Some Experience with Automated Aids to the Design of Large-Scale Reliable Software

BARRY W. BOEHM, ROBERT K. MCCLEA1, AND D. B. URFRI<

Abstract—This paper summarizes some recent experience in analyzing and eliminating sources of error in the design phase of large software projects. It begins by pointing out some of the significant differences in software error incidence between large and small software projects. The most striking contrast, illustrated by project data, is the large preponderance of design errors over coding errors. Based on these observations, a hypothesis was developed regarding the potential cost-effectiveness of an automated aid to detecting inconsistencies between assertions about the nature of inputs and outputs of the various elements (functions, modules, data bases, data sources, etc.) of the software design. This hypothesis was tested by developing a prototype version of such an aid, the Design Assertion Consistency Checker (DACC), using TRW's Generalized Information Management (GIM) System, and using it on a large-scale software project with 366 elements and 957 assertions about their inputs and outputs.

Of the 121,000 possible mismatches between input and output assertions, DACC found 918, at a cost in computer time of $20. Most of the mismatches resulted from shortfalls in the initial version of DACC or the initial data preparation, such as a lack of a synonym capability and a lack of explicit statements about external inputs and outputs. However, a number of serious mismatches were exposed at a time when they were easy to correct, and a most useful work-list generated of items needing resolution before allowing the design effort to proceed to further detail.

In general, the data confirm the hypothesis about the general utility of a DACC capability for large software projects. However, a number of additional features should be considered to compensate for current deficiencies in areas such as manuscript preparation and to fully take advantage of having the software design in machine-readable form.

Index Terms—Automated aids, design validation, information systems, large-scale software projects, reliable software, software design, software productivity, software specifications, software tools, verification and validation.

Differences Between Large and Small Software Design Activities

The development of reliable large-scale software systems presents a number of problems and challenges not generally encountered in small projects. Here, a "small project" is one in which a single individual can encompass and resolve any and all of the significant macro, and micro issues involved in developing the system.

On large projects, problems such as interface definition, ambiguity resolution, management visibility, and consistency of assumptions are the dominant ones. Problems such as computational accuracy, intraroutine control, and correct syntax still exist as error sources, but are relatively less significant than they are on smaller projects.

In particular, the error distributions on large projects differ considerably from the most familiar data on software error sources: experimental studies such as those of Rubey et al. [9] and Youngs [16].

The utility of these experimental studies has been considerable; however, it is limited by the fact that each effort studied began with a clean, unambiguous statement of a stand-alone programming problem. Thus, if these studies, there was virtually no chance for errors due to interface inconsistencies, incomplete problem statements, ambiguous specifications, or inconsistent assumptions. The relative importance of these design-oriented error sources can be seen from the summary data in Table I on the relative frequency of design and programming error found in an analysis performed by TRW for RADC (Thayer et al. [14]) of the errors found in a series of five modifications to a large (100,000 source statements), generally good (on schedule, within budget) TRW software project earlier analyzed by Bosch and Hecht [3].

Design and Coding Error Types

Some indication of the differences between design and coding error types is seen in Table II which shows a classification of the 234 types of software error encountered in the original software development project referred to above. For each general error class, Table II gives the number of error types in each class which arose primarily in the design or the coding phase. The mostly-coding error types tended to involve interface problems between the code and the data bases, peripheral I/O devices, and system user.

For this project, Fig. 1 shows an even more striking contrast with respect to the relative difficulty of eliminating design and coding errors. Not only did the number of types of design error outweigh the coding error types, 64...
percent to 36 percent, but also the design errors took the longest by far to detect and correct. Of the 54 percent of the error types typically not caught until acceptance, integration, or delivery testing, only 9 percent were coding errors; the other 45 percent were design errors. Also, although no related quantitative data were collected, experience indicated that the correction of design errors consumed a great deal more effort than the correction of coding errors.

Additional analysis of the data presented by Shooman and Bolsky [10] indicates that the average time to diagnose and correct design-type errors was about twice that for coding-type errors.

<table>
<thead>
<tr>
<th>Error Category</th>
<th>No. of Error Types</th>
<th>Design (h)</th>
<th>Coding (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mostly design error types</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tape handling</td>
<td>24</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Hardware interface</td>
<td>9</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Card processing</td>
<td>17</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Disk handling</td>
<td>11</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>User interface</td>
<td>10</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Error message processing</td>
<td>8</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Bit manipulation</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Data base interface</td>
<td>19</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>About even</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Listable output processing</td>
<td>12</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Software interface</td>
<td>9</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Iterative procedure</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Mostly coding error types</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computation</td>
<td>8</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Indexing and subscripting</td>
<td>1</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. Software error sources (TRW CCIP-85 data: 220 error types).
than late in the test phase. In order to formulate incisive hypotheses about the potential payoffs of such capabilities, though, we need a better understanding of the causes of errors and a more detailed analysis of the specific error types encountered. These are covered in the next two sections.

A TAXONOMY OF SOFTWARE ERROR CAUSES

Error Causes by Phase and Category: Particularly on large projects, but also on small projects, a great deal of error-prone activity goes on before coding begins. First, an attempt is made to determine what information processing capabilities the users need. In general, this is imperfectly determined and resembles more of a statement of what users know they want. (For example, they may not know they need data base archiving until the first time their irreplaceable file is destroyed.) Next, the various statements of need or want are pulled together (imperfectly) into a functional specification for the software. Subsequent imperfect transformations yield a system specification (preliminary design) and a detailed specification. Once the detailed specification is reviewed and approved, the final transformation into code (and, to some extent, some consistency problems in transforming detailed specifications into code) is performed.

During each of the above transformations into successively refined software specifications, several types of error can occur.

1) Communication: There was a misunderstanding of the requirements expressed in the previous stage.

2) Completeness: There was an incomplete grasp of the requirements expressed or implicit in the previous stage.

3) Consistency: The requirements were well understood, but conceptual errors were made in implementing them at the next stage.

4) Clerical: The requirements were well understood, but clerical errors were made in implementing them at the next stage.

Table III presents a taxonomy of software error sources, giving for each transformation an example of each error type. The data archiving example cited above, for example, is classified as a lack of completeness in transforming user needs into a statement of user wants. The other examples are fairly self-explanatory. The definition of communication error is slightly extended in the coding-phase example ("assume OS manual correct") to cover the class of errors resulting from faulty or misunderstood documentation of support software.

USES OF THE TAXONOMY

One useful aspect of such a taxonomy is as a means of assessing the relative power of alternative techniques to eliminate software errors. For example, proof techniques attack only the types of errors occurring after a specification is available; i.e., errors in the bottom three rows. goto free programming per se only attacks a subset of consistency problems in transforming detailed specifications into code (and, to some extent, some consistency problems in the previous two transformations).

The taxonomy has also been useful as an aid to developing and testing hypotheses about the potential utility of possible new tools for preventing, detecting, and eliminating software errors, particularly in the requirements formulation and design phase. The next section in this paper describes the data analysis and subsequent hypothesis formulation which led to the development and evaluation of such an automated aid to reliable large-scale software design, the Design Assertion Consistency Checker (DACC).

SOFTWARE ERROR DATA ANALYSIS

Table IV (Boehm et al. [1]) shows the first segment of a more detailed analysis of the 224 types of errors on the

---

1 However, the enhanced understandability and testability of such programs generally makes it easier to find and correct other types of errors.
software project summarized above. It includes information on when each type of error typically originated (0) where it was typically found (F), and when various potential text-scanning capabilities would have typically detected the error. For example, the CS-13 capability cited in the table refers to a capability which would scan standard software module header blocks. It would check the consistency of their information, with respect to assertions in the header blocks about the nature of inputs and outputs, including:

1) data type and format;
2) number of inputs;
3) order of inputs
4) units;
5) acceptable ranges;
6) associated storage locations;
7) source (device or logical file or record);
8) access (read-only, restricted access).

The CS-13 entries in Table IV indicate that if coding of the module had been preceded by such a module description with the assertions about its inputs and outputs, then an automated consistency checker could generally have caught the error before coding began.

Of the 12 types of card processing errors shown in Table IV, the CS-13 consistency checker would have caught 6. Overall, out of the 224 types of errors, this capability would generally have caught 18. The next most effective capability would perform checks on the consistency of the actual code with the module description produced during the design phase for capability CS-13 above. (For example, for each output assertion, it would check if the variable appeared on the left of an equals sign in the code, and perform a units check on the computation.) This capability would have caught 10 types of error, but not until the initial code-scanning phase. A total of 29 text-scanning capabilities would have caught at least one of the 224 error types, but none of the others would have caught more than 7. More detailed results of this analysis are in Boehm et al. [1].

### THE DESIGN ASSERTION CONSISTENCY CHECKER (DACC)

**Generalized Information Management (GIM) Support Structure:** Based on the above data analysis, a hypothesis was derived regarding the potential cost-effectiveness of a module input–output consistency checking aid along the lines of capability CS-13. To test this hypothesis, such a capability was developed, the DACC, and tried on a large software specification. By using TRW's GIM system [18], DACC was built in a very flexible way within a relatively short time.

The GIM system provides a set of general capabilities for organizing, manipulating, and interrogating data.

### TABLE IV

**Evaluation of Error-Detecting Capabilities (Metrics) Versus Error Type (First 12 of 224 Error Types)**

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Software Phase</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Errors in Preparation or Processing of Card Input Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Program expects parameter in different format than given in Program Requirement Specification.</td>
<td>Requirement</td>
<td>0</td>
<td>CS-13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Program does not expect or accept a required parameter.</td>
<td>Code</td>
<td>0</td>
<td>CS-13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Program expects parameters in a different order than that which is specified.</td>
<td>Development</td>
<td>0</td>
<td>CS-13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Program does not accept data through the entire range which is specified.</td>
<td>Test</td>
<td>0</td>
<td>CS-13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Program expects parameter in units different from that which is specified.</td>
<td>Validation</td>
<td>0</td>
<td>CS-13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Nominal or default value utilized by program in the absence of specific input data is different from that which is specified.</td>
<td>Acceptance</td>
<td>0</td>
<td>CP-9</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. Program accepts data outside of allowable range limits.</td>
<td>Integration</td>
<td>0</td>
<td>CP-9</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. Program will not accept all data within allowable range limits.</td>
<td>Delivery</td>
<td>0</td>
<td>CP-9</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. Program overflows core tables with data that is within the allowed range.</td>
<td></td>
<td>0</td>
<td>CS-13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10. Program overflows allotted space in mass storage with data that is within the allowed range.</td>
<td></td>
<td>0</td>
<td>CS-13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11. Program executes first test case properly but succeeding test cases fail.</td>
<td></td>
<td>0</td>
<td>CS-13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12. Program expects parameter in a different location than specified.</td>
<td></td>
<td>0</td>
<td>CS-13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 = Error origin  CS-N = Consistency-checking aid N applied at this phase would generally have detected error.
F = Error found  CP-N = Completeness-checking aid N applied at this phase would generally have detected error.
bases in either batch or on-line modes. Versions exist for both IBM 360-370 and Univac 1100-series equipment. Its usage in building the DACC capability is best illustrated through examples of its general design description capabilities. The examples used are from the spacecraft data processing specification chosen as a test case to evaluate DACC.

**DACC Design Element Report Format:** Fig. 2 shows one of the general summary reports of the design element information presented to DACC. Besides accommodating information about the inputs and outputs of system elements, it provides capabilities for storing, updating, manipulating, and reporting a number of other attributes: computational frequency, hierarchical level (system, subsystem, function, routine), mission phases (prelaunch, boost, orbit, etc.), assumptions, references to applicable source documents, etc. The assertions about inputs and outputs included their intended source and destination (optional), their dimensionality, the units and coordinate system in which they were expressed, their type (fixed point, floating point, alphanumeric, Boolean, etc.), and the range of acceptable values (minimum and maximum). Type and range information was not generated in the test application.

**Design Information Representation:** Figs. 3 and 4 show two of the forms which describe how the above information is represented in the GIM system. Definitions of the attributes represented in the first few columns are shown below; the others are self-explanatory.

**Attribute Mark Count (AMC):** A GIM system parameter used to identify the position of an attribute within a file record. **Maximum Length (MAX LEN):** Defines the maximum number of characters allowed in an attribute field.
Type: Contents of the field stored in each attribute, e.g.,

A = Alphabetic,
N = Numeric,
A/N = Alpha/Numeric.

Store Type: Defines the method in which data are to be stored in each attribute; e.g.,

SV = Single value
MV = Multiple values
NS = Nonstore; the user may not physically enter any value.

For example, the first line of Fig. 3 shows that the initial item in the file (with a mark count of 0), is the identifier of the design element or requirement (REQT), represented as three fields separated by asterisks, the first two with ≤3 characters and the third with ≤6 characters (3*3*6). Each of these fields contain single (SV) alphanumeric (A/N) character strings, as shown by the example in Sample Entries in Fig. 3.

Design Data Input: A special purpose input form was used to gather and enter the software design information into DACC. Additional forms were provided for the assertions about inputs and outputs. For many of the attributes, standard definitions of the acceptable variables were given (G for Guidance, N for Navigation, PM for Payload Management, R for radians, B for body-axis coordinate system, N/A for not available, etc.). The bulk of the information was entered thus in a batch mode. The on-line mode was quite useful for additions and corrections; examples of such commands are the following:

ADD TO REQT/G*A*1.0/PAGE "BOOST."
DELETE REQT/G*A*1.0/PAGE.
CHANGE INPUTS/PM*A*1.1*2/3/MAX "56" TO "57."

Design Query Capabilities: Further, the general on-line query functions of GIM provided a number of useful capabilities for DACC besides just consistency checking. For example, here are some commands which can be used to query data base:

LIST ANY REQT AND FUNC-NAME WITH PHASE "BOOST."
COUNT INPUTS WITH SOURCE "G" ANDD WITH DIM "2."
LIST REQT /G*A*3.0/ INPUTS.

(In GIM, ANDD is the logical "and," and is a "convenience connective" which improves readability but is ignored by the interpreter.)

EXPERIENCE WITH DACC ON A LARGE DESIGN SPECIFICATION

The DACC system was tried on a large-scale spacecraft software requirements and design specification which had been compiled by a team of about 10-12 engineers. A total of 186 design elements were entered into the system, with assertions about the nature of 314 inputs and 453 outputs.

Fig. 5 shows the first page of the resulting Inconsistency Report produced by DACC. It includes not only mismatches between assertions about inputs and outputs, but also indications that certain expected inputs are not being produced by any other element, or that certain outputs are not being used. For example, the first entry shows that a certain Guidance Subsystem (G,0,3) produced as its fourth output (4.0) the variable V RESIDUAL, but that this quantity was not being used by any other of the 186 subsystems (NOT USED). The second entry shows that one Guidance Subsystem expected the input V VECTOR to be defined in the Mean 1950 Cartesian Coordinate System (M.), but that the Guidance Subsystem producing V VECTOR was doing so in the Orbiter Body Axis Coordinate System (B). The third entry shows that a Display and Control Subsystem (D,GOS.1) expected the input abort ALTERNATIVE OPTIONS, but that no other subsystem was producing these (NOT AVAILABLE).

Of the approximately 121 000 possible mismatches between inputs and outputs, there were 818 reported by the Inconsistency Report. Of these, 783 were NOT used or NOT AVAILABLE statements, and 35 were mismatches between input and output assertions about the same variable. As shown below, most of these were symptoms of deficiencies in the initial version of DACC than of problems with the software design. But there were a number of actual design mismatches found, which otherwise probably would not have been discovered until late in software
testing. Some of the mismatches were difficult to classify uniquely; thus, some of the summary results below are approximate rather than exact counts.

1) Over 50 percent of the mismatches were NOT USED or NOT AVAILABLE statements which would not have occurred had there been a means to handle inputs and outputs from the external environment. One way to avoid these would have been to include an "external" tag which would trigger a bypass of the consistency checker. However, as many software errors come from erroneous assumptions about the nature of external inputs and outputs, it would be preferable to have assertions about such items explicitly entered into DACC via dummy design elements, or explicit "real world entities" as in ISDOS (Telchroew and Sayari [13]).

2) Over 25 percent of the mismatches were NOT USED or NOT AVAILABLE statements which occurred because of differences in terminology on the names of inputs and outputs. For example, 12 elements specifying PRESENT ORBITER STATE VECTOR as an input were told it was not available, when in fact it was being generated by another element under the name ORBITER STATE VECTOR. Most of these mismatches would be eliminated by either of two steps. One would be to have established standard terminology for inputs and outputs as with units, coordinates, etc.; the other would be to incorporate a synonym capability into DACC. Actually, it is advisable to incorporate both of these steps.

3) Over 10 percent of the mismatches were NOT USED or NOT AVAILABLE statements which indeed referred to inputs and outputs which had not been entered into DACC. Most were being generated and were not yet loaded into DACC, but some were clear oversights which might not have been caught until the software integration phase.

4) Of the 35 mismatches between assertions, 25 were either clerical errors or nonstandard terminology (RAD for radians). Three had one NOT AVAILABLE statement matched with one positive description. The remaining seven were actual mismatches between assertions about dimensions, coordinate systems, and units (e.g., degrees versus radians). These in particular might not have been caught until very late in the testing and integration phase, and are typically the kind of errors which are difficult to correct, as their correction often causes ripple effects into other portions of the software.

In addition, there were a number of second-order effects which were picked up once the first mismatches had been corrected. For example, there were several units terminology mismatches which were flagged once the ORBITER STATE VECTOR terminology was reconciled.

**EVALUATION AND CONCLUSIONS**

**Useful Features:** The experiment showed that a capability such as DACC has considerable value in detecting interface errors in large-scale software at a point where they are easy to resolve. Several of the mismatches detected were of types which characteristically on earlier
projects were not detected until the very late stages, when they were often quite difficult to resolve.

The other major value of the Inconsistency Report was as an engineering work-list generator: the not used and not available outputs provided a checklist of potential interface problems which should be resolved before proceeding to more detailed design. Many of the items were simple matters of terminology resolution, but some involved more fundamental conflicts of assumptions about "who was furnishing what to whom."

In general, the machine-analyzable design information provides a compatible extension to the manual HIPO (IBM 1973) and Structured Design (Stevens et al. [11]) techniques. In addition, the GIM capabilities underlying DACC were found quite useful for the processes of updating and interrogating the data base of design statements. The capability can also be used on software requirements specifications when they are expressed in terms of functional elements.

Costs: The cost of determining the mismatches and running the Inconsistency Report was a very nominal $30. However, it cost several hundred dollars to enter the design information into the GIM data base. This expense would not be considered a difficulty if it replaced typists generating the same information for official design documentation. For highly structured design information, the GIM report generator works quite well, but its manuscript preparation capabilities for free form text were not sufficient to generate acceptable contract reports for the project. This difficulty in the report-generation area has been the main impediment to regular use of the system, and has been a major consideration in evaluating alternative realizations of DACC and related design capabilities.

Additional Design Features: Some of the additional desired features would simply round out the basic consistency-checking capabilities of DACC, e.g., a synonym capability, better report generation, capabilities for better accommodating assertions about external data and about data bases. Another general reaction to the DACC experiment was a feeling that "once all that information is available in machine-readable form, there are some other useful things I would like to have done with it." Most of the capabilities suggested were similar to those proposed or under development in such systems as ISDOS (Teichroew and Sayari [13]), TOPD (Henderson and Snowdon [5]), and ZYGO (Swanson [12]), including:

1) Representation, analysis, and graphic presentation of the control hierarchy of the design elements;
2) Incorporating reachability considerations in the input-output consistency analysis;
3) Generating N-square charts summarizing data flows;
4) Accumulating the individual statements about inputs into first-cut versions of a data base description, an input data requirements list, and an output data summary;
5) Providing traceability information to and from software requirements and detailed design;
6) Accepting additional information on the performance characteristics, memory and source-instruction estimates for design elements and using them to produce first-cut core budgets, software schedules, and inputs to system performance models.
7) Integrating these process-oriented design considerations with properties-oriented capabilities focused on enhancing maintainability, reliability, portability, etc. (Boehm [2]).

Work is currently underway at TRW to assess existing methodologies such as those cited above and those developed to support existing TRW projects, and to determine an integrated architecture for a Design Description Language and a support system called Design Expression and Validation for Information System Engineering (DEVISE), incorporating as many of the desired features as appears initially feasible.

Another open issue involves how best the job of data-type consistency checking should be split between the design phase (via such vehicles as DACC) and the compilation and execution phase, where a number of efforts are or have been focusing their attention (Hansen [4]), (Hoare [6]), and (Liskov and Zilles [8]). It may well be that the answer will turn out differently for large software projects than for small ones. As pointed out above, large projects have a good deal more difficulty ensuring consistency of design terminology and assumptions, and are likely to require more extensive automated design aids than small projects, on which a single unifying individual can perform such functions. However, given the difficulty of coping with all of the multifarious sources of software error, it is probably a safe assumption that both approaches will prove useful to individuals and organizations attempting to produce reliable software.

ACKNOWLEDGMENT

Several individuals at TRW made useful contributions to the formulation and development of DACC: J. H. Drexler, R. C. Eberhard, R. L. Eshbaugh, F. C. Manthey, J. H. Petersen, and J. A. Sewell.

REFERENCES

2. B. W. Boehm, "Some steps toward formal and automated aids to software requirements analysis and design," in Proc. IFIP Congr., 1974, pp. 192-197.
Barry W. Boehm, for a photograph and biography, please see this issue, p. 1.

Robert K. McClean was born in Toronto, Ont., Canada. He received the B.A.Sc. degree in 1959 and M.A.Sc. degree in 1961 in electrical engineering, both from the University of Toronto, Toronto, Ont., Canada. From 1961 to 1963 he worked on aircraft and missile dynamics and control problems at Northrop Corporation. During the 1963-1972 period, at the Aerospace Corporation, he developed vehicle dynamics simulation and guidance software for the USAF Titan III Program. He also provided planning and direction for the USAF on RAD projects for the development of higher order languages, LSI chip design aids, LSI spacecraft computers and computer aids to support spaceborne software design and implementation. He has spent the last two years at TRW Corporation, Redondo Beach, Calif., performing research and development on command and control systems exercising, application of microprogramming to software development and testing, and software requirements and design definition aids.

Mr. McClean is a member of the Association for Computing Machinery.

D.B. Ergin received the B.S. degree from the University of California, Los Angeles, and the M.S. degree from the University of Southern California, Los Angeles. He was Manager of a DOD Space Shuttle On-Board Software Requirements Study just completed for BAMO and an earlier DOD/STS Mission Planning and Software Requirements Study. His 12 years of experience include direct supervision responsibility for Targeting and Trajectory Data Book generation for Minuteman III RAD launches. He supervised pre-flight and post-flight mission analysis tasks for Minuteman III operations training launches. Architect of Automatic Trajectory Analysis programs for both Minuteman mission analysis and Apollo flight program verification. He was Task Manager for independent verification of the Primary GNC Flight programs for the Apollo Lunar Module. He is now Project Manager for TRW Corporation, Redondo Beach, Calif.