AN APPROACH TO SOFTWARE TESTING: METHODOLOGY AND TOOLS

Roger R. Bate and George T. Ligler

Advanced Software Technology
Texas Instruments, Inc. Dallas, Texas

Abstract

In recent years attitudes to software testing have increased in formality, for well understood reasons, from informal test case selection and execution to consideration of mathematically rigorous proofs that a program satisfies its requirements. The parallel development of software development methodologies has heightened awareness that testing approaches must be fully coordinated with software construction practices. This paper presents an approach to utilization of these trends and a prognosis for future testing technique evolution.

Introduction

Over the past few years there has been a growing realization that software testing can no longer be treated as an informal art if software is to meet its requirements with a high degree of reliability. There has been a continual increase in the formality of test planning and of preparation of test procedures to meet more stringent standards for acceptance imposed by customers. Software engineering research has led to a raft of systematic testing techniques applicable during several stages of software development. Effective symbiosis between applicable testing techniques and a total software development methodology is the goal of several long-term programs at Texas Instruments.

We present our methodological model and the approach which we take toward tool construction for what we call an integrated software support system. Within this methodological framework, we discuss both our current testing techniques and our plans for the evolution of software testing tools and procedures.

A Methodological Framework

Figure 1 depicts our current methodological approach to software development and testing, developed for both internal and customer requirements (10). An independent test team coordinates test planning and execution in a requirements-directed manner. Depending upon the types of testing techniques involved, the test team either develops or monitors the development of the "tests" themselves.

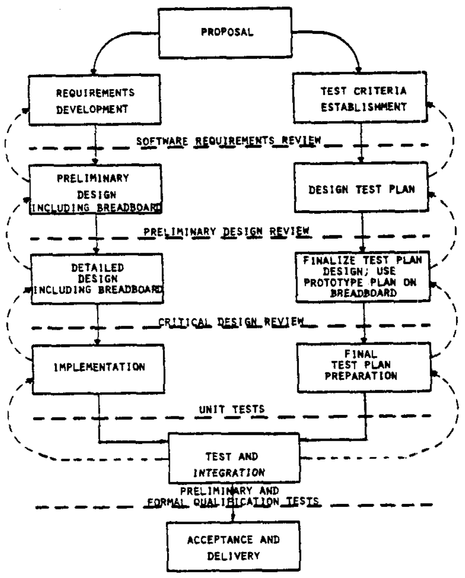
PROPOSAL
REQUIREMENTS DEVELOPMENT
TEST CRITERIA ESTABLISHMENT
SOFTWARE REQUIREMENTS REVIEW
PRELIMINARY DESIGN INCLUDING BREADBOARD
DESIGN TEST PLAN
PRELIMINARY DESIGN REVIEW
DETAILED DESIGN INCLUDING BREADBOARD
FINALIZE TEST PLAN DESIGN; USE PROTOTYPE PLAN ON BREADBOARD
CRITICAL DESIGN REVIEW
IMPLEMENTATION
FINAL TEST PLAN PREPARATION
UNIT TESTS
TEST AND INTEGRATION
PRELIMINARY AND FORMAL QUALIFICATION TESTS
ACCEPTANCE AND DELIVERY

FIGURE 1. THE SOFTWARE DEVELOPMENT CYCLE.

We advocate for software development an integrated software support system, which consists of a set of tools and procedures which can be used during the development of software, the maintenance of software, and its modification. The tools are in five parts:

a) input, which consists of various hardware input devices such as CRTs and card readers, as well as software input controllers to provide capabilities for editing in an interactive mode. The input controllers should be tailored to the syntax of the language being input in order to be of maximum assistance to the user.

b) a software database management system which is capable of holding all of the information pertaining to a software development project, such as source code, object code, test data, test results, management information, and documentation.

c) a set of transformers (or analyzers) which have access to the data held in the database and which deposit their results in the database. Examples are: compilers, assemblers, link editors, test generators, data flow analysis programs, test grading programs, and formal verification programs.

d) output, consisting of hardware such as CRT displays and printers and software for selecting and outputting information useful for the programming process such as pretty printers, management report generators and documentation formatters.

e) A control which determines the rights of certain users to information in the database, schedules the activities of the system, and records the costs and other management information concerning the software development process.

There must also be a set of procedures for the use of the tools described above and a set of standards and management controls to assure that software development is carried out correctly. In Figure 2 we show the flow of information that is currently supported by the tools and procedures of our evolving software support system (4).

Our approach to the construction of an integrated software support system is threefold. Firstly, "horizontal" integration of tools and procedures with stages of the software development cycle presented in Figure 1 is required. Such integration has been discussed and at least partially implemented in numerous quarters.

The second thrust of our procedures and tools development is that there must be support for "vertical" integration of all phases of the software development cycle in the sense that the various tools must have compatible inputs and outputs. The aforementioned inputs and outputs must be easily controlled by the user. In this way, he will be encouraged to use all of the available tools rather than omitting by some because they are too hard to learn to use or their data is too difficult to prepare.

A third aspect of software support system construction is that several levels of formality need to be applied within the tools set. An integrated software support system must provide for management review and the application of current, if imperfect, techniques in a manner as directly extensible as possible to the assessment of properties of software development products with mathematical rigor - we are finding integration of this nature in large measure attainable.

SOFTWARE DEVELOPMENT INFORMATION FLOW

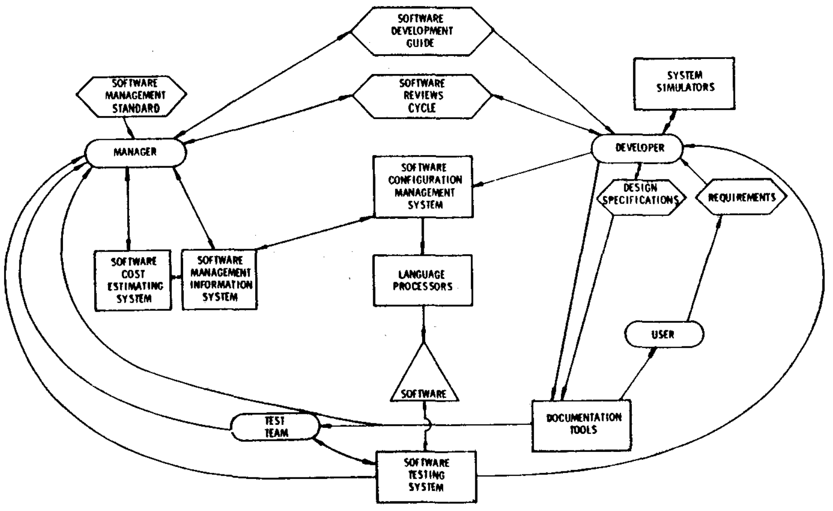

Figure 2

Testing Tools

As the entire methodological approach discussed above is oriented toward building reliability into software, all tools which support the various stages of the software development cycle may rightly be called software testing aids. We emphasize here elements of Figure 2 (and their successors) which are most directly associated with testing at appropriate stages of software development. We define software testing to encompass all of those things which are done to software to make sure that it meets the requirements. Testing can usefully be partitioned into two parts: static testing, which does not involve running the program but performs analysis of the code itself as well as specifications and requirements, and dynamic testing, which involves running the program or a simulation of the program and measuring its behavior.

Static Tools

We have been evolving toward the production and use of static testing tools that are both horizontally and vertically integrated with our software development methodology, during involvement in the Ballistic Missile Defense software engineering program (12) and subsequent internal efforts which are currently scoped to continue for the next five years. Static testing techniques are principally applicable to software requirements, specifications, and programs.

State-of-the-art problems with testing tool support for vertical integration of software methodologies are exhibited in Figure 3. With several noteworthy exceptions, principally in the research community (e.g., 1, 17), requirement language, specification language, and programming language design have proceeded rather independently of each other so that the languages produced differ in style, content, and rationale. This is most unfortunate, because the interfaces between such languages are the points of greatest confusion where the meaning of requirements is to be conveyed to those who will write the specifications and where specifications are conveyed to the designers and programmers. The confusion of language change occurs at the point of the most critical information flow. Conversely, testing of software to requirements involves the relation of software to specifications and specifications to requirements, and again lack of integration of these languages can have devastating consequences.

We have oriented much of our static testing tools thrust toward the provision of related multi-level requirements and specifications languages, features of which may be incorporated into programs in existing languages through embedded assertion techniques. The multi-level approach is necessitated by the desirability of progressively developing requirements and specifications with increasing formality. Within the above framework, a "subset" of both requirements and specification languages could be suitable for potential verification of specifications to requirements and

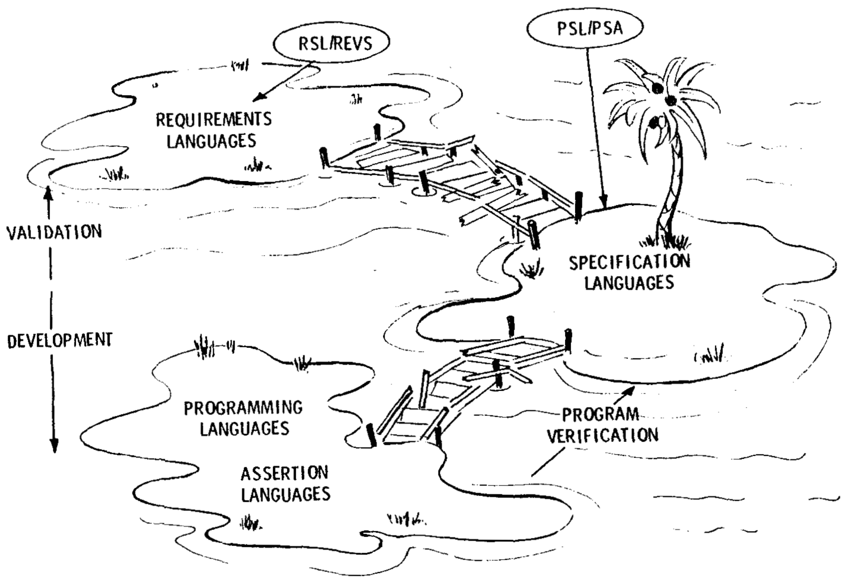

Figure 3

programs to specifications. Techniques necessary for the former verification process, particularly in the area of performance requirements, have been under insufficient investigation. In contrast, there has been highly active research in verification of programs to specifications, with results of debated applicability (9, 16). Apart from "expected" interest in high level-of-confidence testing techniques, we have become actively involved in verification research because the difficulties with reproducing errors in real-time multiprocessor systems seems to imply that program verification may be the only acceptable way of assessing the reliability of concurrent systems. Additionally, the process of verification lends insight into the integration of our near-term tools construction involving fruitful, though less formal, techniques with testing methods of the future.

Given that we must use existing programming languages for some time, a multi-level approach to software implementation may be fruitful for static analysis. The utilization of a program design language which employs an "abstract machine" approach to algorithm specification is under analysis as an intermediate step of implementation more amenable to reliability assessment techniques than typical concrete realization of program specifications. This approach is due to Dijkstra (8) and others and is also employed in other industrial methodologies (6).

Our state-of-the-art in static testing tools is far from the above scenarios, especially in requirements formulation. We currently use informal requirements analysis techniques, have standardized our requirements document format, and are prototyping utilization of specific analysis algorithms and tools (e.g. (2), (5)) as groundwork for our requirements language definition effort. We are beginning the design of the specification language and are developing traceability-level correlation tools between requirements and specifications. We are utilizing one prototype program design language on a real-time navigation system project. The programming languages PDL2 (14) and TI Pascal (15) have been designed with source-level assertions and semantic rules to aid verifiability, and their compilers provide useful static analysis tools, including cross reference, global data and control flow analyzers. Elements of our Software Testing System, which is primarily oriented toward empirical testing, carry out software complexity and "dimensionality" assessments (7). Finally, we have found the somewhat underrated technique of disciplined code audits to be highly useful. Recent projects have indicated that they have spent between 6 and 10 percent of total development time in code audits with excellent effect; integration of future static tools with code audit techniques will ease this people-intensive task.

Dynamic Testing Tools

Empirical testing techniques have received considerable emphasis and automation support. Although exhaustive dynamic assessment of a program is of prohibitive cost, the absence of mature static analysis procedures leaves advanced empirical techniques among the most effective available. Our prognosis for dynamic aids is that they will continue to be of value, especially in validation of performance requirements, and we have defined points of integration of such tools with our static testing tools set. We are undertaking research in "software science" in order to assess the effectiveness of both dynamic and static techniques.

We stress "breadboarding" analysis of software systems during the design phases of the software development cycle (3). An example of tools support for breadboarding is our MULTAS system, an event-driven multi-facility time analysis simulator for performing simulations of distributed multiprocessor systems. Integration of dynamic testing during system design and the correlation of requirements and specification language system descriptions is an area of current research.

We have developed improved techniques for the use of selected case testing during software implementation and test and integration. Our Software Testing System (STS) provides facilities for the measurement of test plan coverage of control paths, source traces in the event of errors, and Histogram-oriented program performance analysis. STS runs on four machines and interfaces with several programming languages, including Fortran dialects and TI Pascal, and is currently similar in scope to several other available testing tools (e.g. (11)). We are working on improved techniques for automatic test data generation. Additionally, we are extending STS to support an assertion language capability both to augment our current use of "dynamic" assertions and to provide another interface point to our static, verification-oriented tools.

Concluding Remarks

The evolution of software testing tools will depend upon at least three factors. First, the feasibility of formal program assessment techniques must be understood; much investigation of relating specifications to requirements must be done here. Second, investigations of the effectiveness of testing techniques is mandatory. To date only isolated studies have been reported (e.g., (13)), and testing technique cost effectiveness presents a fertile area for research, perhaps under the rubric of "software science." An essential part of this research effort needs to be oriented toward the human engineering of testing techniques. Finally, testing tools will evolve with the methodology with which they are integrated.

References

(1) A. L. Ambler, "GYPSY: A Language for Specification and Implementation of Verifiable Programs," SIGPLAN Notices, 12:3, 1-10 (1977).

(2) R. C. Andreu and S. E. Madnick, "An Exercise in Software Architectural Designs: From Requirements to Design Problem Structure," Technical Report, MIT center for Information Systems Research (1977).

(3) R. R. Bate, "Software Design Procedures," in Proceedings of the AIAA/NASA/ACM Computers in Aerospace Conference, 102-107 (1977).

(4) R. R. Bate and G. T. Ligler, "A Software Development Methodology: Issues, Techniques, and Tools," Proceedings of the 11th Hawaiian International Conference on Systems Sciences, Honolulu, 40-44 (1978).

(5) T. E. Bell, D. C. Bixler, and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. Software Engineering, 3:1 (1977).

(6) D. L. Boyd and A. Pizzarello, "Introduction to the WELLMADE Design Methodology," IEEE Trans. Software Engineering, 4:4 (1978).

(7) G. W. Cobb, "A Measurement of Structure for Unstructured Programming Languages," in Proceedings of the ACM Workshop on Software Quality Assurance (1978), to appear.

(8) E. W. Dijkstra, A Discipline Of Programming, Prentice-Hall (1976).

(9) E. W. Dijkstra, "On a Political Pamphlet from the Middle Ages," Software Engineering Notes, 3:2, 14-16 (1978).

(10) Department of Defense, "Military Specification, Software Quality Assurance Program Requirements," MIL-S-52779(AD) (1974).

(11) C. Gannon, "JAVS: The Jovial Automated Verification System," in Proceedings of the 2nd IEEE International Computer Software and Application Conference.

(12) S. N. Gaulding and J. D. Lawson, "Process Design Engineering: A Methodology for Real-Time Software Development," in Proceedings of the 2nd International Conference on Software Engineering, 80-85 (1976).

(13) W. E. Howden, "Theoretical and Empirical Studies of Program Testing," IEEE Trans. Software Engineering, 4:4, 293-297 (1978).

(14) Texas Instruments Incorporated, Process Design Methodology Design Specification, Vol. 1: Process Design Language (1976).

(15) Texas Instruments Incorporated, Report on the Programming Language TI Pascal (1977).

(16) B. Wegbreit, "Complexity of Synthesizing Inductive Assertions," JACM, 24:3, 504-512 (1977).

(17) W. A. Wulf, R. L. London, and M. Shaw, "An Introduction to Construction and Verification of Alphard Programs," IEEE Trans. Software Engineering, 2:4, 253-264 (1976).