Logicon's experience in performing V&V focuses on critical software developed for the Department of Defense. Our earliest projects involved certification of space flight and ground programs used by satellite launch vehicles. Our role has since expanded to include the software that performs targeting and launch control of strategic missiles, as well as software used in allocation, assignment, and routing of strategic weapons. In this work, we have further refined our V&V techniques to address the strict surety and safety requirements of software-controlled nuclear weapons. In related efforts, Logicon is performing V&V of avionics software for both developmental and upgraded strategic aircraft systems, which spans applications ranging from electronic countermeasures to terrain-matched navigation. Our current work also includes V&V of several tracking systems used for missile flight safety and national airspace surveillance.

Logicon has developed an extensive library of software tools that aid in our software analysis and testing during V&V. In analyzing requirements, we use CARA, an automated requirements traceability aid developed from the government-owned URL/URA developed by the University of Michigan. In analyzing design, Logicon uses engineering simulations to confirm the accuracy of the specified equations and algorithms. In analyzing the program code, we employ comparators, flowcharters, syntax analyzers, structure analyzers, cross-referencers, and our symbolic executor named AMPIC. In testing, we use drivers, co-resident monitors, branch analyzers, and interpretive computer simulations.

Logicon's philosophy is to balance the use of tools with careful analysis. We have found that an analyst can effectively anticipate the types of errors made by the developer and direct his attention to the most error-prone program constructs. We have also found that the success of the V&V project depends not on the number of tools used, but upon the correct selection of tools and proper use of them. Having a large number of tools, or even the best tools available, will not make V&V effective unless the proper analytical procedures complement their use.

In internal studies of our V&V projects, we have found that analytical methods can detect the widest range of error types. Software often contains elusive logic errors that would be impractical to find by any method other than manual analysis of the logic and program structure. On the other hand, it is often more cost-effective to use special purpose tools to discover routine coding errors and we have found that tools like comparators, flowcharters, and syntax analyzers are invaluable aids for this reason. Our experience has led us to the conclusion that proper planning is the most vital element in ensuring V&V's effectiveness. The crucial decisions made at the onset of the project select the methods that are used, while the decisions made during the V&V effort determine how thoroughly and effectively these methods are applied to the software.

Position Statement
Validation and Verification vs. The Software Life Cycle

Leon G. Stucki, Boeing Computer Services Company

In attempting to forecast the future trends in software engineering, a very dynamic future appears to be in store. Validation and verification are rapidly becoming recognized as an integral part of the total software life cycle. Validation and verification are no longer being viewed as new names for testing. A modern view of validation and verification encompasses the set of tasks and functions needed for tracing requirements through the evolving stages of the system life cycle down to the final implemented code. At each stage of program development, verification of each successively developed representation back to the next higher level representation needs to be performed. Validation is a separate functional activity which also needs to be performed. The main difference between validation and verification is, instead of examining the mapping between representations, an attempt is made to ensure the system will correctly implement the users preceived requirements; the principle question being: Is the system usable?

Thus, validation and verification are not activities which occur merely at the end of the coding process. Validation and verification involve such issues as verifying the preliminary design document back against its requirements, verifying a detailed design document back against a preliminary design document, verifying the code back to the detailed design. Each of these verification exercises is an attempt to provide a trace between the evolving stages of the system. Validation of the developing design occurs in parallel with system evolution. The preliminary designs is validated by the user, the detailed design ramifications are also presented to the user for validation as is the actual operation of the code. This expanded view of validation and verification is not without its problems. Several examples will be given.

One of the trends that is becoming quite evident is the attempt to add more rigor to these early life cycle processes. Such attempts will facilitate not only validation and verification but will also bring needed order to the current state of chaos that now prevails in many of our large projects.

The software life cycle presents yet a further problem area worth pointing out. This deals with the transition between detailed design and coding. Current state-of-the-art trends advocate numerous representations for this interim period. Examples of the representations involved include such things as hipo charts, state transition diagrams or flow charts, program design languages, structured pre-processors, and finally the high-level language processors themselves. While not advocating that these techniques are bad, it is worth observing that there is still much manual translation between the various representations.

Furthermore, errors that are discovered in one early representation are not necessarily eliminated from subsequent representations. The possible reoccurance of previously discovered errors could be eliminated if more rigorous transitions (automated where feasible) were provided between the various representations. Conceptually, there is really no reason why automated tools could not communicate with each other and build upon past knowledge. Paradoxically, as software people, we have done an excellent job of automating everyone's job except our own.

We will, in the future, see several modifications to the type of tools used today. The first, already eluded to by previous comments, will be the trend toward the creation of computer assisted programming environments integrating various tools around a central data base. This data base will provide a new level of configuration management and change control useful for the maintenance of large systems. The other clearly apparent trend is that the tools themselves will become more widely used. The proven functions performed by these programming environments will be available to wider audiences.

Position Statement
Software Verification and Validation in Practice and Theory

Sabina H. Saib, General Research Corporation

The State Today

The techniques and tools that are available to the developer of practical software systems today are oriented towards the detection of errors in a test environment. That is, the software has been designed, coded, walked through, and tested by the designers, and is now to be subjected to outside verification and validation. Among the test tools that can be used to aid in the testing process are JAVS[1], FAVS[2], RXVP[3], PACE[4], and PET[5].

In such an environment, errors are found by checking test results and inspecting source listings. Unfortunately, it is much more costly to correct errors at this late stage. Every change must be formally documented; the original designers (who have gone on to other projects) must be consulted; and the tests must be rerun from the beginning. In some projects a single change can result in months of effort.

The State Shortly

It would appear that prevention of errors is much more valuable than detection and correction of errors. Where test tools have been applied at early stages, the number of errors persisting after acceptance test have been significantly reduced.[6] An analogy is that vaccination against disease is much less costly than the detection and treatment of the disease.

The only errors that are easily and painlessly removed today are syntax errors. However, we have the knowledge acquired through several studies[7,8,9] of the kinds of errors that are present in software, and efforts are presently underway to make the early removal of these errors as painless as possible. For example, DAVE [10] and SQLAB[11] attempt to detect certain types of errors in the source code before testing.

A few systems are even being developed to address problems that can arise at the requirements and design stage, notably PSL/PSA[12] and SREM.[13]

Language Assistance

At the same time that verification and validation tools are being improved, programming language designers are also concerned with the early elimination of errors. This appears to be done best with the addition of redundancy checks and the imposition of restrictions. The appearance of a language such as PASCAL[14] which contains some redundancy has demonstrated that some semantic errors can be detected and removed at compile time.

Redundancies such as the definition of types in the procedure invocation and in the procedure heading, and the use of logical assertions and statements concerning variable usage, have been shown to prevent many types of errors.[15] Restrictions on program size, control structures, and access to global variables are also valuable precautions.

The Future

Analysis that is currently performed in separate tools will be incorporated in future compilers. This is necessary because there is a tendency today to bypass the use of analysis tools. It is impossible to bypass the compiler, and much of the expense in present-day analysis is in the syntax analysis and symbol table building that is done in compilers anyway. This will overcome the present need for programmers to be trained in the use of a verification tool.

The future will see the incorporation of small proofs of program characteristics. Unlike today's consistency proofs, these proofs will not require the programmer to supply lemmas, axioms and loop