Verification of a virtual storage architecture on a microprogrammed computer

by W. A. SCHWOMEYER
International Business Machines Corporation
Endicott, New York

INTRODUCTION

The verification of a virtual storage architecture shares many commonalities with the verification of any computer architecture. This paper will therefore define an approach to architecture verification in terms which are independent of virtual storage. It will then discuss the application of this approach to a virtual storage architecture. The paper will conclude with a discussion of a method for determining the completeness of an architecture verification process on a microprogrammed computer.

ARCHITECTURE

The functional description of a computer as observed by an external user or programmer is usually contained in an architecture for that computer. It is the purpose of an architecture to provide a functional description based upon the requirements of the external user or programmer, independent of any particular implementation. In fact, an architecture may serve as the functional description for several models of a computer family with each model implementing the same functions in different technologies or media. The architecture is the base for determining functional compatibility between models.

The architecture will include a definition of the priorities for handling all simultaneously occurring functions. Where the architecture justifiably defines the priority to be "unpredictable," the exact priority definition is left to each implementation.

An architecture defines two types of entities. The computer state entities define the state of the computer as it can be interrogated by the external user or programmer. Some examples of computer state entities are: main storage contents, programmable register contents, comparison and arithmetic computation result indicators, interrupt mask indicators, and supervisor or problem state indicators.

The computer function entities define processes which, when performed, result in a change, or a significant lack of change, in the computer state entities. The most familiar computer function entities are the execution of machine instructions. These instructions are usually grouped into one or more classifications based upon the type of operands on which they operate. Some examples of these classifications are: storage instructions, register instructions, decimal arithmetic instructions, floating point arithmetic instructions, emulation instructions, and timing facility instructions. Often these classifications are analogous to certain optional or standard computer features.

Other examples of computer function entities are: interrupt processing, manual operations, and timing facilities.

Architecture verification in the context of this paper is the process which verifies whether a particular computer implementation conforms to the functional definition in its architecture. In terms of the previously defined architecture entities, this means: execute the computer function entities and observe and verify the resulting change (or lack thereof) in the computer state entities. It is generally expected that the architecture has already been verified as adequately meeting the external user's requirements.

TEST PHILOSOPHY

To understand the principles discussed later in this paper it is necessary to be familiar with the circumstances under which an architecture verification test is performed. Architecture verification is only one of the many tests performed during the course of designing a computer, building a prototype, and manufacturing a finished product.

In addition, these tests, including the architecture verification, may be performed repeatedly at different times within a development cycle.

Architecture verification tests are usually conducted in two environments. In the first environment, each computer function entity is tested under static or quiescent conditions with no interaction from other function entities. Ironically, testing in this environment has been the most difficult to control.

Historically, it has been characterized as being greatly dependent on manual operations. Tests were entered into
the computer manually and the results recorded manually. The test was a slow and tedious process. Heavy reliance on human intervention created many exposures in the effectiveness of the test. Oversights and discrepancies in recording test results, misunderstandings in communicating exact manual procedures from one group to another, and oversights in performing required tests at all necessary stages of development are results of human error in performing the manual procedures.

Improvements in the control of testing in the static environment will be evident in the discussion under TEST OBJECTIVES and TEST APPROACH.

The second environment involves testing the computer function entities under dynamic conditions. Differing amounts of random interaction are introduced to verify that there are no problems when many function entities contend for CPU (central processing unit) resources. Test programs are written which strictly control and verify the execution of the various computer function entities.

These test programs execute under a supervisor program which allows them freedom in controlling all computer state entities. The supervisor program controls the initiation and termination of execution of the test programs. Interaction between the function entities tested by concurrently executing test programs occurs at random. The existence of the test and supervisor programs ensures the capability to repeat the test as needed and decreases the amount of manual procedures which must be communicated between various groups performing the test.

Deficiencies in the test result from the fact that it is still a manual procedure to verify the thoroughness of the test and the test programs usually employ only valid architecture entities. The latter deficiency results from the assumption that the computer correctly detects architectural violations. Enhancements to testing in the dynamic environment will be discussed under VIRTUAL STORAGE.

TEST OBJECTIVES

Based upon the previous discussion, the following desirable traits or objectives can be defined for a basic test approach to be used to verify an architecture in the static test environment.

- **Controlled Environment**—The test must be performed on a static or quiescent computer with no interaction between functions.
- **Ease of Use**—The test must be easy to perform (usually implying a minimum of manual intervention) and the results must be easy to interpret. It will be recalled that the test will be performed at different times in the development of a computer. As may be expected, there are differing users who will perform the test and have differing information requirements from the results of the test.

In addition, the test will be performed on a variety of computers or models with different features installed. It must be easy to perform the test on each of these computers with little concern on the part of the user for the features installed.

- **Repeatable**—Because of the number of times the test will be performed, it must be repeatable with the expectation that the results of the previous tests will be duplicated each time it is performed.

Often a simulator is used in the early stages of computer development. It is advantageous to develop an architecture verification process which not only can verify the architectural implementation of the simulator but also can verify the architectural implementation of the resulting computer. This will ensure that the computer accurately duplicates the design in the simulator.

- **Thorough**—It is the intent of the test to verify 100 percent of the architectural implementation. However, in practice some compromise is usually made between the thoroughness of the test and the time and cost required to achieve 100 percent verification.
- **Transparent**—The test must be performed in a way which does not alter the architectural implementation being verified.
- **Fast**—Due to the number of times the test is performed, it is desirable to perform the test as fast as possible. An increase in speed can be obtained by decreasing the amount of manual intervention required.
- **Model Independent**—The test should be model independent to allow the same test to be used in verifying multiple implementations of the same architecture. This will also verify the compatibility between the computer models.
- **Extendable**—While the test should be model independent, it should also be extendable to allow the testing of model dependent results (architecturally "unpredictable") when required. This capability will also allow extension of the test if additional architectural verification requirements are determined.
- **Portable**—The test tool must be easily transportable from one computer or model to another, whether they are located at the same computer installation or at locations separated by large geographic distances.

In addition to the above objectives, there is one additional objective not directly related to the effectiveness or usability of the test vehicle itself.

- **Ease of Implementation**—The test tool should be easy to implement, providing as much flexibility in the implementation process as possible. As indicated above, the test should be easily extendable to provide additional architectural verification as the requirements are determined.
TEST APPROACH

The test approach for verifying an architecture in the static test environment is divided into the following parts: control program, test cases, and support programs.

Control programs

The control program is a “stand-alone” software program implemented in the architecture to be tested. It controls the initialization of the computer state entities, the execution of the computer function entities, the interrogation and verification of the resultant computer state entities, and the indication, to the user, of the test results. It is not designed to explicitly verify any portion of the architecture implementation.

The number of failures which may be experienced during execution of the control program is reduced by minimizing the number of different computer functions and computer state entities utilized in implementing the control program.

Selection of the computer entities to be used is based on their simplicity. The simplest entities are usually those whose principles are common to the definition of most computer architectures and are usually included as standard features on the computer. Due to their simplicity and familiarity, these computer entities are less susceptible to design errors and any failures which do occur are more easily defined.

The control program systematically interrogates a computer state entity for each optional computer feature. The results of the interrogation indicate to the control program which optional computer features are installed. This information is used by the control program to determine which tests may be meaningfully executed. The user is also provided the capability to manually select the tests to be executed, overriding the selection of tests made automatically by the control program.

The control program indicates to the user the success or failure of the computer in conforming to its architecture. For most users it is sufficient to provide this indication in a brief message for each installed computer feature. The individuals responsible for defining and correcting a problem will have need to manually select the messages providing more details concerning the problem.

The time for execution of the test is reduced by minimizing the amount of manual intervention required for normal execution and by employing efficient, straightforward coding techniques in designing and implementing the control program. Equally important is the means by which the control program obtains the description of the test to be performed. The information required to describe the test is defined in the next section. The format in which this information is presented to the control program should be carefully selected to advantageously use the properties of the computer entities utilized by the control program in handling the information.

Test cases

A test case is a data record providing information to the control program completely defining a test to be executed by the control program. Each test verifies the action of one computer function entity upon a single combination of all the computer state entities. Consequently, many test cases are required to test the action of all computer function entities upon all combinations of all the computer state entities.

Each test case contains the following information:

1. Computer function entity
2. Computer state entities
   a. initialization values
   b. resultant values
3. Test case execution control information

The computer function entity is usually represented by a single computer instruction with a prescribed beginning and termination. Computer function entities, which are not executions of computer instructions, are represented in the test case by a sequence of one or more computer instructions which initiates execution of the computer function entity.

As indicated earlier, the computer state entities define a particular state of the computer, including contents of main storage and programmable registers. The test case defines the initialization values of the computer state entities for use by the control program in setting the computer to that particular state prior to execution of the computer function entity. The resultant values of the computer state entities are also defined in the test case for use by the control program in verifying that the execution of the computer function entity results in the expected changes in the computer state entities.

The test case execution control information defined in the test case provides a communication link from the individual writing the test case to the control program. When determining which tests are to be executed, the control program must be able to determine if a test case is dependent upon the presence or absence of a particular computer feature. The control program must also be able to determine if a test case is dependent upon a particular computer state entity having a specific value. Changes in the architecture and detection of errors in the test cases sometimes result in changes to the test cases.

The control program must have the capability to indicate, to the user, the change level of the test case being executed. In addition, the control program must have the capability of uniquely identifying each test case from all other test cases. All of the above situations require that specific information be communicated from the individual writing the test case to the control program.

Following are some principles which, if applied during the writing of test cases, tend to improve the overall effectiveness of the test:
• In general, each test case should be independent of the other test cases. This is achieved by initializing all pertinent computer state entities in each test case where they are used. An exception to this principle is the situation where several test cases require large amounts of main storage to be initialized to the same values and execution of the test cases is not expected to change the contents of this main storage.

• The test of the architecture should be as complete as possible. One method of enhancing the completeness of the test is to subdivide the entire architecture into manageable pieces for the purpose of test case writing. Usually the architecture can be satisfactorily subdivided for this purpose into the classifications discussed in the section entitled ARCHITECTURE.

Once the architecture has been subdivided, the completeness of the test can be further enhanced by systematically writing test cases which terminate at each architecturally defined termination point. Execution of these test cases will ensure that the computer correctly detects all architecture violations and all functional entity error conditions.

• The termination of a test case should not depend on the occurrence of any external event. This dependence can be avoided by ensuring that no continuous, never-ending program loops result when using computer instructions to test the computer functional entities.

• It is clear that no computer states which are architecturally “unpredictable” should be included in the architecture verification test which is performed on all the computer models. However, it may be beneficial to test these model dependencies and other implementation peculiarities as an extension to the architecture verification test.

• Usually the number of computer state entities makes it prohibitive to attempt to verify the value of all computer state entities for all possible executions of all computer function entities. Clearly, all computer state entities which are architecturally defined to change or remain constant must be verified. In addition, it may be equally important to verify that the values of certain other computer state entities remain unchanged.

It is clear from the above discussion that the number of test cases included in the architecture verification test will be quite large. It follows that some procedures should be established to control test case development, maintenance, and distribution to the many users.

• The independence of the test cases from each other and from the control program allow several individuals to simultaneously develop the control program and test cases for individual classifications of computer function entities (once the test case format has been defined). Procedures must be established to ensure that test cases are implemented for all classifications of computer function entities.

• Due to the large number of test cases, it is convenient to organize them into a library system. Procedures must be established for adding test cases, deleting test cases, and changing test cases already in the library.

• Many times architecture implementation problems are detected by other programs or other tests. Procedures must be implemented which ensure that a test case is written to adequately test the problem. The test cases resulting from these procedures will improve the completeness of the architecture verification test.

• Changes in the architecture and detection of errors in the test cases sometimes result in changes to the test cases. Procedures must be established which ensure that the library contains the latest level test cases for the latest level architecture.

• It was indicated earlier that the architecture verification test is performed many times by many different users. Procedures should therefore be established for distributing the test to those individuals having a need for it. Magnetic tape and disk are usually satisfactory media for transporting and executing the programs.

Support programs

As previously indicated, the format of the test case, when presented to the control program, should be carefully selected to allow simple and efficient use of it by the control program. This usually means that the test case format is closely associated with the architecture and the internal code of the computer.

Most often this format is not the easiest and most efficient with which an individual can work. Usually a “higher level” format is defined for use by the individuals writing the test cases. A translator program is required to translate the test case definition from the “higher level” format to the format accepted by the control program.

The remaining support programs are library maintenance programs. These programs may take many forms in automating the library maintenance procedures discussed in the previous section.

All of the above support programs are written to execute on a computer whose architecture is not being tested. This procedure has the following advantages:

• Test cases can be developed in parallel with the computer they are to test.

• These programs can use existing computer facilities, such as interactive terminals and data management, to make the jobs of test case development and maintenance simpler and more efficient.
TESTING VIRTUAL STORAGE

Upon investigation of the facilities provided in a virtual storage environment, it would be apparent that most of them are implemented in software. Facilities such as paging, page fixing, resource allocation, and task selection are some examples. Since these functions are implemented in software, they will not be included in the architecture definitions of computer function entities and computer state entities. Therefore, the verification of these facilities is not included in the architecture verification process discussed by this paper.

The virtual storage environment does, however, contain several architecturally defined computer function entities and computer state entities. Some examples of these computer entities are: address translation, page fault detection, associative arrays, page boundary crossings, page referenced indicators, page changed indicators, and computer instructions to control and interrogate the virtual storage environment. All of the architecturally defined computer entities must be tested by the architecture verification process.

This testing is achieved by writing additional test cases for execution with the static environment approach discussed in the section entitled TEST APPROACH.

Test cases must be written to verify correct execution of the virtual storage computer function entities. The test cases previously generated for the basic architecture must be rewritten to include verification of the change in the virtual storage computer state entities and verification that the execution of the address translation entity is transparent in the execution of the basic computer function entities. In addition, test cases must be written to test any "special case" computer function entities which execute differently in the virtual storage environment than they do in the non-virtual storage environment. The virtual storage environment may also give rise to some model dependencies, such as virtual equals real translation, for which it is desirable to write additional test cases.

The section entitled TEST PHILOSOPHY discussed testing in a dynamic environment utilizing many test programs executing under control of a supervisor program. The same section discussed some deficiencies in the thoroughness of the test conducted in the dynamic environment. Most of these deficiencies can be eliminated by writing an additional test program which executes under control of the same supervisor program and utilizes the virtual storage computer entities.

Figure 1 depicts the new, simulator-test program in the dynamic environment. It is referred to as a "simulator-test program" because it actually simulates the architecturally defined computer entities associated with a supervisor or privileged mode of operation. Only those supervisor or privileged mode entities used by the static environment control program are simulated. In addition, this simulator-test program controls the virtual storage address translation entity in such a way that the absolute addresses of the architecture verification test control program are interpreted as virtual addresses and are translated to real addresses in the main storage controlled by the dynamic test supervisor program.

The combination of the two techniques allows the architecture verification test control program to execute in the dynamic environment controlled by the supervisor program as though it were the only program in the computer. The benefits of this arrangement are:

- The completeness of the architecture verification test in the dynamic environment is enhanced by execution of the basic test cases in this environment.
- The need to rewrite the basic test cases for execution in the virtual storage environment has been eliminated by actually executing the basic test cases for execution in the virtual storage environment.
- The concept of multiple virtual memories is verified by concurrent execution of multiple copies of the simulator-test program.

The following limitations imposed by the changes in the execution environment restrict which of the basic test cases can be correctly and meaningfully executed in the dynamic environment:

- Test cases depending upon specific timing considerations cannot be executed in the dynamic environment because of the interactions with the other test programs.
• Test cases requiring manual intervention may not be executable due to limitations in the manual operation-simulator-test program interface.
• It is not meaningful to execute test cases employing supervisor or privileged mode operations since these operations are simulated by the simulator-test program and do not result in an actual test of the architecture implementation.

TEST COMPLETENESS

One of the test objectives identified in the section entitled TEST OBJECTIVES is thoroughness. Some procedures have been discussed which can help ensure a thorough test. This section discusses an approach for measuring the completeness of an architecture verification test as applied to a microprogrammed computer.

In a microprogrammed computer, most of the external specifications of the architecture are implemented in microcode. As a result, some software debugging techniques can be employed. Specifically, a microcode trace could be obtained, using some hardware or software tool, and the results analyzed to determine which paths through the microcode have been exercised and which have not. However, the number of possible paths through the microcode and the number of test cases which would have to be individually traced make this approach prohibitive.

An alternative is to record an indication of which microinstructions have been executed and which have not. Additionally, an indication of which microinstruction branches have been executed can be recorded. Since only the total number of microinstructions and branches executed is of importance (and not the order in which they were executed), a separate record is not required for each test case. Only a composite record reflecting the execution of the entire set of test cases is required. However, for informational purposes, separate records of the static and dynamic environments may be desirable.

The completeness of the architecture verification test is then expressed as the percentage of the total number of microinstructions which were executed and the percentage of the total number of microinstruction branches which were taken. In analyzing the results of completeness verification, two pitfalls should be avoided:

• The total percentage coverage figures can overshadow the fact that one or more computer function entities may not be tested at all.
• An individual writing test case to test a specific computer function entity may desire to know what the completeness of his test case development activity is at a given point in time. For the percentages of a partial completeness report to be meaningful, it should be ensured that they are based on the number of microinstructions and branches which the test cases are designed to test.

With some knowledge of the implementation of the architecture and the information as to which microinstructions and branches have been covered, the individual writing test cases can determine which test cases are required to fill the holes in the test. Discretion should be used here, for the effort involved in generating test cases usually follows the law of diminishing returns. That is, it generally will take more effort per microinstruction to move from a 90 percent to a 95 percent coverage than it will to move from a 50 percent to a 75 percent coverage.

SUMMARY

This paper has described in general terms the techniques for implementing a basic architecture verification process and applying that process to a virtual storage architecture. This will allow the reader to mold the techniques and apply them to a given, specific architecture. The exposures and principles indicated throughout the paper will aid the reader in implementing a thorough and efficient test.

ACKNOWLEDGMENTS

The author acknowledges Mr. F. W. Ball, Mr. E. M. Shimp, and Mr. D. A. Shiner as the initial creators of the static test approach. Numerous other individuals have used both the static and dynamic approaches. Their discussions with the author have resulted in the approaches defined in the paper.

Especially appreciated were the efforts of my coworkers in patiently and diligently reviewing and critiquing this paper.

From the collection of the Computer History Museum (www.computerhistory.org)