CASE STUDY: APPLYING EXPERT SYSTEMS TECHNOLOGY TO TESTING PHASE OF SOFTWARE LIFE CYCLE

Margaret J. Davis

The Boeing Company
Boeing Aerospace Division
P.O. Box 3999 m/s 9Y-37
Seattle, Washington 98124-2499
(206) 657-6797
uw-beaver!ssc-vax!mjdavis

ABSTRACT
This case study describes the use of an expert systems approach to automation of systems and integration testing for validation of complex, real-time communications software. The approach permits a "state"-based rather than path or branch-based testing style. States can be associated with high-level system requirements to give a measure of test coverage. The benefits and weaknesses realized from using the Boeing-built embeddable expert systems shell with a custom relational database interface to construct an automated software verification tool supporting this approach are discussed along with a brief summary of the utility of applying expert systems technology in this software engineering area. Measurements of the productivity increase from a prototype demonstration are also included.

The Boeing-built embeddable expert system shell is written in "C" with a custom interface to a relational database and support for user functions written in "C". Approximately 10 person-years were invested from late 1985 through 1987 to produce the prototype system of approximately 500 rules. Another 3 person-years were invested during 1988 to produce about 5500 rules covering the entire communications protocol.

1. PROBLEM DEFINITION
Software system validation and verification is the tedious and arduous process of demonstrating that a software program correctly implements its requirements specification. There are two major approaches to performing such validations: testing and formal proofs of correctness. Testing involves executing a program with a known set of inputs and analyzing the results; formal proofs of correctness involve proving that the expected results are as required for all valid inputs.

The Boeing Company uses the testing approach to validate that software it delivers for the Airborne Warning and Control System (AWACS) aircraft adheres to the complex, real-time communications protocol required. Over the past ten years, Boeing has expended millions of dollars validating the AWACS baseline and subsequent software releases. A test consists of (1) recording on a magnetic tape periodic snapshots of computer memory for a predeterminated scenario from a laboratory simulation of an AWACS mission; (2) printing a selected portion of the snapshots, which results in printout that is more than 6 inches thick, and (3) hand analyzing the printout for compliance. When challenged by a government contract to prototype a tool that would improve software engineering productivity in this validation area, we responded by developing an Automated Software Verification (ASV) tool that supports encoding the hand-analysis as the rule base of an expert system and abstracting the computer memory snapshots to supply basic knowledge of what occurred during the laboratory simulation. Encoding the hand-analysis is done by representing specific protocol requirements and the test engineer's temporal reasoning as situation/action rules. Abstracting the snapshots is done with a two step process that translates (1) from the bits and bytes on the recording tape to representations of JOVIAL program data structures in a relational database (2) to representations of time-ordered significant events as forward-flowing sequences of facts available to the knowledge base.

The salient characteristics of the problem of validating software for a communications protocol are: (1) the rules of behavior were contrived by humans as opposed to natural law; (2) the behavior of interest can be described unambiguously; (3) the data to be analyzed describe the states of a system and are time tagged, and (4) there is a high volume of data. Our successful demonstration of the prototype tool and the extension of the prototype to cover the entire communications protocol shows how well the ASV approach fits this problem.

2. PREVIOUS APPROACHES
A survey of automated software testing techniques that was conducted for the contract concluded that
dual programming is the essence of all forms of automated software system testing. In the usual form of dual programming, alternate implementations of a set of requirements are coded and then competed against each other. This alternate implementation is called a test oracle. The oracle implementation, especially in the case of unit testing, is often a human armed with pencil, scratch paper, and calculator. Disagreement indicates an error in one of the two implementations or an incomplete, ambiguous requirement. Agreement increases confidence that the requirements have been implemented correctly.

Developing dual, independent implementations is an expensive process in terms of time and money since two teams performing separate design and coding efforts are needed. It has also been reported that testing with dual programming has a subtle shortcoming. Alternate implementations often share misconceptions which arise from their common perspectives of designing software constrained by the same target hardware and programming language paradigm.

The automation [3,4,5] of validation against an oracle has been achieved by constructing testing systems with automatic test drivers that instrument the code, run the test cases, collect the results, and compare each test case result against what is expected. This automation works well at the unit testing level and for system and integration level testing of software where the expected result is independent of the current system "state". This automation approach is cumbersome for system and integration level testing where the expected result is dependent on current system "state", particularly for embedded real time software. Communications protocol software falls in the latter category.

3. APPROACH OF THIS CASE STUDY

Because the Boeing AWACS simulation and testing laboratory provides semi-automatic testing capabilities through canned scenarios and test recording, the best opportunity for productivity improvement in system validation was automation of the test analysis. This automation required replacement of the human as test oracle and comparator. Since the test engineer uses the protocol requirements as a basis for deciding what the software behavior should be and these requirements are natural language in the style of "When this message has been received and the incoming values disagree with internal values, do ...," it was natural to use situation/action rules to act as test engineer/analyst.

Although the data on the recording tape would provide the input test cases and output results, the volume of data available from one-half hour test session made it unreasonable to represent the entire contents of the tape in the knowledge base at one time. The low level of abstraction of the data was also deemed unsuitable. We elected to raise the level of abstraction of the data, store the abstractions in a relational database, and provide an interface between the expert system and the database that would supply facts to the knowledge base on demand. Raising the data's abstraction level represents an automation of the data preparation that the test engineer performs by marking up the test results printout with the help of colored pencils, clips, and a hexadecimal-octal-binary calculator. Although this abstraction process is a classical form of software automation, it is critical to successful application of expert system technology to the problem. It reduces the expertise that needs to be encoded in the expert system to a manageable size and complexity and separates knowledge about implementation details of the software under test from the knowledge about the behavior required by the protocol. The latter reason reduces the unconscious bias of test engineers to avoid exercising program functionality that is likely to be incomplete and faulty based on their knowledge of the software and its design.

We found no commercially available system that would meet our requirements for an embeddable shell that runs on Apollo workstations and provides a relational database interface, so in mid 1986 we began developing what we needed. We designed the shell to have the following characteristics: (1) run in the Apollo environment; (2) allow rule actions to invoke arbitrary procedural code; (3) support a "C" environment for efficiency reasons; (4) be embeddable in a larger software system with a reliable interface; (5) support development of a rule base by non-programmers by providing a simple rule syntax and symbolic rule debugging; (6) support structuring the rule base as multiple knowledge sources; (7) provide rule facilities for conflict resolution rather than provide a built-in conflict resolution strategy; (8) provide an easy-to-use explanation facility; (9) represent facts as objects; and (10) provide a demand-driven interface to the relational database that exploited the similarities between the tuples of relational databases and objects. These capabilities do not differ radically from other expert system shells but do blend many of their useful features while addressing some of the practical issues of rule-based programming. After 15 person-months of development effort, these capabilities were implemented in a shell that was embedded in a larger program handling the mouse-drive human interface. Rule base development for a prototype ASV analysis of the most complex functional area of the communications software, which required 10 person-months, was not so straightforward.

Developing the prototype rule base consisted of two tasks (a) modeling the protocol requirements of the
functional area as a state machine and (b) devising a control strategy that provided temporal reasoning. Protocol requirements modeling involved (1) formulating objects to represent the incoming and outgoing messages, operator inputs, and system states in terms of virtual relations (queries) on the relational database; (2) selecting situations of interest from the very large cross product of these objects; (3) writing rules that recognized these situations and predicted what behavior the system should subsequently exhibit; and (4) writing rules that compared the expected behavior to the actual behavior. Devising a control strategy for temporal reasoning demanded (1) precise definition of temporal concepts we all assume such as "time interval", "before", "within", and "after"; (2) a method for synchronizing the time interval under consideration with the asynchronous timestamps of the facts currently supplied by the relational database; and (3) coordinating the temporal concepts with notions of cyclic, non-cyclic, expected, unexpected, and flawed behavior. Inventing a control strategy was, by far, more difficult than modeling the protocol requirements because of the data-driven nature of rule-based programming. The lack of a built-in conflict resolution scheme helped us discover (1) flaws in our understanding of the protocol, (2) ambiguous protocol requirements, and (3) the correct control strategy. The most difficult problem of the control strategy involved switching back and forth between a coarse and fine-grained temporal analysis to resolve conflicting requirements from "simultaneous" events.

4. PERFORMANCE ANALYSIS

Based on our experience with development of the prototype ASV analysis, we have demonstrated the following benefits: (1) The syntax and semantics of rules conveniently supports modeling the test analysis and protocol requirements as state machines. (2) The relational database interface supports providing facts that contain the relevant inputs, outputs, and state information when demanded. This keeps the fact base small, reducing execution time, and alleviates some of the rule base's burden of situation identification. (3) The rule base structuring mechanism permits groups of rules to focus on small, separable analysis steps. This simplifies rule encoding and lowers the number of rules pertinent to the match step of one inference engine cycle to again reduce execution time. (4) The ability to invoke user-supplied "C" functions both as true/false predicates for the situation match/unification and as actions allows the use of compact code in those instances, such as sorting and number crunching, where procedural code is more convenient. (5) The interactive rule debugging capability eases the paradigm shift for software engineers coming to data-driven, rule-based programming from the real-time embedded software systems world. (6) The explanation facility provides a running travelogue of a test analysis, which aids in creating confidence that the test engineer's expertise has been correctly implemented in the rule base. (7) The object-oriented nature of facts permits writing rules for categories of facts that represent generalizations of more specific concepts. For example, it is sometimes useful to treat all facts that describe situations triggering specific behavior as simply trigger facts. In practice, we often created rules and classes for a specific requirement, re-engineered these rules for a similar requirement, then found the generalization that simplified the control strategy's reasoning. This anecdotal statement emphasizes that the persons writing rules were not protocol experts.

In addition, we also realized the following general benefits from using an expert systems approach: (1) The rules set can be developed in an incremental and modular fashion that grows with the understanding and expertise of the rule developer. (2) ASV reduces uncertainty in the software testing processing because the explanation facility give immediate feedback on the analysis reasoning and its limitations. (3) Associating rules with possible system states allows the use of a simple bookkeeping procedure that produces a measure of the "coverage" provided by a particular test session. (4) The rapid prototyping environment provided by rule-based programming avoids the economic necessity of salvaging past, possibly highly flawed conceptualizations of the analysis needed. (5) An unexpected side effect of writing rules that model the significant system states is that the situation part of the rules suggest those test cases that should be incorporated in a good test scenario. (6) The activity of writing rules to describe the system's expected behavior provides immediate and obvious insight into the ambiguities latent in the requirements and the limits of the rule developer's problem understanding. This suggests that development of an ASV analysis prior to design may have practical impact on software development because ambiguities and misunderstandings identified early in the software life cycle are much less expensive to correct than those identified following coding. In fact, the 1988 rule development effort paralleled the design process for an AWACS software update and the rule developers did uncover numerous ambiguities that were resolved in the on-going design process. (7) ASV creates a competition between a rule-based implementation, i.e.: an executable requirements definition, and the operational code. This unique competition between declarative and procedural descriptions avoids the aforementioned subtle shortcoming of dual programming by removing the common viewpoint of programming paradigm. (8) The expertise of the test engineer has been codified in a form that is easily transferable to other test engineers and installations. This is a significant
benefit since it typically takes 18-24 months for a test engineer to become expert at analyzing the AWACS test sessions. This codification also aids regression testing since the test analysis is consistently repeatable and does not require re-invention for each new software release.

During our experience developing the prototype ASV analysis, we suffered the following problems with our expert systems shell: (1) The lack of any built-in conflict resolution means the rule developer must always consider the consequence of multiple activations on any inference engine cycle. We dealt with this problem by imposing a regime to create control facts that track the state of the analysis. (2) Although the shell provides actions that erase facts, a uniform method of fact garbage collection was needed to prevent premature loss of information needed by the analysis. (3) The fact base has no structuring mechanism similar to the one use for the rule base. It would be convenient if "possibly relevant" facts only are considered on one inference engine cycle. We have speculated on solutions but have not implemented any.

We encountered these general difficulties with using an expert systems approach: (1) Senior personnel with a well-developed, mental paradigm for procedural programming must again endure the embarrassment and mistakes of being a neophyte programmer. (2) Real-time programmers must learn a problem approach that emphasizes object/event/situation modeling rather than functional decomposition and serialization. (3) Modifying the analysis to be more and more detailed is so easy that deciding when an analysis is reasonably sufficient is difficult.

We tested the effectiveness of the prototype ASV analysis against an AWACS baseline known to be faulty and identified both documented and undocumented errors. We also analyzed the prototype rules set and its development to establish what productivity benefits were realized from this approach. Our findings are summarized in Table I. We saw improvement in performing an analysis, the skill level needed in a test engineer, initial investment time, and, most dramatically, the re-investment time needed for regression testing of later software releases. A more thorough discussion is available in [1].

5. STATUS OF IMPLEMENTATION

The prototype rules set was completed and demonstrated to a group of Air Force software development contractors in 1987. The success of the prototype induced the Air Force to fund a field test of the concept and improvements to the ASV user interface and automated data abstraction capability. The field test, which will be completed by the end of 1988, consists of a three-person effort to develop rulesets covering six functional areas of AWACS software supporting a different communications protocol. This effort is being done in parallel with the development of the procedural code to support the protocol.

Acknowledgements

This work was supported in part by the USAF under Get PRICE [2] contract F19628-85-C-0177. The author has summarized the insights and contributions of the entire ASV project team: David Addleman, Darrell Athay, Ronald Hildahl, Richard Kelley, Bruce Mansfield, and Frank Sutton. Without their efforts, ASV would not be a success.

References

### TABLE I

<table>
<thead>
<tr>
<th>FACTOR</th>
<th>HAND ANALYSIS</th>
<th>ASV ANALYSIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Time</td>
<td>120 minutes of analysis per 30 minute laboratory session. Shows a ratio of 4 to 1 for analysis flow time and a 4 to 1 ratio for required attention from human expert</td>
<td>60 minutes of ASV execution time and 10 minutes of human participation per 30 minute lab session. Shows a ratio of 2 to 1 for analysis flow time and a 1 to 3 ratio for required attention from human expert</td>
</tr>
<tr>
<td>Minimum Skill Level</td>
<td>Domain expert, 18 months</td>
<td>Domain Novice, 0.5 months</td>
</tr>
<tr>
<td>Initial Investment</td>
<td>18 months (domain expertise + analysis skills)</td>
<td>9 months (1 month ASV expertise + 8 months domain expertise)</td>
</tr>
<tr>
<td>Re-investment</td>
<td>0-18 months</td>
<td>0-1 month</td>
</tr>
</tbody>
</table>