A tool-based approach for software testing and validation

by J.C. HUANG, PETER VALDES, and RAYMOND T. YEH

International Software Systems, Inc.
College Park, Maryland

ABSTRACT

This paper describes a methodology for software testing and validation. By recognizing that there are several major error types, this methodology uses different test strategies to expose a particular type of error. To facilitate these strategies, specific tools are needed. This paper not only identifies the desired tools, but also discusses the design concepts behind various tools as they have been built at International Software Systems, Inc. (ISSI).
INTRODUCTION

A number of software tool systems (e.g., RXVP80\textsuperscript{2}, TOOLPACK\textsuperscript{3}, and MAP\textsuperscript{4}) have been developed recently; however, none provides an effective testing methodology that facilitates exposure of software errors during testing and maintenance. These tool systems implicitly assume that traditional methods for software testing (e.g., coverage-based testing,\textsuperscript{1,7,10} functional testing,\textsuperscript{1,8} boundary value testing,\textsuperscript{14} mutation testing,\textsuperscript{1} and domain testing\textsuperscript{18}) are used in the actual process of exposing errors in programs. It is well recognized that none of these traditional testing strategies is powerful enough to expose all the possible errors in a program.\textsuperscript{1,6} The best that can be hoped for is to use a specific test strategy to expose a specific error.

We have recognized the limitations of existing test strategies and testing tools and have attempted to address these problems in our SEQUEL (Software Quality Evaluation Laboratory) tool system. SEQUEL is a testing tool system whose basic objective is to increase the quality of software and productivity of software engineers during the software development process by providing a methodology for software testing plus the tools to support the said methodology. We are currently implementing SEQUEL to accept ISO (International Standards Organization) PASCAL under VAX/VMS.

The testing methodology of SEQUEL will be discussed in the next section. In the remaining sections, design concepts behind various tools will be discussed and their use illustrated with examples.

TESTING METHODOLOGY IN SEQUEL

The main purpose of testing in the software development life cycle is to verify conformance of the software with respect to its intended requirements. The intended requirements include the following:

1. System requirements developed prior to software design.
2. Functional requirements developed during software design. This category may very well include error condition requirements as well as boundary condition requirements.
3. Programming requirements developed after software design. This include syntax, semantic, compiler, and hardware restriction requirements.

Even if the policy is to always do things right the first time (e.g., the cleanroom idea), testing can never be eliminated from the software development life cycle. As long as software is developed by human beings, there is always a need to demonstrate that the software conforms with its requirements.

Any nonconformance of a given software with its intended requirements is known as a software error. An important prerequisite, therefore, in exposing software errors is a clear statement of software requirements (possible written in a requirements specification language). Testing to verify conformance with software requirements is really equivalent to testing a hypothesis that a given software error does/does not, in fact, exist in the software. A testing methodology that hypothesizes and tests on all possible errors in a software addresses the original objective of software testing.

Selecting an appropriate testing strategy for a given software error is still an art. Researchers have yet to collect data on software errors that frequently occur in a given environment and map these errors with the appropriate test strategies. However, it has been well recognized in the field that implementation of a testing strategy is greatly facilitated if it is supported by software tools. A very good example is the compiler, which is effective in exposing syntax errors and sometimes a few semantic errors. Another example is a debugger, which facilitates detection of software faults. The use of software tools not only enhances the error detecting capability of a testing strategy, but it can also be cost effective. The savings in using tools is due to (1) reducing the amount of time (and therefore cost) to expose any embedded software error and (2) reducing the amount of time needed to find the cause of the exposed error. SEQUEL addresses the need for a tool-based testing methodology.

Methodology

The methodology can be formulated in the following way:

\[ \text{Let} \]

\[ [SR_1, SR_2, \ldots, SR_i, \ldots, SR_N]: \] Set of software requirements collected at various phases of the software development life cycle.

\[ e(SR_i): \] Error associated with software that does not conform to \( SR_i \).

\[ E: \] Set of all errors (initially unknown) that actually exist in the software. \[ E = \bigcup_{j=1}^{N} e(SR_j) \]
**Basic Method**

Test of Error Hypothesis (for each SR<sub>j</sub>)
Null Hypothesis H<sub>0</sub>: e[SR<sub>j</sub>] in E
Alternative Hypothesis is H<sub>1</sub>: e[SR<sub>j</sub>] Not in E

In software testing, we hypothesize the existence of specific software errors (nonconformance with requirements) embedded in the software. The error hypothesis is then tested by a strategy appropriate for the given error. The test result may lead to acceptance of H<sub>0</sub> and rejection of H<sub>1</sub> or vice versa. Not all testing strategies can be recommended for a given e[SR<sub>j</sub>]. Some are imperfect relative to e[SR<sub>j</sub>] (for example, using path testing to expose boundary errors); others are nearly perfect with respect to e[SR<sub>j</sub>]. It is imperative, therefore, that an appropriate testing strategy be properly selected to minimize the error of rejecting H<sub>0</sub> when, in fact, it is true.

Nonconformance with software requirements can take many forms. The following are the more frequently occurring errors:

1. Nonconformance with compiler rules/restrictions
   = syntax/semantic errors
2. Nonconformance with intended functions
   = logic/computation errors
3. Nonconformance with erroneous input
   = error handling errors
4. Nonconformance with proper program boundaries
   = boundary condition errors
5. Nonconformance with proper data flow
   = data flow anomalies

An important assumption of the methodology is that the software requirements are clearly stated.

An obvious example using this hypothesize-and-test approach is the detection of syntax and some semantic errors in programs.

**Error Hypothesis**

H<sub>0</sub>: Syntax or semantic error in E
H<sub>1</sub>: Syntax or semantic error not in E

**Strategy**

1. Compile and check for syntax/semantic error. Quit if none. Tool: Compiler

The creative energy of a programmer should not be wasted in manually exposing syntax/semantic errors (as, for example, in a code walkthrough). Compilers are very good at this, and they should do the job.

The more interesting types of software errors that one would like to expose are those that remain after successful compilation. Following the basic methodology discussed above, we hypothesize on the existence of each of these error types in a program (being tested) and specify a specific strategy to test each hypothesis. SEQUEL specifies (in its current form) test strategies for logic/computation, data flow and boundary/error condition errors. These strategies (see the next two sections) are supported by the following basic tools:

1. Program Attribute Generator
2. Static Reports Generator
3. Program Instrumenter
4. Branch Coverage Counter
5. Symbolic Trace Generator
6. Symbolic Trace Data Flow Analyzer
7. Symbolic Trace Slicer
8. Symbolic Trace Analyzer

It is not claimed that these are the only tools one would need to fully support any testing strategy. There are certainly a lot more tools one would desire to have (especially for integration testing and concurrent program testing). Some of these tools, it is hoped, will be included in future versions of SEQUEL. The role of SEQUEL tools should be emphasized:

1. These tools only support the overall testing methodology. They indirectly aid the programmer in detecting and removing software errors.
2. These tools should complement other existing tools—e.g., compilers and debuggers—and should not compete with them.

**Test Strategy for Logic/Computation Errors**

A computation error occurs when the set of computational statements (usually assignment statements) directly affecting a program output variable does not conform to requirements. On the other hand, a logic error occurs when the set of control statements and all other statements affecting the control statements cause traversal of an incorrect path in the program.

**Error Hypothesis**

H<sub>0</sub>: Logic/computation error in E
H<sub>1</sub>: Logic/computation error not in E

**Strategy I**

1. Generate a test case to exercise the intended software subfunctional requirement. A subfunctional requirement maps to a single program path. A set of subfunctional requirements may be contained in a specified functional requirement; hence, a corresponding set of test cases should be generated:
   a. Get the input conditions that invoke the functional requirement. This should be found from the specified software requirements.
   b. Pick an interior element that satisfies the intended function's input conditions.
   c. Check untraversed branches as (possible) guides in generating the next test case.
   Tool: Branch coverage counter
2. Produce a compile clean program (if program has previously been changed).
   Tool: Compiler
3. Generate the program's attributes and sequenced program listing.
   Tool: Static Analyzer
4. Instrument the sequenced program.
   Tool: Program Instrumenter tool of Dynamic Analyzer
5. Generate a symbolic trace of the path traversed by running the test case generated in Step 1.
   Tool: Symbolic Trace Generator of Dynamic Analyzer
6. Check for data flow anomalies on the generated symbolic trace (if desired)
   Tool: Data Flow Analyzer of Dynamic Analyzer
   Data Flow Anomaly = [Referencing an undefined variable; Not referencing a defined variable; Defining a currently defined variable]
   Note: This step can be done separately if the sole intention is to find symbolic trace data flow errors.
7. Conditional on the complexity of the symbolic trace, slice the trace to focus attention of the sublogic/subcomputational part corresponding to a suspected erroneous variable in the trace.
   Tool: Symbolic Trace Slicer
8. (Optional depending on specific situation)
   Generate the backward-substituted predicates of the slice/trace.
   Tool: Symbolic Trace Analyzer
   There are situations where the set of backward-substituted predicates are easier to compare with the specified functional requirements. A sample situation occurs when the symbolic trace is mostly composed of logic statements (control statements and other statements affecting control).
9. Compare the slices or the trace with the specified functional requirements. The specified functional requirements may be in mathematical/symbolic form or in English-prose form. The programmer/tester detects any logical discrepancies in the slice/trace and in the specified functional requirements.

It should be emphasized that the role of the test cases in Step 1 was simply to generate a symbolic trace and not to expose a logic/computation error directly. Any exposed error from the test case is only coincidental. The logic errors are detected after comparing the slice/trace with the software functional requirement. Exposing logic/computation errors directly from test cases can be difficult and time-consuming. Exposing logic errors by comparing slice/trace with functional requirements minimizes the difficult task of generating a lot of test cases and the task of comparing the test results with the expected software result. It has to be pointed out that in comparing a slice/trace with functional requirements when debugging a program, we may need information on certain attributes of the program to verify, for example, mixed mode computations or calling sequence errors. This information may be queried from the program database or by invoking any of the following tools of the static analyzer: the Variable/Statement Cross Reference Table Generator, the Sub-Program Calling Sequence Table Generator, and the Sub-Program Cross Reference Table Generator.

Test Strategy for Boundary/Error Condition Errors

The declared range of input variables in a program plus its various predicates to control logic define the boundaries of the program. A boundary condition error occurs when an input point in the boundary yields results that do not conform with intended requirements. An error condition error occurs when an input point outside the legal boundary of the program is not handled properly or causes the program to crash.

Error Hypothesis

H0: Boundary/error condition error not in E
H1: Boundary/error condition error not in E

Strategy II

1. Start from a previously generated trace/slice. This may require doing Steps 1 to 6 of testing strategy recommended for logic/computation errors.
   Tools needed:
   a. Compiler
   b. Static Analyzer
   c. Program Instrumenter
   d. Symbolic Trace Generator
   e. Symbolic Trace Slicer
   f. Data Flow Analyzer
2. Generate equivalence class conditions for the selected trace/slice. Quit if no more trace/slice.
   The equivalence class conditions are generated by performing backward substitution on the predicates of the trace/slice. The backward-substituted predicates are expressed purely in terms of constants and input variables. The predicates essentially define the boundaries of the subfunction being implemented by the trace/slice.
3. Compare the generated (possibly erroneous) equivalence class conditions with the specified boundary/error requirements. Any observed discrepancy (due to incorrect or missing boundary) is a boundary condition error.
4. An alternative or complementary step is to generate a test case to test the boundary/error conditions of the selected trace/slice. The equivalence class conditions expressed purely in terms of constants and input variables greatly facilitate generating these test cases. The equivalence class conditions may be simplified symbolically using a text editor. Simplification may be necessary to further facilitate test case generation.
   Guideline:
   For each suspected erroneous boundary, generate test cases near the predicate’s boundary. Test cases should be immediately inside and outside the boundary. Test inputs immediately outside the boundary should be
properly handled by the program. Test cases in the interior of the equivalence class do not really yield additional useful information. They only duplicate what the previous test case did. In a way, the set of backward-substituted predicates serves to filter out redundant test cases.

5. Execute the generated test cases and observe any erroneous program output. Test cases outside the boundary that are also illegal/invalid program inputs should not cause the program to crash. They should be properly handled by appropriate error condition routines. Test cases that are outside the boundary but that are valid program inputs should be processed by the appropriate program path. Finally, the test output should be compared with expected program output.

6. Remove any detected boundary/error condition errors. Go back to 1.

Notice that executing a test case outside the boundary of a predicate (hence, outside the boundaries of the equivalence class) traverses a different path in the program. A new path implies a new symbolic trace/slice. This new trace/slice can be compared with specified requirements for purposes of error detection and may serve as the next trace/slice to be analyzed.

SYMBOLIC TRACE SLICER

Why a Symbolic Trace Should Be Sliced

The traditional approach to program slicing is to extract the smallest possible independently executable subprogram from a given program and slice criterion, which behaves equivalently with the given program as far as the variables in the slice criterion are concerned. There are, however, difficulties in following this traditional approach:

1. Treatment of array and record elements during the slicing process. Current slicing algorithms treat the whole array or record as a scalar. This assumption would obviously collect more statements in the slice than necessary.

2. Treatment of functions and procedures (subprograms) in the slicing process.

We used a different approach for SEQUEL to solve these difficulties. The approach is to slice the symbolic trace generated by the Dynamic Analyzer instead of slicing the original program. The advantages of the approach are as follows:

1. There is a need to deal only with a single path (trace) in the program. This facilitates treatment of functions and procedures.

2. The specific array or record elements are known as a result of dynamically generating a trace from a test input. Thus, array and records need no longer be treated as scalars.

This approach does not in any way diminish the error-detecting capability of SEQUEL. SEQUEL’s testing methodology always deals with a trace/slice in exposing program errors. Therefore, there is really no difference between slicing the program first, and then generating a symbolic trace from the slice; and generating a symbolic trace first, and then slicing the symbolic trace.

The main purpose, of course, in extracting a slice is to focus attention on the variables (possibly erroneous) in the slice criterion. This enhances error detection and facilitates finding causes of exposed errors. An example would best illustrate our point.

Command Processor in SEQUEL

The command processor in SEQUEL will integrate all the testing strategies described into one overall testing strategy. It has the following basic form:

1. User invokes strategy to remove syntax/semantic errors from the program. Correct any detected errors.
2. Hypothesize an error embedded in the software. Quit if no more errors to hypothesize.
3. CASES
   a. Logic/Computation Error: Invoke Strategy I
   b. Boundary/Error Condition Error: Invoke Strategy II

The basic flow of the testing methodology has the following features:

a. It is easy to integrate new test strategies and tools in the future.

b. The user has flexibility to hypothesize the more important errors in the program first. This may be critical when testing time/resources are limited.

c. The program to be tested may be a single module, a set of modules, or the whole program. Testing single modules or a set of modules in a bottom-up or top-down fashion may require a driver and a set of stubs. Drivers and stubs are necessary to make the module separately executable. We thus have a uniform approach for unit, integration, and system testing.

We recognize that additional features should be integrated in the command processor for it to be user-friendly. The following features are being implemented:

1. Menu-driven user interface.
2. HELP routines to
   a. Guide the user on how to use the package.
   b. Recommend the appropriate tool to use at a given point in the testing process.
3. Ability to invoke system tools (e.g., compiler or text editor) inside the processor.
4. Ability to save and recall input/output files. This can be useful, for example, in these situations:
   a. Ability to save and timestamp test cases run on the program being tested.
   b. Ability to recall previous test cases for further analysis.
5. Ability to gather and document error statistics on program being tested.
Example/Results Interpretation

Given
PASCAL program which finds the maximum and minimum value in an array. (Figure 1 gives a sample program.)

Test Case #1

N = 5
Array A: 4,3,1,2,5

Figure 2 is the symbolic trace generated by Test Case #1.

Note: Column 1 gives the Symbolic Trace Sequence Numbers (ST#); Column 2 gives the Sequenced Program listing sequence numbers (SPL#). generated by the Static Analyzer.

Suppose we wish to focus our attention on whether the program, in fact, correctly computes the maximum of the array given Test Case #1. We have to note that one of the properties of Test Case #1 is that the maximum element lies at the end of the array. Thus this particular test case is exploring the correctness of the program in the case when the maximum lies at the end of the array with an odd number of elements. It does not test the correctness of the program when the maximum element is inside or at the beginning of the array, or at the end of an array with an even number of elements.

Slicing Criterion:
Symbolic Trace Sequence #: 19
Variable(s): MAX

Symbolic Trace Slice (Option 1):
The I/O statements were excluded to make the example short. The STS #’s also started (in this example) after the I/O statements.

STS #  SPL #  Statements
 3     16  I := 2
 9     33  I := I + 2
13     29  MAX := A[I + 1]
19     41  WRITELN (MAX, MIN)

Remember that Slice Option 1 extracts only computational statements directly affecting the variable(s) in the criterion (MAX in this case), which contributed to the final value of MAX. If the output value of MAX is incorrect, then the cause can easily be detected by looking at the statements in the slice. The debugging process is thus facilitated. Two major causes are possible if MAX is incorrect:

1. At least one of the statements in the outputted slice is erroneous (e.g., wrong arithmetic statement, mixed-mode computation, referencing an undefined variable, etc.)
2. One or more statement in the slice is missing.

In Figure 3, MAX is correctly computed.

<table>
<thead>
<tr>
<th>Code:</th>
<th>Program minmax (input, output);</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Var i, n, min, max : integer;</td>
</tr>
<tr>
<td></td>
<td>a : array[1..10] of integer:</td>
</tr>
<tr>
<td>1</td>
<td>Begin</td>
</tr>
<tr>
<td>2</td>
<td>writeln(&quot; no. of elements in the array = &quot;);</td>
</tr>
<tr>
<td>3</td>
<td>read(n);</td>
</tr>
<tr>
<td>4</td>
<td>For i = 1 to n DO</td>
</tr>
<tr>
<td>5</td>
<td>Begin</td>
</tr>
<tr>
<td>6</td>
<td>read(a[i]);</td>
</tr>
<tr>
<td>7</td>
<td>writeln(a[i]);</td>
</tr>
<tr>
<td>8</td>
<td>i := i + 2</td>
</tr>
<tr>
<td>9</td>
<td>End</td>
</tr>
<tr>
<td>10</td>
<td>End</td>
</tr>
<tr>
<td>11</td>
<td>End</td>
</tr>
<tr>
<td>12</td>
<td>End</td>
</tr>
<tr>
<td>13</td>
<td>End</td>
</tr>
<tr>
<td>14</td>
<td>End</td>
</tr>
<tr>
<td>15</td>
<td>End</td>
</tr>
<tr>
<td>16</td>
<td>End</td>
</tr>
<tr>
<td>17</td>
<td>End</td>
</tr>
<tr>
<td>18</td>
<td>End</td>
</tr>
<tr>
<td>19</td>
<td>End</td>
</tr>
<tr>
<td>20</td>
<td>End</td>
</tr>
<tr>
<td>21</td>
<td>End</td>
</tr>
<tr>
<td>22</td>
<td>End</td>
</tr>
<tr>
<td>23</td>
<td>End</td>
</tr>
<tr>
<td>24</td>
<td>End</td>
</tr>
<tr>
<td>25</td>
<td>End</td>
</tr>
<tr>
<td>26</td>
<td>End</td>
</tr>
<tr>
<td>27</td>
<td>End</td>
</tr>
<tr>
<td>28</td>
<td>End</td>
</tr>
<tr>
<td>29</td>
<td>End</td>
</tr>
<tr>
<td>30</td>
<td>End</td>
</tr>
<tr>
<td>31</td>
<td>End</td>
</tr>
<tr>
<td>32</td>
<td>End</td>
</tr>
<tr>
<td>33</td>
<td>End</td>
</tr>
<tr>
<td>34</td>
<td>End</td>
</tr>
<tr>
<td>35</td>
<td>End</td>
</tr>
<tr>
<td>36</td>
<td>End</td>
</tr>
<tr>
<td>37</td>
<td>End</td>
</tr>
<tr>
<td>38</td>
<td>End</td>
</tr>
<tr>
<td>39</td>
<td>End</td>
</tr>
<tr>
<td>40</td>
<td>End</td>
</tr>
<tr>
<td>41</td>
<td>writeln(max, min)</td>
</tr>
<tr>
<td>42</td>
<td>End</td>
</tr>
</tbody>
</table>
418 National Computer Conference, 1984

**SYMBOLIC TRACE ANALYZER**

**Purpose**

The main function of the Symbolic Trace Analyzer tool is to perform backward substitution on the predicates of a trace or slice. The end result of this process is the same set of predicates of the given slice/trace, with the difference that all are now expressed purely in terms of input variables and program constants. The conjunction of all the predicates defines the symbolic trace. The logic looks reasonable (based on our requirements), and we can infer that given an input with the same properties of test case #1, the program is logically correct. An incorrect logic would easily show in the slice. It may be an incorrect predicate, a missing statement, or a statement that should be removed. The location of the fault is facilitated by referring to the sequence program listing numbers.

Option 2 generates, in addition to the computational statements directly affecting MAX, the logic ingredient that went into the traversal of this particular program path. The logic tells us that

1. MAX is initialized to A[1]
6. Terminates when I > N

The logic looks reasonable (based on our requirements), and we can infer that given an input with the same properties of test case #1, the program is logically correct. An incorrect logic would easily show in the slice. It may be an incorrect predicate, a missing statement, or a statement that should be removed. The location of the fault is facilitated by referring to the sequence program listing numbers.

**EXAMPLE**

Let us use the Symbolic Trace Slice (Option 2) given in the section entitled "Example/Results Interpretation."

1. The predicate statements are located in STS #’s 4, 5, 6, 11, 12, 16, 17, and 18.
2. Start Backward substitution with predicate #4, getting (2 < N). We then continue with predicate #5 and so on, until we finish predicate #18.
3. Figure 4 shows the generated set of backward-substituted predicates:

<table>
<thead>
<tr>
<th>Predicate STS#</th>
<th>Backward Substituted Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>(2 &lt; N)</td>
</tr>
<tr>
<td>10</td>
<td>(I+2) &lt; N</td>
</tr>
<tr>
<td>16</td>
<td>NOT ((I+2+2) &lt; N)</td>
</tr>
<tr>
<td>17</td>
<td>NOT ((I+2+2) = N)</td>
</tr>
<tr>
<td>18</td>
<td>NOT (A[N] &gt; A[I+2+2+1])</td>
</tr>
</tbody>
</table>

Figure 4

Predicates 4, 10, 16, and 17 imply that N (an integer) should be equal to 5. Any other value of N will violate one or more of these predicates, causing a program boundary error. We may opt to use a text editor and simplify the predicates in the following form:
Predicates 5, 6, 11, 12, and 18 collectively show the logic that caused the extraction of MAX. A total of five comparisons were used. At this point we can design test cases that explore the boundaries of the path. These test cases would have one or more of the following properties:


This information guides us in designing further test cases that we hope will explore more logic and boundary errors in the program.

**STATIC ANALYSIS**

The main objective of static analysis (as well as dynamic analysis, discussed in the following section) is to determine that a given computer program has certain properties. To determine whether the program has a certain property, we need first to identify attributes that reflect the quality in question and then to devise an effective method for computing the values of the attributes. Generally speaking, there are two main types of program attributes. The first type consists of those associated with components of a program, whereas the second type consists of those associated with points in the control flow. To be more specific, let us consider the fragment of a flowchart depicted in Figure 6. In this figure, \( s \) represents a statement or a program segment, and \( i \) and \( j \) identify points in the control flow. \( A[i] \) and \( A[j] \) denote the sets of attributes associated with the corresponding points; \( B[s] \) denotes the set of attributes associated with program component \( s \). In general, the value of \( B[s] \) (i.e., the attribute of the first type) will not be affected by an execution of \( s \). However, an execution of \( s \) may cause a change in the value of \( A[j] \) (i.e., the attribute of the second type). Furthermore, \( A'[j] \), the new value of \( A[j] \) upon an execution of \( s \), can be computed on the basis of \( A[i] \), \( B[s] \), and the old value of \( A[j] \). To put it formally,

\[
A'[j] = f(A[i], B[s], A[j])
\]

where \( f \) is some function.

For convenience we shall refer to the attributes associated with program components as B-attributes and the attributes associated with points in control flow as A-attributes. A B-attribute is a local and static attribute whose value can be obtained from the associated program component. The value of a B-attribute will not be altered by an execution. An A-attribute is a global and dynamic attribute whose value can be computed on the basis of local attributes and the attributes associated with other points in control flow. The values of A-attributes may be altered in an execution. In most cases it is the values of A-attributes that reflect on the quality in question.

The above concept clearly suggests a unified approach to the problems of program analysis and validation as outlined below:

1. Identify the A-attributes that directly or indirectly reflect the quality in question and the B-attributes that are required in computing the values of the A-attributes.
2. Identify the relations among the attributes.
3. Use the result of Step 2 to devise an effective algorithm for computing the values of the A-attributes.

The values of the B-attributes and the initial values of the A-attributes are obtained by systematically parsing the program text. These extracted values are then stored in a program database. This generated program database forms the central part of the system and allows implementation of various error type specific test strategies in a single software testing tool system. Thus, the program database allows the user to query program attribute information relevant at any point in the testing process and allows building of other SEQUEL tools without reparsing the program.

**DYNAMIC ANALYSIS**

SEQUEL performs dynamic analysis on a program through instrumentation. Program instrumentation is the process of inserting additional code statements at proper locations in the
program to compute the values of program attributes. The objectives of dynamic analysis are as follows:

1. To generate symbolic trace of the program path traversed by the submitted input
2. To update branch coverage counters (i.e., number of times each program branch is traversed)
3. To detect data flow anomalies in the path traversed

Detection of data flow anomaly by means of instrumentation\textsuperscript{10} is a unique feature of this tool system, can be briefly explained as follows.

It is observed that, in program execution, a statement may act on a variable (datum) in three different ways: define, reference, and undefine. A variable is defined in a statement if an execution of the statement assigns a value to the variable. A variable is referenced in a statement if an execution of the statement assigns a value to the variable. A variable may become undefined in many circumstances. For example, in a FORTRAN program, the index variable of a DO statement becomes undefined when the RETURN statement is executed. Also, if a program is written in a language that allows block structure, the local variables of a block may become undefined when control exits from the block.

A sequence of actions may be taken on a variable in a program being executed. A reference to a variable constitutes a programming error unless it is to be referenced (i.e., its value to be used) later. Therefore, if we find that a variable in a program is (1) undefined and then referenced, (2) defined and then undefined (not referenced), or (3) defined and then defined again, then we may reasonably conclude that a programming error might have been committed. This idea has been used by Fosdick and Osterweil\textsuperscript{20} to detect programming errors.

The three types of data flow anomalies mentioned above can be detected by means of static analysis, as suggested by Fosdick and Osterweil.\textsuperscript{20} However, the method has some inherent limitations.\textsuperscript{12}

The following presents a new method for detecting data flow anomalies by means of program instrumentation. For this purpose, it is useful to regard a variable as being in one of the four possible states during program execution. The four possible states are state U: undefined, state D: defined but not referenced, state R: defined and referenced, and state A: abnormal state. For error detection purposes it is proper to assume that a variable is in the state of being undefined when it is declared implicitly or explicitly. Now if the action taken on this variable is "define," then it will enter the state of being defined but not referenced. Then, depending on the next action taken on this variable, it will assume a different state, as shown in the state transition table (Figure 7).

Note that in Figure 7 d, r, and u stand for "define," "reference," and "undefine," respectively. The three types of data flow anomalies mentioned previously can thus be denoted by ur, du, and dd in this shorthand notation. It is easy to verify that, if a sequence of actions taken on the variable contains either ur, du, or dd as a subsequence, the variable will enter state A, which indicated the presence of a data flow anomaly in the execution path. We let the variable remain in state A once that state is entered. Its implication and possible alternatives will be discussed later.

It is obvious from the above discussion that there is no need to compute the sequence of actions taken on a variable along the entire execution path. Instead, we need only to know if the sequence will contain ur, du, or dd as a subsequence. Since such a subsequence will invariably cause the variable to enter state A, all we need to do is to monitor the states assumed by the variable during execution. This can be readily accomplished by means of program instrumentation.

To see how this can be done, let us consider a fragment of a flowchart, shown in Figure 8. Suppose we wish to detect data flow anomalies with respect to variable, say, x. If s is in state q before statement S is executed, and if a is the sequence of actions that will be taken on x by S, then an execution of S will cause x to enter state q' as depicted above. Given q and a, q' can be determined on the basis of the state table given previously. However, for the discussions that follow, it is convenient to write

\[ q' = f(q, a) \]

where \( f \) is called the state transition function and is completely defined by the state table given above. Thus, for example, \( f(U, d) = D, f(D, u) = A \). For the cases where a is a sequence

\[
\begin{array}{c|c|c|c}
\text{state} & \text{action=d} & \text{action=r} & \text{action=u} \\
\hline
U & D & A & U \\
D & A & R & A \\
R & D & R & U \\
\end{array}
\]
or more than one action, the definition of \( f \) can be naturally extended. For example, \( f(U, d) = f((U, d), u) = f(D, u) = f(D, u), r) = f(A, r) = A \).

Note that in this case \( a \) is the B-attribute associated with \( S \), and \( q \) and \( q' \) are the A-attributes associated with the respective control points.

Next, we observe that the computation specified by \( q' := f(q, a) \) can be carried out by using a program statement of the form:

\[
q := f(q, a).
\]

Now if we insert the above statement next to statement \( S \) as shown below, then the new state assumed by \( x \) will be automatically computed upon an execution. The augmented program depicted here is said to have been instrumented with the statement \( q := f(q, a) \). This statement should be constructed in such a way that there will be no interference between this inserted statement and the original program. A simple way to accomplish this is to use variables other than those appearing in the program to construct the inserted statement.

In practice, it is more appropriate to instrument the program with procedure calls instead of assignment statements. The use of a procedure allows us to save the identification of an instrument as well as the state assumed by the variable in question. Thus the programmer will be able to tell the exact location as well as the type of data flow anomaly detected. This greatly facilitates anomaly analysis.

CONCLUSION

A tool-based approach for testing and validating software has been described in this paper. The approach specifies error-specific test strategies for path logic/computation and boundary errors. These are the two major error types that remain after the successful compilation of a program. For the application of the approach to be cost effective, the specified test strategies are supported by software tools. This paper also describes the concepts behind the design of these software tools.

The testing methodology and supporting software tools provide a number of unique features:

1. The methodology allows the user to focus attention on exposing software errors in a specific program path belonging to a specific error type.
2. The user can further focus on the sublogic of a generated symbolic path through program path slicing. Slicing is a powerful approach for testing and validating the correctness of a program with respect to logic/computation errors.
3. The user can generate the boundary conditions of a given path through predicate backward substitution. This facilitates design of boundary value test cases for exposing path boundary errors.
4. Path data flow analysis is done through program instrumentation.

ISSI has implemented this approach in its SEQUEL project for testing and validating ISO PASCAL programs.

REFERENCES
