Experiences in building and using compiler validation systems

by PAUL OLIVER
E.D.S. Federal Corporation
Washington, D.C.

INTRODUCTION

Software Validation

For purposes of this discussion, the term "software validation" refers to the process of testing a completed software product in its operational environment. The scope of this paper is further limited in the following ways: the experiences described are not applicable to applications software; the validation systems discussed must be capable of functioning on a variety of dissimilar hardware and operating systems; the staff performing the validation is not involved in the development or the maintenance of the products being tested; and the result of a validation could impact the eligibility of the product for procurement. This environment imposes unusual and stringent requirements on the portability of the validation systems and the auditability of the validation.

The methodologies available for software validation are few. Design simulation can and is being used, concurrently with software system design, to evaluate the effects of various system requirements or design alternatives once the system has been defined and modeled. Problem statement languages provide a formal syntax and semantics with which to communicate needs of users to analysts, and are applicable during the system definition and design phases of a software development effort. Neither design simulation nor problem statement languages are applicable to the testing of a completed product. The experiences of the Federal COBOL Compiler Testing Service (FCCTS) suggest that functional testing is the most thorough technique presently available for testing a completed software product.

Functional Testing

Functional testing is the process of executing a series of generally independent tests designed to exercise the various functional features of a software product. The examples discussed in this paper will address the testing of compilers, but the findings and conclusions are applicable to system software in general. The exclusion of application software is not meant to suggest that such software is not amenable to functional testing, but is, rather, a reflection of the general inadequacy of specifications for applications software. It is difficult to test for the valid implementation of functions when the functions themselves are ill-defined. This is of course a subjective observation, and it is also true that the specifications of a compiler's architecture (generally, the standard) often leave much to be desired.

Functional testing can be used to test characteristics of software such as performance and integrity, but is most commonly used for specifications testing. Thorough functional testing requires a complete test plan, systematic control of the testing effort, and objective measurements of test coverage. Functional testing is applicable during implementation (verification), during evaluation (validation), and during the maintenance phase of a software product's life-cycle. This applicability to all phases of test activities is a principal advantage of functional testing. Furthermore, the thoroughness of testing is measurable in terms of number of functions tested, and the revision and evaluation of test specifications is relatively simple. Also, functional testing offers a high degree of visibility to a customer, and is apt to be well understood by that customer.

With these advantages come some disadvantages. The one most frequently cited is that it is generally not possible to assure that all possible features or decision points of a software product are in fact tested. With regard to compilers it is certainly true that it is not practical to test all possible combinations of language components and data types. We have not, however, found this to be a serious shortcoming, since it is certainly possible to test all reasonable combinations. What is "reasonable" is admittedly a subjective judgment, but such subjectivity regarding test limits is hardly unique to software testing. A more serious problem, from the FCCTS experience, is that functional testing can only be as good as the specifications being tested. Thus, it is an unfortunate fact that many important features of a compiler cannot be tested because the pertinent language specifications are ambiguous.

THE FCCTS EXPERIENCE

The Federal COBOL Compiler Testing Service has, during the years 1973-1978, performed official validations of over four dozen COBOL and FORTRAN compilers, in
Validation of FORTRAN and COBOL Compilers

The importance and utility of higher-level languages is, in this writer's opinion, no longer at issue in data processing. The steady decline in hardware costs, a growing awareness of the importance of programmer productivity and software reliability and maintainability, and improvements in the quality of code produced by compilers have nearly terminated arguments regarding the relative effectiveness of lower- and higher-level languages.

This trend has made the compiler the most visible component of system software. To many programmers, the compiler is the system, since it is the tool which is most frequently used in building a piece of software. It is therefore important that the compiler conform to its specification. For some compilers, most notably COBOL, FORTRAN, and PL/I, these specifications (at the functional or architectural level) are embodied in a standard.

The software development manager is faced with a multitude of problems. The productivity of his average programmer ranges from three to nine lines of code per hour, and has been increasing at a rate of only about 3 percent per year. Furthermore, the quality of his product is not too good, and he may require as much as 75 percent of his resources just to maintain the product once it is developed. These are not very encouraging figures. Thus, it is paramount that the manager, and his programmers, at least be able to have some faith in the tools of their trade. In particular, it is not unreasonable to expect that a compiler conform to existing standards in its translation of programs written in standardized languages. There are of course limits to how much a standard can do. A language standard is not like most engineering standards. The state of the art in software technology (or the level of maturity of the data processing industry) is not yet at the point where such precision is possible. Furthermore, rightly or wrongly, most standards still allow quite a bit of latitude to the implementor with regard to the meaning of certain language constructs. Nevertheless, a language standard can and should provide a framework for a workable, well-disciplined approach to software development. This is in fact the fundamental contribution of a language standard. It is important to recognize this because one commonly hears dubious claims made on behalf of standards—the most frequent of which is that the presence of a language standard makes program conversion (within the language, e.g., COBOL-COBOL) costs disappear. This is simply not true. Programming practices, imprecise standards, and environmental factors (e.g., operating system differences) all contribute to keeping the cost of conversion fairly high. An analysis performed by the FCCTS of some 32 COBOL-COBOL conversions reveals, for example, that the conversion cost of one line of COBOL code can range from $5.50 to $6.00, and this suggests that conversion costs are still high, even for programs written in "standard" COBOL.

To repeat—the principal purpose of a language standard is to provide a disciplined, predictable, efficient framework for software development. To fulfill this goal, a compiler must perform according to the standard. Testing is required in order to determine conformity.

The FORTRAN and COBOL Compiler Validation Systems

The FCCTS has performed functional testing of FORTRAN and COBOL compilers since its inception in 1973. This testing has been done using the COBOL Compiler Validation System (CCVS) and the FORTRAN Compiler Validation System (FCVS). A detailed description of these systems can be found in the references; only a brief summary is presented here. The CCVS74 (COBOL 74) consists of nearly 219,000 lines of COBOL code which collectively exhaust the meaningful constructs in the COBOL language. The FCVS 78 (FORTRAN 77) consists of over 62,000 lines of FORTRAN code, and represents a subset of the full standard. The FCCTS has also produced a HYPO-COBOL Validation System and an FCVS for FORTRAN 66, but the size of these projects was so small (12,000 and 38,000 lines of code, respectively) that data derived from them would be misleading. The interested reader is referred to References 9 and 10.

The Validation Systems consist of syntactically correct programs and an executive routine. The executive routine is used to perform certain text editing (e.g., specification of implementor names), for program and test selection (hardware dependent language elements may not be testable), and for the generation of job control language statements appropriate to the host operating system. Furthermore, an audit log of these actions is produced. These functions are necessitated by the environment in which the FCCTS operates—portability of the validation systems and audibility of the test procedures may not be required for in-house testing (although they are not bad features to have).

The Validation Process

The Compiler Validation Systems used by the FCCTS are necessarily generalized products, almost in the sense that an operating system is generalized when it is first delivered to a customer. A system generation process must take place to produce a product which is tailored to the installation in which it is implemented. In the case of the Compiler Validation Systems, this generation process consists of inserting, through the executive routines, implementor names in the source code, generating operating system control statements, and deleting (actually, not generating) tests which the compiler is unable to process at all, e.g., language fea-
tures which the compiler does not implement. This generation process is performed by the organization whose compiler is being validated, with some assistance from the FCCTS.

Validation is performed by the FCCTS staff, and consists of executing the audit routines on site, reviewing the resulting raw data, and producing a Validation Summary Report.5

Results

Experiences with COBOL compiler validations were reported by Baird and Cook.10 It is encouraging to note that many of the anomalies reported in 1974 have disappeared from compilers today. Errors reflecting sloppiness or poor judgement, such as performing syntax correctness checks on comments, no longer occur. Errors in simple constructs such as ADD, SUBTRACT, and COPY statements in COBOL are no longer prevalent. There is little if any justification for such errors, and their disappearance suggests that the availability of an independently-administered validation service has had an impact on compiler developers.

Table I illustrates the downward trend in both the average and the maximum number of errors discovered during a validation. It should be noted that the failure to implement a particular language feature is regarded as an error by the FCCTS. While this type of error is irritating to a programmer who wishes to use the feature it is not as serious as the presence of a feature which is incorrectly implemented, since this latter type of error is deceiving. We are finding no strong patterns in the errors found—i.e., our data does not reveal any meaningful distribution of errors according to language elements. The distribution according to modules is what one might expect—65 percent of all errors were found in the Nucleus, while 29 percent were found in the I/O modules. It is perhaps surprising that only 1 percent of all errors occurred in the remaining modules. We do note that a large number of “errors” are in fact caused by purposeful implementation decisions on the part of compiler developers, particularly in input-output functions, data representation, and other language areas which are closely related to operating efficiency of the total data processing system. This seems to suggest that implementors, when faced with the decision of implementing a feature correctly or efficiently, will chose efficiency. Given a community of users which traditionally has favored “efficiency” over correctness this is not surprising; but it is not desirable. The misconception that a rigorous standard inhibits efficient implementation is still a prevalent one.

Finally, it should be noted that the reduction in errors found in COBOL compilers took place during a period of transition from COBOL 68 to COBOL 74. This makes the error trend even more significant since it took place in a development rather than a maintenance context, and occurred in the face of an expanded, more complex version of COBOL.

A less encouraging trend has been the number of language features which the FCCTS has been unable to include in its Validation Systems. The FCCTS COBOL Validation Summary Reports list 24 language elements which are fully or partially implementor-defined. Some, such as “computer-name” in the COBOL SOURCE-COMPUTER paragraph are innocuous and acceptable. Others, such as the representation of a valid sign in certain forms of numeric tests remain a prevalent one.

TABLE I.—Validation Errors Summary

<table>
<thead>
<tr>
<th>Result</th>
<th>Description</th>
<th>1973</th>
<th>1977</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Average number of errors found for COBOL 68 compilers</td>
<td>50</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Maximum number of errors found for COBOL 68 compilers</td>
<td>50</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Minimum number of errors found for COBOL 68 compilers</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Average number of errors found for COBOL 74 compilers</td>
<td>9</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>Maximum number of errors found for COBOL 74 compilers</td>
<td>30</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Minimum number of errors found for COBOL 74 compilers</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Average number of errors found for FORTRAN 66 compilers</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Maximum number of errors found for FORTRAN 66 compilers</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Minimum number of errors found for FORTRAN 66 compilers</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

A typical result is that one recent COBOL Validation Summary Report listed 15 results “for information only,” i.e., applying to tests whose results are not well-defined by the standard. Six of these referred to input-output functions, and four to computation or comparisons. Thus, a COBOL programmer cannot use such common and useful statements as COMPUTE with any degree of certainty as to what the results are to be from one compiler to the other. This is perhaps to be expected in a language which is defined by a committee, whose national standard is set by a second committee, and whose Federal standard is set by a third committee.

The results of FORTRAN compiler testing are not so dramatic. This is due partly to the fact that testing to date has been with respect to the 1966 FORTRAN standard. The features contained within this standard are largely a subset of most FORTRAN dialects, and have remained stable over the past 12 years. Furthermore, FORTRAN is a simpler language than COBOL. It is therefore not surprising that the error rate found in FORTRAN compilers has been much lower than that found in COBOL compilers.

Two generalizations come to mind. One is that methodically and independently applied functional testing of compilers has produced meaningful and beneficial results. The other is that functional testing is limited by the quality of specifications.

Resources Required

Table II summarizes workload and resources expenditures data compiled by the FCCTS during the 1974-1978 period.


The CCVS74 elapsed time data is somewhat misleading. The 49 months given for the development of CCVS74 were really devoted to development of three distinct versions of the product, each of which was really a distinct, usable product in itself. Thus, periods of relative inactivity are included in this timespan. The reason for this approach is that development of the CCVS had to be coordinated with development of COBOL compilers, and these were developed and released incrementally. The first releases of COBOL 74 compiler implemented subsets of the language, and therefore the audit routines tested only these subsets. It is a peculiarity of compiler testing that development of the test tool must rely on the compiler; it fact many vendors used portions of the CCVS74 in testing their compilers during development. The 49 months cited were therefore caused by a continuing mutual "piggy-backing" of compiler-audit routines development. 

Productivity was quite high on all the validation systems development projects. This is due to a variety of reasons—the product specifications were to some extent already in existence (the standard represents specifications for audit routines as much as for compilers), the FCCTS staff was recruited for the specific purpose of software testing from the inception of the FCCTS in 1973, and the experience factor of the staff is quite high. Also, although a validation system may be quite large, as with the CCVS74, it really consists of a large number of relatively small, relatively independent modules. Productivity in developing this type of system will naturally tend to be well above average due to the lowered incidence of interpersonal communication required. 

The project time distributions show a greater percentage of the development effort is devoted to coding than one would generally expect. Again, this is attributable to the nature of the product being developed. The major phases of design, coding, and testing are not as distinct in this type of product as they are generally. The modularity and independence of the audit routines make it convenient to combine much of the documentation and unit testing activities with the code production phase in a way which makes it impossible to separate the time associated with each. 

Table III summarizes the FCCTS resources expended for validations. We believe that a good portion of the professional time presently spent in the report production could be shifted toward clerical time, and are taking steps to do this. 

A few observations regarding the makeup of the staff might be of interest. The FCCTS was formed in 1973 and performs some of its functions in support of Federal procurement regulations. Previous experience with compiler validation existed, but much remained to be learned. Recruitment and staffing therefore leaned toward experienced, highly competent personnel, and the positions established were fairly senior. This was wise at the time, but it is not the optimal staffing pattern once some experience has been gained, nor is it optimal outside the Federal procurement support environment. Development of validation systems requires a high level of skill, ingenuity and maturity during the design stages of the product, and for the development of certain support functions such as the executive routines and code generators; but the coding and testing processes bear such a resemblance to a production line that high-caliber personnel is simply neither required, nor desirable. Rather, such a project should be staffed along the lines of a Chief Programmer Team, with some obvious modification (it perhaps should be called a "chief designer team"). The project manager should also act as the chief designer, with the primary responsibility for developing the test specifications. A software specialist should be responsible for developing support software such as the executive system. The coding task can be relegated to junior programmers. Ideally, a single individual should be responsible for computer-based testing. It has been found very useful to have programmers visually review each other's code—the simple structure of audit routines makes this practice very cost effective. Documentation can be voluminous; the validation system itself should be self-documenting, but such documents as user guides should be produced by a literate technical writer. Finally, a "librarian" is both feasible and useful. 

Technical Problems

It was indicated earlier that lack of completeness can be a problem in functional testing. It has not been a problem in compiler validation. The reason is that the potentially combinatorial number of logic paths which should be tested in application software (whose inputs are unpredictable) is not a factor when testing a compiler. Ensuring that all meaningful language constructs are tested may tax the patience of the developer, but not his intellect. The problem is further alleviated by limiting the tests to correct language constructs, i.e., we do not insert erroneous code in the audit
routines. This decision is motivated by the environment in which the FCCTS functions. If we were developing tests for an in-house product we would very likely want to introduce erroneous code in the tests. We would then be faced with the traditional question of what errors to introduce, at what frequency distribution, and by whom should they be introduced.

A problem we have encountered is that of properly identifying the building blocks of the audit routines. This is a problem unique to validation of compilers. Since the audit routines must be compiled by the compiler which is being tested it is necessary to identify certain language constructs for which correct compilation is taken as a set of axioms, in the sense that if these features are not correctly implemented there is no point in continuing the validation. An example of this building block approach can be seen from the FORTRAN Compiler Validation System (FORTRAN 77). Some of the assumptions we have made regarding the Subset FORTRAN are:

1. Six (6) character symbolic names and five (5) digit statement labels are permitted.
2. Comment lines do not affect a program in any way.
3. Execution of the unconditional GO TO statement
   \[ \text{GO TO s} \]
   causes the statement identified by the statement label \( s \) to be the next statement executed.
4. Branching to a CONTINUE statement causes the statement following the CONTINUE statement to be the next statement executed.
5. The arithmetic assignment statements
   \[ \text{variable} = \text{constant} \]
   \[ \text{variable} = \text{variable} \]
   function correctly.
6. The arithmetic IF statement functions correctly:
   \[ \text{IF (e) s1, s2, s3} \]
   where \( e \) is any arithmetic expression of the form
   \[ \text{variable + constant} \]
   \[ \text{variable - constant} \]
   and \( s1, s2 \) and \( s3 \) are statement labels.
7. The character assignment statements
   \[ \text{character variable} = \text{character constant} \]
   \[ \text{character variable} = \text{character variable} \]
   function correctly.
8. The logical expression
   \[ \text{IF (character relational expression) executable statement} \]
   where the form of the character relational expression is
   \[ \text{character variable .EQ. character constant} \]
   functions correctly.
9. The following form of the WRITE statement functions correctly:
   \[ \text{WRITE (u, f) iolist} \]
   where \( u \) is a unit specifier, \( f \) is a FORMAT identifier, and \( iolist \) is an input/output list containing arithmetic or character variables. The format statement contains \( nH \) edit descriptors, \( X \) edit descriptors, numeric editing descriptors and/or \( A \) edit descriptors.
10. In order for the output report to have the correct format, the use of the first character of a formatted record for vertical spacing on a printing device must function correctly.
   The two characters which are used in printing the report are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Vertical Spacing Before Printing</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td>One line</td>
</tr>
<tr>
<td></td>
<td>To first line of next page</td>
</tr>
</tbody>
</table>

11. The system output device has at least 56 characters per line.
12. An integer datum consists of at least 16 bits of which one bit is a sign bit.
13. A real datum contains at least 16 bits in the mantissa and eight bits in the exponent.
14. A character datum of length 14 is permitted.

These assumptions are necessarily subjective—a different designer might have made different choices. The point is that choices must be made, and that the validity of the choices must be determined prior to full testing.

Because the type of validation systems we are discussing consist of a large number of small independent modules, and because the high-level specifications are a "given" (i.e., the language standard) there is a tendency to bypass the design phase (after having determined the "axioms") and to plunge immediately into the coding phase. The danger with this approach is not that the resulting product will be a faulty one, but rather that it will be an unnecessarily voluminous one. We have found it useful to develop general specifications which define in a broad way a set of tests for a given language module or construct (e.g., arithmetic expressions), and to follow these with a set of detailed specifications which define each test. The general specs allow us to refine the estimates of time and resources required for the project, and to ensure completeness "in-the-large," while the detailed specs provide thorough documentation and enable the actual coding to be done by relatively junior personnel.

Audit routines should be self-checking. The volume of output is far too large to be eyeballed. We have also found it useful to be able to suppress printed outputs during the testing phase of a validation system development.

We have encountered two problems which are unique to an outside testing group which is testing a product on different systems. One is portability of the validation system; the other is auditability of the test process itself. We must be able to execute the audit routines on any compiler-operating system-hardware combination, and we must be able to assure the consistency of our testing procedures. The executive systems allow us to fulfill both the goals. In-house testing would require no more that a good text-editor and a library management function. Additional details of the validation process using the executive routines are to follow.

Although our productivity has been high we believe that it is necessary to improve it. We would like to be able to develop a validation system for a new language, e.g., PL/I, in six months or less, while limiting the project staff to, say,
five people. This is not yet possible. We have taken some steps to increase our productivity. One has been to adopt very stringent and precise development procedures, so that tasks are completely interchangeable among project staff members. These are described in greater detail below. We have also found it useful to use the COBOL Compiler Validation System executive routine in developing the audit routines. This not only ensures consistency but also provides us with a useful and complete log of all significant development tasks. Our major effort at improving productivity has been the development of a pre-processor, or generator, as part of our FORTRAN validation system project.

The generator is used to speed up production of "boiler plate" material and repetitive code which changes only with regard to statement labels. Specifically, the generator is used to:

1. Produce standard comments which appear, unchanged, before each grouping of tests or before each individual test.
2. Produce code to be executed when a test is passed, failed, or deleted.
3. Produce "standard" variables declarations.
4. Generate statement labels which render the common code segments unique.

While the generator has been useful, it should be noted that it is not used to generate the test code itself. We have given this possibility serious thought for over three years, and have concluded that while test code generation is feasible, it is not productive. Such an attempt would be useful if either of these conditions were true:

1. We needed to produce multiple but differing versions of a validation system for a given product.
2. The generator could be used to produce validation system for different languages or products.

The former condition does not exist, while the latter is not feasible except for the most simple-minded languages. We are therefore faced with the uneasy feeling that we are reaching a practical limit on productivity with regard to validation systems development.

We do expect to achieve some improvements through a refinement of our programming practices. Our standard development rules presently specify the following:

- Symbolic name conventions
- Assumption or axioms
- Statement label conventions, particularly in distinguishing among test, pass, fail, delete, verify, and I/O code
- Convention for external unit identifiers
- File naming and contents rules
- Composition and organization of routines and tests
- Phases in the development cycle
- Use of the generator

We hope to find additional ways of expanding and using the generator, but are skeptical as to the chances of making quantum improvements in our productivity.

The Executive Routines

The executive routine is used to control the generation of a specific Validation System, to control the generation of job control language statements, and to perform updates. System generation consists of program selection, identification of options to be used and the validation environment, and the control of report outputs.

Job control statements are generated for a given operating system through a "higher-level language" which is used to indicate the need for accounting statements (e.g., JOB or RUN), for the invocation of processors such as compilers and collectors, and for the initiation of execution.

Finally, a simple set of text-editing statements is available for replacing, adding, or deleting source statements. As was indicated earlier, the need for an executive system is predicated on the degree to which the validation system itself must be portable.

FUTURE WORK AND UNRESOLVED PROBLEMS

Much of our thinking during the past few years has been directed at determining the feasibility of automating the test generation process. Our motivation has been to increase the rate at which we can produce compiler validation systems. Others have been motivated by the perception of additional problems: 13

1. Lack of a formal construction method
2. No reference to measures of test effectiveness
4. High costs

We believe, based on our experiences, that rigorous procedures, augmented by some support tools, adequately address the first three problems. We furthermore do not believe that cost is an inhibiting factor. The cost of CCVS74 has been approximately $400,000 (including maintenance), or somewhat under $2/line of code. This is an almost insignificant cost compared to development costs in general. Naturally, the fact that we act as a central service facility whose testing product is applied to many compilers reduces the relative cost—$400,000 spent to test 40 compilers is a less disturbing figure than $400,000 spent to test one compiler.

Our experience to date suggests to us that attempts at using a formal notation for the specification of programming languages as the means by which a language's syntax and semantics may be described, and by which tests may be generated, are ill-conceived. This approach is certainly feasible. But, as anyone familiar with syntax-directed compilation knows, this approach will not significantly reduce the time required to produce compiler validation systems. The
reason is simple enough—it is no easier to specify a test using a formal notation than it is to specify it in the programming language directly. Furthermore, the differences among languages are great enough to preclude using a single test specification to generate multiple tests, each in a different language. We have not closed the door on this approach, but we have become very skeptical of its merits.

A more serious problem, in our opinion, is that of the poor quality of language specifications. It is in fact somewhat fruitless to concern oneself with the completeness of a test vehicle when so many significant language features are ill-defined. This is a solvable problem, but its solution will require a "consumer movement," whereby users become more critical of what is handed to them by standards committees. As of now, inadequate language specification represent the most significant limitation on function validation. The PL/I Standard is a major step forward in addressing this problem, despite the criticism it has received.

A related problem has to do with the stability of the product being tested. The FCCTS receives numerous complaints that we often insist on revalidating a compiler because of new releases of the compiler or of its operating system. Our rejoinder is that we will drop our requirement for revalidation when offerors of compilers stop putting out a new release every six months. The industry as a whole has standards of software quality control that would embarrass most other professions.

Finally, more attention needs to be given to the development of specifications languages, so that the process of testing the specifications, producing a product from these specifications, and producing a testing system with which to test the product could be a largely automated, coordinated effort. We are aware of the work that is under way in this regard, but results have been very meager.

REFERENCES