a defined test set. It may also recommend modifications to the test set.

e. It is assumed that verification tests can be considered information sources and that software errors (potential isolation to anomalies) are conclusions that may be drawn from these tests. The performance of a test will provide information as to the proper or improper functioning of some subset of the code under consideration. Information content is a measure of the quantity and quality of the information provided. The primary concern in software testability is evaluating the information content of these information sources as related to the desired conclusions for the purpose of verifying the design or implementation of the system. Thus, the problem is treated as a knowledge-base V&V problem. The analysis performed identifies ambiguous and indistinguishable points within the knowledge base. It also identifies unreachable conclusions, hidden conclusions, falsely indicated conclusions, knowledge-base circularities, and redundancy of information sources.[3] For purposes of this paper, we are assuming a module level analysis. That is, tests provide information as to the proper functioning of modules, and the proper/improper functioning of modules are conclusions.

f. In developing an approach to software verification, several similarities were discovered to exist between software and hardware testability analysis. Both hardware and software systems may be modeled as a flow of events and, therefore, may be represented as a knowledge-base V&V problem. The systems consist of information sources and fault-isolation conclusions that are drawn from the information provided. As a result, the actual design of the tests (information sources) may be treated as independent of the fault-isolation task. Fault isolation occurs using tests built into the system. Test design then verifies the information source dependencies or recommends modifications to them.

g. There exist some fundamental differences between hardware and software systems. Hardware analysis assumes the existence of a verified system in which components may fail. When the components fail, the system is fault isolated and the component is either repaired or replaced. Software, on the other hand, does not have components that can fail and be repaired or replaced. A fault in software is a design flaw. By design flaw, we mean a coding, logic, or compiler-acceptable syntax/semantic error. (A requirements error is not considered a design flaw and is assumed to be handled during validation.) When a fault is uncovered, the system must undergo some level of redesign to correct the fault.

Path Analysis

One of the most common approaches to software testing is path analysis (which incorporates elements of graph theory).[4] In performing path analysis, the module to be tested is modeled

as a directed graph (digraph), and tests are performed to cover the various paths through the module.

Functional Analysis

Another approach to software testing is functional analysis, which is based upon the assumption that a program is made up of one or more functions, each of which is a combination of smaller functions.[5] Within a module, the functions may be visualized as blocks corresponding to sequential blocks, conditional blocks, and iterative blocks, or calls to other modules. In verifying these functions, all lower level functions must be verified as well. This indicates a hierarchical interdependency of functions which is ideally suited to verification throughout design and implementation.

System Testability and Maintenance Program (STAMP®)

ARINC Research Corporation's STAMP is a computer-aided testability design and fault-isolation development tool. It has been applied to a number of hardware systems, often achieving field maintainability improvements of 60 to 100%. STAMP analyzes a model representing a system's functional interdependencies to determine the system's level of testability. It then provides a detailed strategy to isolate anomalies (faults) in the system.

The system to be analyzed by STAMP must be modeled according to the functional flow of information through the system. This flow of information is represented as interdependencies of information sources; therefore, the immediate (or first-order) dependencies of each information source (test) must be determined. These first-order dependencies are then input for analysis, and STAMP generates all higher-order dependencies, analyzes the system's tests, makes recommendations for modification of these tests, and generates a fault tree using the tests provided.[6] In performing its analysis, STAMP uses a unique inference engine that pre-chains the dependencies of the system under study, using a form of matrix closure.[7]

In STAMP, the assertion of a functional information source as true corresponds to a failed (or bad) test outcome. STAMP evaluates each information source according to the amount of information that can be gained from the assertion of these information sources as true (fail) or false (pass) and determines the best sequence of procedures to locate the anomalies that may exist in the system based on the amount of information available at each point.[8]

A system for analyzing software testability is currently under development. The method proposed below follows well defined rules for analysis of software structures. As a result, it is ideally suited to automatic analysis of design documentation and system code. In fact, the system under development currently performs automatic modeling and analysis of coded systems written in FORTRAN and various FORTRAN dialects. Currently, auto-analysis occurs only at the module level; however, extension to more detailed levels is not seen to be a problem. The prototype system uses the inferencing process employed in STAMP and will generate metrics directly related to software in addition to testability metrics.

A SOFTWARE TESTABILITY APPROACH

Functional Path Analysis

Functional path analysis of software systems combines path and functional analysis approaches to software testing with the functional approach to testability analysis performed by STAMP. This approach evolves through three levels which should be applied throughout design and implementation where appropriate.[9] These three levels are:

1. Module Level analysis
2. Functional Path Level analysis
3. Independent Function Level analysis

These three levels provide a means of efficient verification of a system and successive localization of anomalies in the system in the event verification fails.

During the code development and maintenance phases of the software life cycle, the design process consists of iterative refinement of the system. At each step in development, the system may be modeled at the module level in the form of a digraph where modules are represented as nodes, and the modules' functional interdependencies are represented by edges between nodes with a direction specified.

Module digraphs are primarily represented in the form similar to the visual table of contents (VTOC) of Hierarchy-Input-Process-Output (HIPO)* diagrams (Figure 1). Control and data

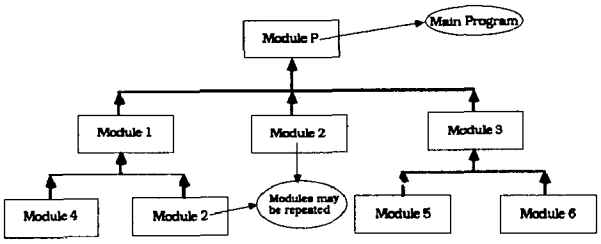

FIGURE 1

MODULE DEPENDENCIES

flow are not indicated by the diagrams; however, the diagrams serve well to show the overall interdependency of the system's modules. The direction indicated on the diagrams reflects the functional flow of information (as opposed to data) through these modules; the lower levels

*IBM terminology

feed the higher levels, thus the higher levels depend on the lower levels. (In representing recursion, only the initial call to the recursive routine is indicated in the hierarchy. Recursive testing is handled at the functional path level with the specific call.) Generally, it is assumed that a functional test will be placed on the "output" of each module. This is represented by indicating a test point attached to the line feeding the higher-level module. This level of analysis corresponds to integration testing in which the way modules interconnect with each other in the system as a whole is of interest.

Digraphs representing the structure of the modules (Independent Function stage) are also drawn with functions (or statements when at the code level) being represented as nodes and control/data flow represented as directed edges. Once again, the direction reflects the functional flow of information through the nodes, but it also combines the information represented in data flow and logic flow digraphs. The resulting model represents the module at a high level of parallelism (Figure 2). For example, a sequence

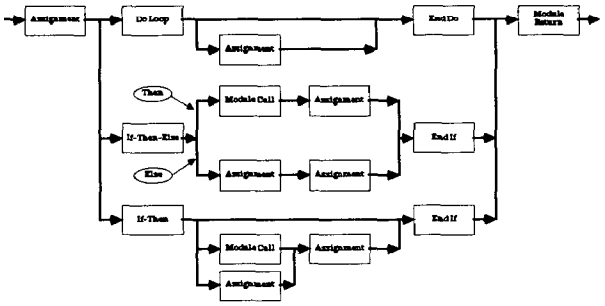

FIGURE 2

STATEMENT DEPENDENCIES

of assignment statements that do not share variables are diagrammed in parallel. In a serial machine, they are dependent upon one another because of the order imposed on them, but functionally, they have no interdependency.

In Functional Path Level analysis, the first step is to generate the functional digraph of the module to be analyzed (discussed above). Paths representing the flow of information at a single level of control are then separated out of the graph. The level of control is said to have changed upon (a) calling another module, (b) executing the result of a conditional, or (c) entering an iterated set of functions (a loop). The resulting digraph (Figure 3) represents the functional dependency of these levels with the higher-level elements (lower-level numbers), depending upon the lower-level elements.

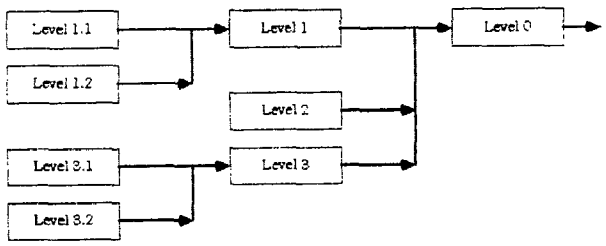

FIGURE 3

PATH LEVEL DEPENDENCIES

As these three levels of analysis (Module, Functional Path, and Independent Function) are completed, an algorithm, such as the one employed by STAMP, is used to determine the test strategy to verify the system. The software testability prototype inference engine is based on an information theoretic search algorithm that chooses tests that provide the most information. The resulting sequence of tests may then be followed to isolate an anomaly in the system in the fewest steps. A specific example of the application process is given below.

Example Application

An analysis was made of an existing program at the module level. The program chosen, called COMBINE, merges two STAMP input files (File 1 and File 2) into one file (File 3) for STAMP processing. The prototype system automatically scanned the coded system and generated the dependency model for analysis.

The module level dependencies were generated automatically by our prototype system using the method outlined above. The module level digraph appears in Figure 4. A test is assumed to be

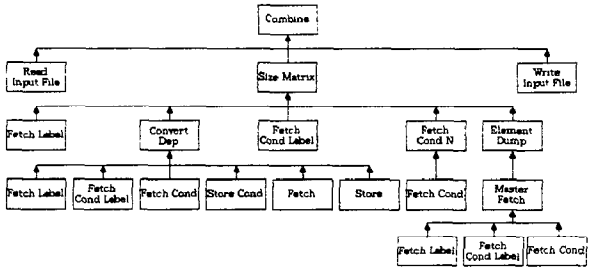

FIGURE 4

MODULE DEPENDENCIES - COMBINE UTILITY

associated with the output of each module. and the output should feed its calling module. Next, the first-order dependencies are generated. The first-order dependency of a test is determined by following the paths that feed the test

back until another test is found or until the path ends. The first-order dependencies for COMBINE's modules are as follows:

	Test	Dependencies
1.	Test Combine	Combine, Test Read Input, Test Write Input, Test Size Matrix
2.	Test Read Input	Read Input File
3.	Test Write Input	Write Input File
4.	Test Size Matrix	Size Matrix, Test Fetch Label, Test Convert Dep, Test Fetch C Lbl, Test Fetch C N, Test Elem Dump
5.	Test Fetch Label	Fetch Label
6.	Test Convert Dep	Convert Dep, Test Fetch Label, Test Fetch C Label, Test Fetch Cond, Test Store Cond, Test Fetch, Test Store
7.	Test Fetch C Lbl	Fetch Cond Label
8.	Test Fetch C N	Fetch Cond N, Test Fetch Cond
9.	Test Elem Dump	Element Dump, Test M Fetch
10.	Test Fetch Cond	Fetch Cond
11.	Test Store Cond	Store Cond
12.	Test Fetch	Fetch
13.	Test Store	Store
14.	Test M Fetch	Master Fetch, Test Fetch Label, Test Fetch C Lbl, Test Fetch Cont

Once these dependencies have been determined, all higher-order dependencies are calculated. These higher-order dependencies are then analyzed to determine the test strategy. The decision tree for COMBINE is given in Figure 5.

To read this tree, start at Step 1 and perform the requested test. If the test passes, proceed to the step indicated in the Pass column. If the test fails, proceed to the step indicated in the Fail column. When an anomaly is isolated, the module name is given in the column corresponding to the last test outcome. If the system verifies as correct, the isolated conclusion should be "System Verified." If functional "debug" prints are coded into the module, then system anomalies can be isolated with the resident data that gave the first indication of anomalous behavior.

Let us assume that there is an anomaly with the utility; the labels do not properly correspond to the dependencies being mapped into the new file.

Step 1:	Test Convert Dep Convert Dep maps the dependencies from the input files File 1 and File 2 into File 3. After checking the mapping procedure, it is determined that the dependencies are being mapped properly.	PASS
Step 2:	Test Size Matrix Size Matrix determines the size of the new file (File 3), maps the dependencies from the two input files (File 1 and File 2), and calls Element Dump. The dependencies map correctly, but the labels do not correspond with the default labels assigned by STAMP.	FAIL
Step 6:	Test Elem Dump Element Dump displays the dependency mappings as they occur. As in Step 2, the labels are wrong.	FAIL
Step 8:	Test M Fetch Master Fetch returns the label of any element given its row in the matrix. Some of the labels are wrong.	FAIL
	Master Fetch is isolated.	

In many cases, it is sufficient to isolate to a problem module. Once in the module, given the symptoms of the anomaly, it becomes obvious where the problem lies. However, at times the complexity of the module may be such that the solution is not obvious. This is when the next level of analysis is performed.

TEST CHOICE AND EVALUATION

It must be pointed out that the problem of designing the appropriate test for use in any form of software verification is a difficult one, and such is the case in this approach as well. Therefore, the subject of designing tests will only be dealt with generically in this paper and will be limited to the module level.

Testing of any item may be thought of as a stimulus-response analysis. In general, a functional test must provide as outputs both the stimulus and the response so that the oracle may evaluate the response in terms of adequate or anomalous (pass/fail) behavior. A special test may provide its own stimulus. In software systems, this may be a "debug" call to a routine with the appropriate data contrived for the test. In such cases, the known stimulus should provide a known output to be evaluated as adequate or anomalous.

A module test is intended to verify that a module, independent of the system, is in compliance with that module's design specifications. It need not take into account how the

Step	Test	Prev Step	Pass	Fail
1	Test Convert Dep	0	Step 2	Step 9
2	Test Size Matrix	1	Step 3	Step 6
3	Test Combine	2	System Verified	Step 4
4	Test Read Input	3	Step 5	Read Input File
5	Test Write Input	4	Combine	Write Input File
6	Test Elem Dump	2	Step 7	Step 8
7	Test Fetch C N	6	Size Matrix	Fetch Cond N
8	Test M Fetch	6	Element Dump	Master Fetch
9	Test Elem Dump	1	Step 10	Step 13
10	Test Store Cond	9	Step 11	Store Cond
11	Test Fetch	10	Step 12	Fetch
12	Test Store	11	Convert Dep	Store
13	Test Fetch C N	9	Step 14	Fetch Cond
14	Test Fetch C Lbl	13	Fetch Label	Fetch Cond Label

FIGURE 5

COMBINE DECISION TREE

module has been integrated in the system. The module may be called from several locations using different parameter values, yet only one test is to be designed for that module. In designing the module test, however, one must take into account the stimulus and response (data passed into the module, the expected results given the passed data, the possible effects on global variables and data structures, and the ramifications associated with side effects). The test being dependent on called modules, as well as the module it is testing, must also take into account the effect these called modules have on the tested module.

A first attempt at designing the module level test is derived from the Functional Path Level analysis of the module. Once a module has been modeled at the functional path level, a test strategy for the module may be generated using the functional path level tests. The design of these tests are more directly related to the structure of the module and should therefore be more straightforward.

EXTENSIONS OF CURRENT WORK

The procedure discussed above shows considerable promise in the area of software validation and verification. However, there are several issues we have just started to consider that will need further investigation. These include:

a. Test design issues - Determining techniques and approaches to mapping system requirements to tests of the system.

b. Test evaluation techniques - Evaluating large quantities of data to determine whether the test passes or fails based upon system specifications.

c. Real-time modeling and temporal dependency - Modeling systems to consider timing factors concurrent with functional dependencies.

d. Inconsistency in test outcomes - Determining consistency between test design and evaluation, system implementation and requirements, and system implementation and modeling.

e. Software metrics - Analyzing quantifiable attributes of software to tag potential sources of error and candidates for redesign.

f. Automated software modeling - Analyzing program design languages, software modeling schemes, and coded systems to automatically develop the dependency model for the system.

g. Built-in tests - Embedding code in the system to allow testing of the software according to the generated test strategy.

h. Extension to all development phases - Applying the above techniques to design and requirement phases of the software life cycle.

SUMMARY

An approach has been presented for verifying software systems at the code development and maintenance phases of the software life cycle. The approach involves modeling the system hierarchically (module level) and then breaking down each module to the lowest level and modeling its implementation. The lowest level is then generalized to a point in between where functional paths are modeled. These paths are

hierarchical in nature yet are based on the control flow of the module. After the system is modeled, verification occurs from the top level down. Modules in conflict with design specifications are isolated. Then, the path that contains the conflict is identified. Finally, if the problem is not readily apparent, the independent function containing or contributing to the problem is located (these latter two points have not been fully automated yet). A development effort that incorporates this type of approach in verifying the system will end with (a) a well documented system, (b) a verified software system, and (c) a detailed strategy to isolate the sources of future software errors as they become apparent.

REFERENCES

1. James Martin and Carma McClure, Structured Techniques for Computing, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985, pp. 671-675.

2. William E. Howden, "The Theory and Practice of Functional Testing," IEEE Software, September 1985, p. 6.

3. Jerry L. Graham, "Knowledge Base Verification," Proceedings 1987 Symposium on Space Technological Challenges for the Future, U.S. Naval Academy, Annapolis, Maryland, May 1987.

4. Shashi Phoha, "A Quantifiable Methodology for Software Testing: Using Path Analysis," United States Air Force Report ESD-TR-81-259, Hanscom Air Force Base, Massachusetts, December 1981.

5. William E. Howden, pp. 6-17.

6. W. R. Simpson and H. S. Balaban, "The ARINC Research System Testability and Maintenance Program (STAMP)," Proceedings 1982 IEEE AUTOTESTCON Conference, Dayton, Ohio, October 1982.

7. W. R. Simpson and B. A. Kelley, "Multi-Dimensional Context Representation of Knowledge-Base Information," Proceedings of the 1987 Data Fusion Symposium (DFS-87), Laurel, Maryland, June 1987.

8. B. A. Kelley and W. R. Simpson, "The Use of Information Theory in Propositional Calculus," Proceedings of the 1987 Data Fusion Symposium (DFS-87), Laurel, Maryland, June 1987.

9. The appropriateness of a stage being applied depends upon the level of detail of the system documentation.