The Enhanced Condition Table Methodology for Verification of Critical Software in Ada and C

M. Hecht, K.S. Tso, and S. Hochhauser

SoHaR Incorporated
8421 Wilshire Blvd., Suite 201, Beverly Hills, CA 90211
(213) 653-4717

1 Background and Motivation

As the software in control and other critical systems grow more complex, it is necessary to segregate certain critical portions of the code which is involved with preventing major catastrophes and keeping the system alive from other code which may perform normal functions. This segregated portion of the code, which we refer to as the *hard core,* requires special verification measures well beyond the normal practices of unit testing at the functional or structural levels. This paper and the accompanying viewgraphs describe a test-based methodology developed for this purpose.

Usual structural testing techniques such as path or branch testing are inadequate for such software, and other methods must be developed. The work described in this paper and the accompanying viewgraphs investigated enhancement of a verification methodology based on work performed by J. Goodenough and S. Gerhart based on condition tables [GOOD75]. Their method required testing not only all *paths* through the software, but all feasible combinations of *conditions*. The distinction between path testing and condition table testing is that in the former, the completion criterion is that all feasible paths are traversed at least once; in the latter, paths will be traversed many times with significantly different data. As a result of multiple traversals, it is more likely that coding errors such as incorrect conditions (e.g., IF A>B rather than if $A \geq B$), incorrect operations (e.g., A=B+C instead of A=B*C), or missing branches (e.g., check for a value not being zero before dividing) will be detected.

The difficulty with the method is that the effort to develop condition tables and test cases can be much more than that required to develop the software undergoing test. Thus, it is impractical for full scale software development projects. Over the past three years, we have been investigating methods to improve the methodology by automating labor intensive portions and incorporating functional knowledge related to critical failure modes. The enhancements that were developed include:

- Automated tools to generate the condition table implemented for both C and Ada
- An analysis format called the Test Case Enhancement Analysis (TCEA) for developing additional test cases which have functional, reliability, or safety significance
- Implementation rules which simplify the generation of condition tables, test cases, and the creation of a test environment

2 Methodology Description

The steps for developing an enhanced condition table are:

1. Develop a condition table based on the code

2. Develop additional test case specifications by resolving "don't care" conditions into test cases which relate to specific functional, safety, or reliability concerns

3. Define test cases which satisfy the specifications developed in the first two steps

4. Create the test environment, run the tests and analyze the results.

Viewgraph 7 shows a simple C code segment for managing a fault tolerant system consisting of two nodes, designated node1 and node2. If the outputs of the two nodes agree, then no further processing occurs. However, if there is disagreement between the nodes, then the first node is checked. If the check function returns a value of not OK (i.e., the node has failed the check), then the fail_node1 variable is set to TRUE. If the check function returns a value of OK (i.e., the node has passed the check), then the second node is checked. If the second node check value is OK, then a retry function is invoked. However, if the check indicates a failure, then the *fail_node2* variable is set to TRUE. If either of the two nodes have failed, then a reconfiguration function is invoked.

The resultant condition table is shown on the same viewgraph. The predicates from the 4 conditions are shown in the left hand column. The feasible combination of these predicates, called "rules", are shown in the 4 succeeding columns. The notation is as follows: y: condition set true; n: condition set false; (y) condition is necessarily true because of the state of other conditions; (n) condition is necessarily false; and -: irrelevant or "Don't Care". This format is an adaptation of the limited entry condition table first proposed by King [KING69].

The "Don't Care" conditions are the ones of concern in the condition table methodology. Under the Goodenough and Gerhart approach, the right-most column of the table would have to be decomposed into 8 (2^3) additional rules for which test cases would have to be written. The ECT approach instead requires that the analyst consider significant failure modes of the module using a format called the Test Case Enhancement Analysis (TCEA) shown in Viewgraph 7. The TCEA shows that only one additional case needs to be run: that the two nodes should agree *and* that they should be failed. This rule uncovers a significant flaw in the routine because such a case is not properly handled. Instead ordering a reconfiguration, the implementation of this module results in no action at all.

3 Tools Development

The tools development objective was addressed by the creation of two separate programs called *ECT* and *SEM*. The ECT program performs the following operations:

1. Lexical analysis that assembles terminal symbols (e.g., carriage return/line feed) and eliminates white space and comments

2. Syntactic analysis that parses C or Ada programs and detects grammatical errors

3. Generation of a condition tree

4. Generation of a condition table from the condition tree.

The condition table generated by the ECT program is syntactically and structurally correct but contains many semantically infeasible rules. The second program, SEM, reduces the condition table generated by *ECT* based on an input file which contains the semantics of the program.

Semantic information is input to the SEM program by means of an ASCII file using the following notation:

&	AND
|	OR
!	NOT
->	Implies that
~	Don't Care

For the code segment shown in viewgraph 6, the semantics are

!c2 | !c3 -> c4 (if the check on node 1 or node 2 is OK, then the node can not have failed).

The *ECT* and *SEM* tools decompose multiple conditions into unary conditions and generates tables based on the relationships of the multiple conditions. That is, a condition based on two predicates, e.g., if (a < b) and (c < d), becomes two conditions using the *ECT* and *SEM* programs; under the original approach, they would be regarded as a single condition. A second enhancement is that case and *while* statements are handled by the *ECT* and *SEM* programs whereas the original work dealt only with if-then-else constructs. *ECT* and *SEM* were implemented in C on a Sun SPARCstation I computer running SUNOS (UNIX 4.2 BSD), release 4. Figures 1 and 2 shows the top level structure of these programs; the parsing and lexical analysis portions were generated using the UNIX *yacc* and *lex* programs.

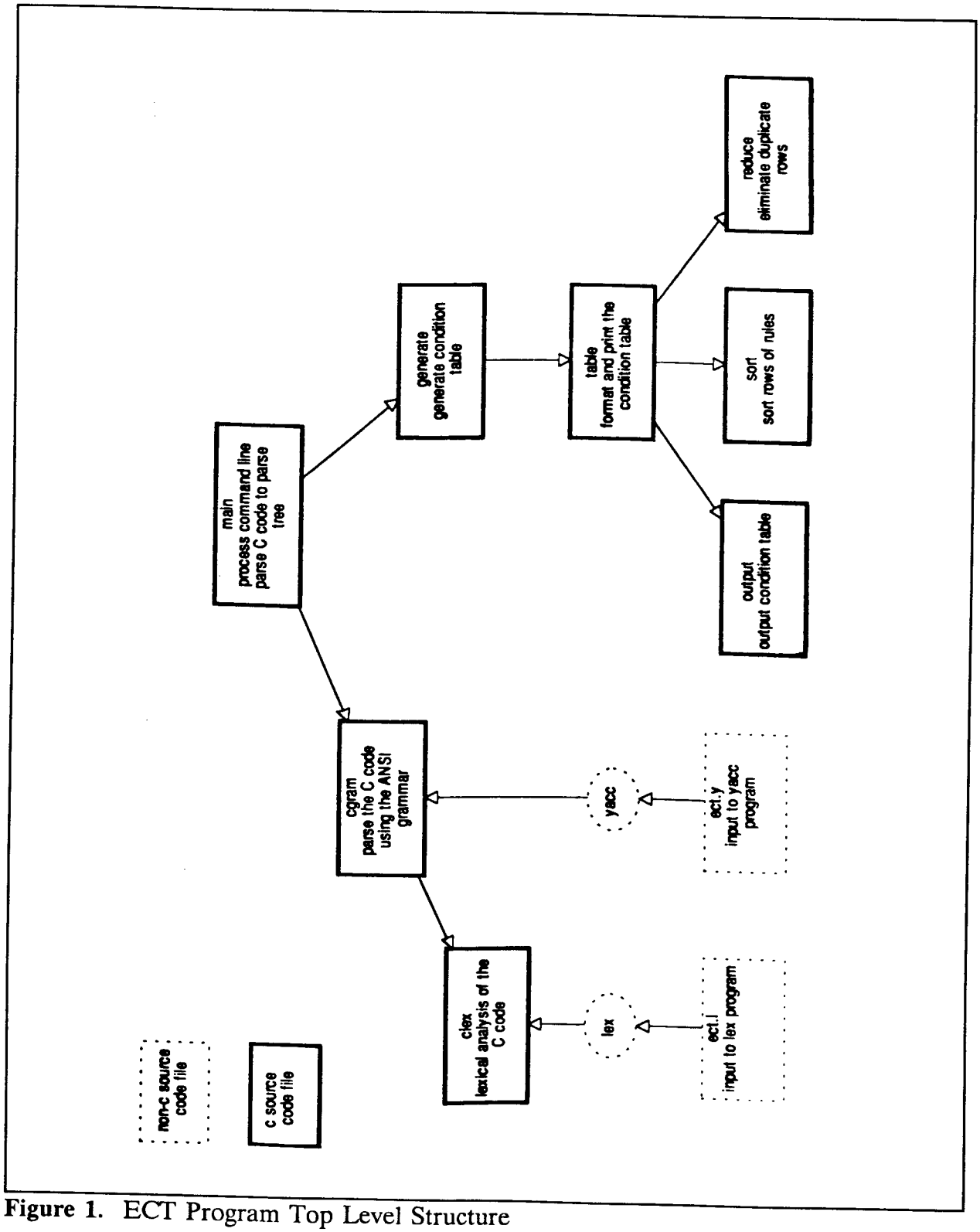

Figure 1. ECT Program Top Level Structure

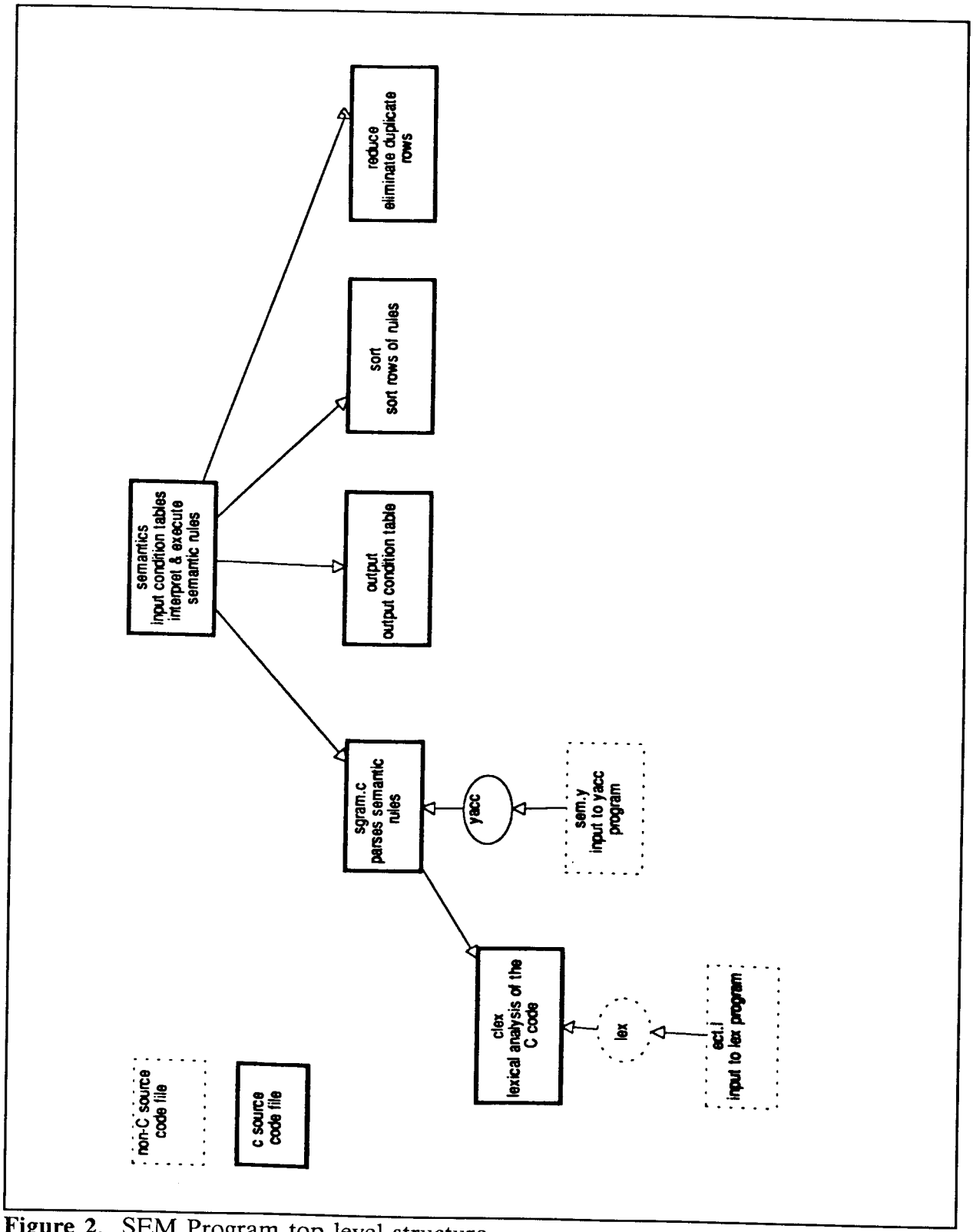

Figure 2. SEM Program top level structure

4 Experience

The ECT methodology has now been exercised on three software systems whose size and complexity is representative of the intended target systems. The following table lists these systems together with the steps in the methodology that have been performed.

System	Size (SLOC)	Language	Steps Completed
EDRB (Distributed Fault Tolerant Real Time Control System)	2000	ANSI C	ECT, SEM Test Case Generation TCEA Testing and Analysis
Control code for a storage subsystem from a major hardware manufacturer	>10,000	Microsoft C	ECT, SEM Test Case Generation TCEA for small segment
SIMTEL repository Ada modules	>10,000	Ada	ECT, SEM

As shown in the table, the most complete analysis has been conduced on a a fault tolerant distributed reactor control system being developed under contract to the Department of Energy. The system, called the Extended Distributed Recovery Block, or EDRB, consists of approximately 2000 lines of ANSI standard C code organized as 17 tasks in a multitasking operating system. A description of the EDRB system may be found in [HECH89].

Generation of condition tables was performed using the automated tools described above for all stable portions of the EDRB. The accompanying viewgraphs show an excerpt of code which is typical of the EDRB and the condition table generated by the ECT and SEM tools described in the previous section. The rules are listed in a horizontal format rather than the vertical format shown in the previous section because of the large number of rules. 700 rules were generated for the 30 modules. No difficulties were experienced in running the tools. The entire procedure took less than 1 hour, most of which was related to file transfers.

The next step was generation of the test cases from the rules defined by the condition table. In this paper, we will discuss test case generation for a task called HEARTBEAT. This task is responsible for generating its own heartbeat, monitoring that of its shadow, and deciding whether to synchronize with its shadow or signal the system supervisor for a recovery action. A total of 109 test cases were developed for the routine. Multiple test cases were developed for some rules in order to test special values (i.e., values at or close to boundaries, discontinuities, etc.). Figure 3 shows two test cases that were generated for Rule 40. They differ in that the frame counts in one case are 99 and 100 whereas in the second they are 100 and 103. These two different sets of values test the limits of the behavior of the program in this branch. This approach exemplifies the combination of structural testing with other forms of testing in order to provide complete coverage. The fault found using the ECT which was not found using conventional inspection-based methods was due to a second test case on the 66th rule in the condition table.

The third step was generation of additional test cases that reflect concerns on the function of the module. A procedure was developed to generate such test cases and resulted in the formulation of the Test Case Enhancement Analysis, or TCEA, an excerpt of which is shown in Viewgraph 6. By using the TCEA, an additional several hundred rules were developed to resolve "Don't Cares" into y and *n* outcomes. However, the generation of these enhanced test cases was simplified by the fact that they are readily derived from existing test cases and their outcomes will be precisely identical to the existing test cases from which they were derived.Execution of the test cases required (1) modification of the source code, (2) a specially written test driver routine which reads in the test cases and outputs the results, and (3) stubs to substitute for subroutines and functions invoked by the unit under test. As will be discussed in the next chapter, creation of a unique test environment for each task is a labor intensive process which could be eliminated by an appropriate debugger capable of setting variables, tracing execution, and output results for off-line analysis.

5 Results and Discussion

Success in Use of Automated Tools

The most significant result of this work was that the automated tools *ECT* and *SEM* have eliminated a tedious and error-prone step in the verification process. The tools have been used to analyze hundreds of pages of source code in both C and Ada, and have generated many tens of thousands of rules for

A related result is that automated tools provided a rapid unambiguous indication of excessive complexity. For example, one task in the EDRB had 20 conditions and 1050 rules. This result was in and of itself sufficient motivation for recoding of the module. Because these tasks were not contrived examples, it is reasonable to assume that the tools used to generate these condition tables can be used for any code which has less than 15 conditions.

Differences in Rule Generation Between Ada and C

The Ada version of the ECT tool required significant modification from the C version. Extensions that had to be handled include

> C1 XOR C2: Each rule incorporating this condition has to evaluated for the additional condition that C1 and C2 could *both* be true in addition to the cases that either C1 *or* C2 could be true.
>
> IF C1 AND THEN IF C2: Both C1 and C2 must be evaluated in Ada (note that the evaluation of C1 and C2 could involve the invocation of a complex set of conditions in and of itself). By way of contrast, if C1 were false in the C language, C2 would not be evaluated.
>
> Nested Procedures: Additional sets rules had to be generated for procedure within a procedure
>
> Probelms of Parsing Complex Expressions: Viewgraph 16A shows examples of complex conditions within the HEAPSORT procedure of the Sort_Utilities generic package of the Army SIMTEL Ada repository. These expressions are significantly more complex than the C expressions shown in viewgraph 14.

Routine: hb.c
Rule no: 40
Test Case no: 40.1

Desc. Author: S. Hecht
Date: June 30, 1989

	Condition	Value
C1	!(mon=name_locate())	n
C2	my_count<0	n
C3	insync	y
C4	TICKSPERFRAME>time	y
C5	need_sync()	y
C6	initial	-
C7	status==mon	-
C8	comp_count>=0	-
C9	initial	-
C10	need_sync()	-
C11	comp_count>=0	-
C12	initial	-
C13	need_sync()	-
C14	creceive()==trans	y

Variable Name	Applicable Condition /role*	Input Value	Expected Output	Observed Output
my_count	!c1/s,c2/t	100	101	101
buf1(comp_count1,elapsed1)	c3/s	103,0	103,0	103,0
buf2(comp_count2,elapsed2)		0,0	0,0	0,0
buf3(comp_count3,elapsed3)		0,0	0,0	0,0
insync	c3/t,c5/s	1	0	0
time	c4/t	6	6	6
need_sync1	c5/t	1	1	1
need_sync2		0	0	0
need_sync3		0	0	0
initial	!c1/s	1	0	0
send2mon1	c3/s	0	1	1
send2mon2		0	0	0
send2mon3		0	0	0
monstat		0	0	0
transrep	c14/s	-1	101	101
transnam	c14/t	trans1	trans1	trans1

* /t - variable tested only
/s - variable set
/b - variable both tested and set

Routine: hb.c
Rule no: 40
Test Case no: 40.4

Desc. Author: S. Hecht
Date: June 30, 1989

	Condition	Value
C1	!(mon=name_locate())	n
C2	my_count<0	n
C3	insync	y
C4	TICKSPERFRAME>time	y
C5	need_sync()	y
C6	initial	-
C7	status==mon	-
C8	comp_count>=0	-
C9	initial	-
C10	need_sync()	-
C11	comp_count>=0	-
C12	initial	-
C13	need_sync()	-
C14	creceive()==trans	y

Variable Name	Applicable Condition /role*	Input Value	Expected Output	Observed Output
my_count	!c1/s,c2/t	99	100	100
buf1(comp_count1,elapsed1)	c3/s	100,2	100,2	100,2
buf2(comp_count2,elapsed2)		0,0	0,0	0,0
buf3(comp_count3,elapsed3)		0,0	0,0	0,0
insync	c3/t,c5/s	1	0	0
time	c4/t	6	6	6
need_sync1	c5/t	1	1	1
need_sync2		0	0	0
need_sync3		0	0	0
initial	!c1/s	1	0	0
send2mon1	c3/s	0	1	1
send2mon2		0	0	0
send2mon3		0	0	0
monstat		0	0	0
transrep	c14/s	-1	100	100
transnam	c14/t	trans1	trans1	trans1

* /t - variable tested only
/s - variable set
/b - variable both tested and set

Figure 3. Two Test Cases for Rule 40

Traceability of Test Cases and Results

The ECT methodology provides a complete and traceable test program. Traceability of all conditions in the code is provided through the automatically generated condition table. Safety and reliability analyses are traceable to the TCEA. The ECT and TCEA together form a test specification with unambiguous completion criteria. The test cases and results are in turn traceable to the test specification. This traceability makes the ECT a manageable process and facilitates IV&V, customer review, and regulatory agency review.

Faults Found Using the ECT Methodology

The ECT methodology is useful in uncovering subtle faults. For example, in the HB routine synchronization a special values test in a feasible path created a state in which a "stale" heartbeat count from the companion was greater than or equal to the local count. The local node synchronized on this stale count. Analysis of the anomaly showed that although the HB routine synchronizes on the frame *number*, it does not consider the age of the heartbeat message. This age is measured by a variable called *elapsed*, which counts the number of tenths of a frame that have elapsed since receipt of the most recent heartbeat. When the HB routine requests the companion heartbeat count from the monitor task, it also receives the elapsed value. If elapsed is greater than 2 ticks, then the program specification defines the nodes as no longer being in synchronization. However, in this case, a program variable called *insync* which indicates whether the nodes are in synchronization, was set to true even though no synchronization had actually occurred.

Earlier work with the ECT methodology also found a subtle fault [TAI87] in a fault detection and recovery section of a smaller module. A possible explanation of the nature of faults found by the methodology is that more obvious coding errors will have been detected during development and testing by the original software implementer. Thus, only errors in infrequently exercised paths with unusual values, i.e., "subtle" errors, will remain. However, previous research on data from the JPL Deep Space Network [MCCA87] has shown that such errors account for a disproportionate number of system failures. The structural aspect of the ECT methodology considers all code without regard to the functional aspects of the program. Because functionally oriented testing by the original developers will have occurred before the start of verification, only the subtle errors will remain.

Implementation Practices for Testability

A significant result of this research is the importance of designing and implementation of code to ensure testability. The following rules were found to be effective:

1. *No more than 12 conditions per module*: One measure of complexity of a module is the number of branches formed by if, while, else, and related conditions. As modules become more complex, they become more difficult to verify using either manual or automated approaches. Although the automated portions of the ECT methodology can handle modules of more than 20 conditions, they are difficult to understand and have an excessively large number of rules.

2. *Minimize setting of variable after using*: If the same variable is set and used several times for each execution or iteration of the unit under test, then it is difficult to conclusively evaluate its success because intermediate values are obliterated before the results are output. Following this rule allows

the return values of *function* to be written to a record without an undue number of changes to the code and is particularly important for operating system calls and other black box modules. This method also reduces unintended interaction effects.

3. *Use parameters for subroutine calls, minimize use of global variables and pass subroutine arguments by value*: The principal advantage of information hiding for the purpose of the ECT is that it makes writing of a test driver easier and facilitates the creation of test cases. The other advantages of this rule are well known and would apply in general to high quality software.

4. *List parameters in the following order: (1) input parameters, (2) input/output parameters, and output parameters*: Although the input and output parameters must be separated in Ada, other languages such as C and FORTRAN, do not require explicit separation. The primary motivation for this rule is to facilitate testing. However, it also reduces the probability of inadvertently switching arguments or misunderstanding.

Following these rules reduced the number of changes by more than 60% in the modules tested. This reduction results in a more credible verification and also reduces the test effort.

Test Case Generation

The approach to test case generation for the EDRB was to manually define the state of input values so that the path for each case was known a priori. The usual approach is to instrument the code and vary the input (many path testing programs vary the input randomly) within predefined ranges in the hopes of traversing most branches. When the automated testing is concluded, manual test cases are created only for the untraversed paths. The benefit of the a priori approach are:

1. A large reduction in the number of test cases that must be generated, stored, and evaluated

2. Explicit traceability of each test case to a rule, special values, and path

3. Easy creation of enhanced test cases generated from special values analysis and the Test Case Enhancement Analysis.

Test cases examine the states of internal variables as well as the input and output. Thus, testing requires the manipulation of internal variables, dynamically changing parameters, and other intrusive actions. The test environment can be entirely custom developed or can be built from existing tools within the operating system. In the QNX [QUAN88] operating system under which the EDRB runs, a test environment was specially written to read in test data, output results, and set values of internal variables and dummy variables substituted for operating system functions.

Different considerations would apply in more sophisticated software development environments. For example, the *dbx* tool in many implementations of UNIX 4.2 can set all internal variables and print out all output variables without the need for a driver routine; the test input can be read in through a script file and the output can be directed to the appropriate output file. Thus, the first and second principles would no longer apply. *dbx* also has some capabilities to control interaction with the operating system thereby reducing the need to replace calls to the operating system with test data variables.

Promising Areas for Additional Tools Development

The accompanying viewgraphs include an estimation the time requirements for the ECT given the current state of its development (i.e., the *ECT* and *SEM* tools). For a system the size of the EDRB, close to 1.5 technical staff years would be required. However, most of the effort is concentrated in three major tasks: generation of test cases, development of test environments, and resolution of don't cares and the creation of additional test cases.

An additional candidate for automation is the generation of semantic relations. Although not a particularly labor intensive task, it requires a detailed understanding of the semantics of the module and is prone to errors. Errors in the statement of semantic relations can in turn invalidate the rest of the ECT effort. Therefore, the most promising areas four tools development are:

1. *Semantic Analyzers:* The *SEM* program requires a semantic data input file which is manually created. Experience in creating such files has shown that the process is subject to analyst error. A semantic analyzer can generate such files automatically.

2. *Test Case Generator*: Test case generation requires identifying those input and externally set variables that satisfy (or do not satisfy) each condition and the values that should be assigned to these variables in order to execute the path specified by each rule. This identification is a matter of tracing how variables are set and used and may be amenable to automation using an existing static analyzer. This tool would find each condition in the code, trace variables associated with these conditions to their inputs, determine what ranges input values should be used to satisfy the specifications imposed by the rule, prepare a test case input file, and print the test case.

3. *Debugging Script Generator*: The work needed to generate a test environment can be largely reduced if batch oriented debugging tools such as *dbx* are used. The importance of the batch orientation is that test cases can be written off-line and input as files to the debugger, and debugger output can likewise be examined either manually or automatically.

4. *Support for Generation of Additional Test Cases*: Resolution of *Don't Cares* and the generation of additional test cases requires understanding of system-level concerns and association of these concerns with variables and control flow of the unit under test. By definition, the process can not be automated; otherwise the designed could be analyzed and the concerns would be resolved. However, providing cross references to variables and control flows which reduce the repetitive labor involved in performing this analysis, will result in a significant reduction of labor and in more thorough and uniformly high quality analyses.

6 Conclusions

The result of this work was that it was possible to analyze a large section of code which shares many characteristics in common with future real time distributed control systems that will be implemented in the next generation of aircraft and space vehicles. The results further showed that traceable test case specifications are generated, that unambiguous completion criteria can be established, and that automated tools can be successfully used.

Additional work is necessary to reduce time and resource requirements by the development of appropriate

tools. The benefits of such tools are exemplified by generation of the basic condition table which was previously an error prone and labor intensive task. With the *ECT* and *SEM* tools, this step has been reduced to a negligible portion of the total effort.

7 Acknowledgements

This paper is the result of work sponsored by the Naval Ocean Systems Center (NOSC) under contract N66001-90-C-7006 and by the NASA Langley Research Center (LaRC) under contract NAS1-18811. The authors wish to acknowledge the interest and support of Dr. Barry Siegel of NOSC and Mr. Carlos Liciega of LaRC, the technical monitors, as well as Ms. Susan Voigt of LaRC, who provided the motivation which resulted in this research.

References

KING69 P. King, "The interpretation of limited entry decision table format and relationships among conditions", Computing Journal, Vol 12, p. 320, November, 1969

GOOD75 J. Goodenough and S. Gerhart, "Toward a Theory of Test Data Selection", *IEEE Transactions on Software Engineering*, Vol. SE-1 No. 2, June, 1975, p. 156

MCCA87 J. M. McCall, et. al., *Methodology for Software Reliability Prediction*, Rome Air Development Center, RADC-TR-87-171, November, 1987

QUAN88 Quantum Software Inc., *QNX Operating System Reference Manual,* available from Quantum Software, Kanata, Ontario, Canada, 1988

TAI87 A. Tai, M. Hecht, and H. Hecht, "A New Method for the Verification of Fault Tolerant Software", *Proc. EASCON '87*, IEEE Catalog No. 87CH 2491-9, November, 1987, p. 53