Novel Test Case Design Techniques for Logical Specifications of Safety Critical Systems Software in Aerial Vehicle

Lakshmi KVNS
Advanced Systems Laboratory
DRDO
Hyderabad, India
lakshmikvns@asl.drdo.in

Sanjeev Kumar
Advanced Systems Laboratory
DRDO
Hyderabad, India
sanjeev_kumar0305@yahoo.co.in

Abstract: **Logical specifications are an essential part and are of major focus to describe and control behavior in safety critical systems software like avionics, control and code decoder system software, which are prone to introduction of bugs. This becomes major challenging task during software verification and validation (V&V) activities, which is an important activity for safety critical systems. To detect the bugs in software implementation or specifications related to logical specifications, for software assurance, test cases are to be designed to assist in thorough analysis and testing. Since traditional test case design techniques are not sufficient and capable of analyzing and testing specific requirements of our safety critical systems software, we propose novel test case design techniques to assist in this process. The secondary aim of these test case design techniques is to investigate the ways of minimization of test effort in terms of test cases, without affecting the fault detection capability negatively.**

This approach eliminates the effort needed to run exhaustive test cases and could be used in verification and validation activity with appropriate effort. Two case studies regarding safety critical systems software are presented whose logical specifications are related to Boolean inputs. Test cases are designed using novel techniques and are targeted to identify specific type of bugs in unique contexts of these software. We evaluate the generated test cases through dynamic testing and manual program analysis for both the software modules.

Keywords
Logical specifications, safety critical system software, test case design, Boolean domain

I.INTRODUCTION

In safety critical embedded systems, due to increased complexity and widespread use of software, rigorous verification and validation activities have been mandated which are advanced in nature to address at specification, design, coding and testing phases. This is required for achieving assurance for such critical systems. Software testing phase is identified as one of the most important and critical phase in software development life cycle. This poses intellectually challenging part during designing of test cases especially for logical specifications. Test cases are usually designed based on specifications, structure of the program code i.e. black box and white box techniques and experience [1], [2]. Along with these methods, test case design is also partly intuitive but largely systematic for generation of comprehensive test suite. Good testing involves much more than just running the program a few times to see whether it works or not [1].

In practice, taking into consideration of logical specifications, the possible number of test cases are very large due to permutations and combinations of related Boolean inputs as well as their sequence. This will result in significant amount of time spent in verification and validation effort to achieve goals that are targeted by test plan, which is bottle neck for the completion within due time of verification and validation activities. This demands efficient test case generation to find the bugs effectively.

We identified the need for research on testing techniques available for logical specifications/Boolean inputs. Many testing techniques are available but they differ in the type of test inputs. Under specification based test technique, [4] specifies the Combinatorial testing like All Combinations which results in very large number of test cases for our Boolean inputs in case study-I and don't cover sequences requirement in case study-II. Several authors have proposed approaches for designing test cases from Boolean expressions and specifications. Pair wise testing is used in [5] considering the combinations of input variable values, but it is used for testing only logical specifications in expressions form in software. [6] presents the strategies for automatically generating test data for Boolean formula. [7] focuses on faults in Boolean expressions for testing. Test cases are automatically generated from logical

specifications which are based on courteous logic representations and not related to Boolean inputs in [8]. After due research, it has been identified that these test case design techniques are not sufficient and capable of analyzing and testing specific logical requirements which are related to Boolean input.

In this paper, we propose the novel techniques for designing test cases for any implementation intended to satisfy given logical requirements which are related to Boolean domain. This results in selection of most appropriate and proper set of test cases, for which various rules are build. The contribution of this paper is to demonstrate the approach of novel test case design techniques for two realistic case studies in safety critical embedded systems and bugs revealed in these cases.

This paper is organized as follows

Section II presents the brief overview of the approach being followed in defining the framework to design the test cases. Section III covers case study-I describing our process of designing, generation and execution of test cases for Code decoder system software. Case study-II covering our methodology for designing the test cases and usage in program analysis for Navigation system software is discussed in Section IV. Section V presents the results of these techniques. Section VI summarizes and concludes the paper. Section VII describes the ongoing and future work.

II. APPROACH

We defined and developed a unified framework for the test case design and its application as shown in Fig. 1. This comprises of sequence of steps for designing and applying the test cases to identify the bugs. This fits within the V&V activities and is given below.

1) Identify the test inputs and study their behavior.
2) Identify the logical specifications. Analyze the operation modes i.e. impact on system behavior with respect to these test inputs and logical specifications.
3) Depending on the test inputs and logical specifications, define the test case design criteria to build a series of rules to generate test cases solely based on the application specifications/code implementation. Novel techniques are being used in building the rules for designing test cases.
4) The designed test cases are demonstrated in appropriate mechanisms of testing or analysis to uncover defects in software implementation.

This approach is unique in both the case studies. Each step is decomposed progressively in greater level of detail for both the case studies in the next subsections.

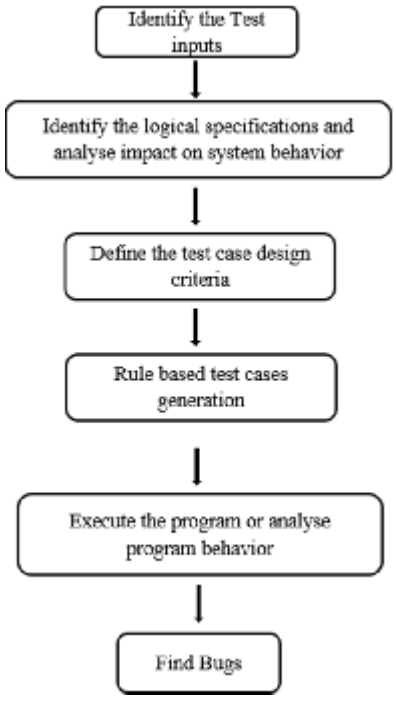

Figure 1. Unified framework for test case generation.

III. CASE STUDY-I

We present a case study of Code decoder system software. We identified code decoding module as module under test. Four number of unique hexadecimal codes are identified as test inputs for this module. These are the parameters that drives the behavior of the module under test. Based on the related logical software specifications and behavior, test case design criteria are defined resulting in test cases. The designed test cases are executed in target based test setup. The results show that this technique can effectively find the bugs in software implementation and substantially reduce the number of test cases generated. Schematic for this procedure is shown in Fig. 2.

A. Test Inputs

As per system specification, the system handles four number of unique hexadecimal codes of 3 digits each which are converted to binary numbers of 12 bits each. These are considered as test input codes.

B. Logical Specifications

We identified following logical software specifications for validation of each of the test input code. These are identified to be tested using test cases.

- If there is single error bit in test input code, test input code is considered as valid one, decoded and acted upon it
- If there are double error bits in test input code, test input code is considered as valid one and decoded and acted upon it
- If there are triple error bits in test input code, test input code is rejected and the corresponding failure handling mechanism is to be invoked.

C. Structural Requirement

As per software code implementation, each hexadecimal code is checked for number of error bits in byte wise manner.

Operational modes of the system under these logical specifications and structural requirements is not presented in this paper.

D. Test Case Design Criteria

Exhaustive testing with all possible combinations of 12 bits of binary numbers is typically prohibitively impractical, because this would require 4096 (2^{12}) distinct test cases. Therefore our goal is to define a mechanism to identify the test cases that would substantially be smaller than exhaustive test sets, but nonetheless be highly effective at detecting bugs.

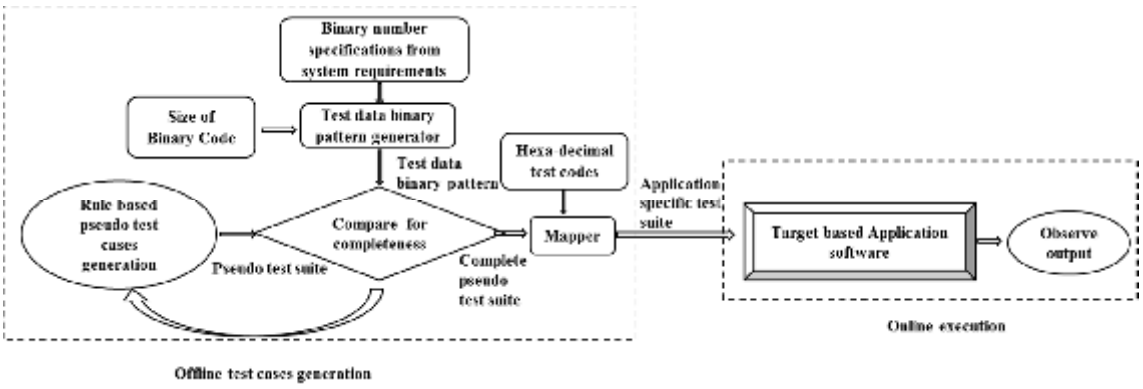

Figure 2. Schematic procedure for test case generation and execution.

In defining the test case design criteria, unified approach of combination of both the specification based black box testing technique with implementation based white box testing techniques are taken into consideration i.e. gray box testing [1]. We applied them in synergy to benefit from their strengths.

Based on the above mentioned logical software requirements i.e. specifications and code implementation i.e. structure, we identified number of error bits and their position in the test input codes as test case design criteria. This is used as basis for test case generation by following a specific rule based strategy as presented in next sub section.

E. Test Case Design

Taking into consideration of test case design criteria, we design the test cases using novel technique of building the rules. As a first step, we build the rules for generation of pseudo test cases.

1) Rule based Pseudo Test Suite (Test cases) Generation

The notation for all the codes is defined to facilitate the discussion. Let F represents pseudo test input code (binary number) corresponding to hexadecimal test input code Z=0xpqr that is being tested. We refer to least significant bit in the binary number as f0, second bit as f1, third bit as f2 and so on. We use '1' and '0' to represent the 'error bit' and 'no error bit' conditions respectively in the pseudo test input code.

Rules are built to incorporate the error bits in pseudo test input code in a unique pattern. Thereby, pseudo test cases are generated.

For example, consider the pseudo test input code modified with f0=1 and rest all bits as 0. This generates a pseudo test case having single bit error. Another code modified with f4=1, f7=1 and rest all bits as 0. This generates another pseudo test case as having double bit errors.

We generated complete pseudo test suite comprising of three test sets including pseudo test cases related to single error bit, double error bits and triple error bits test sets using unique series of rules based strategy as described below. Each test set comprises of test cases related to each of the logical specifications.

Following list provides the generalized rules build to generate test cases

- For single error bit case, one has to introduce the error bit from least significant bit as the starting point and propagate till the most significant bit.
- For the case of double error bits, as a starting point, introduce first error bit in least significant bit and second error bit in next adjacent bit and is propagated till most significant bit. This is repeated with the introduction of first error bit in all bit positions.
- For the case of triple error bits, as a starting point, introduce first error bit in least significant bit and other two error bits in next adjacent bit positions and is propagated till two adjacent most significant bit positions. This is repeated with the introduction of first error bit in all bit positions. This whole set is repeated again by increasing the distance between second and third error bits from 1 to 9.

Details of each of the rule build is presented herein for clarity and understanding.

Test Set I: Single error bits

Rule 1: Starting from least significant bit (f0), introduce error ('1') in single bit. Subsequently, this is repeated from f1 to f11, each at a time.

This rule generates 12 number of pseudo test cases related to single error bits comprised as Test set I. Fig. 3 shows this sequence of steps for generating single error bits and their corresponding pseudo test cases.

Test Set II: Double error bits

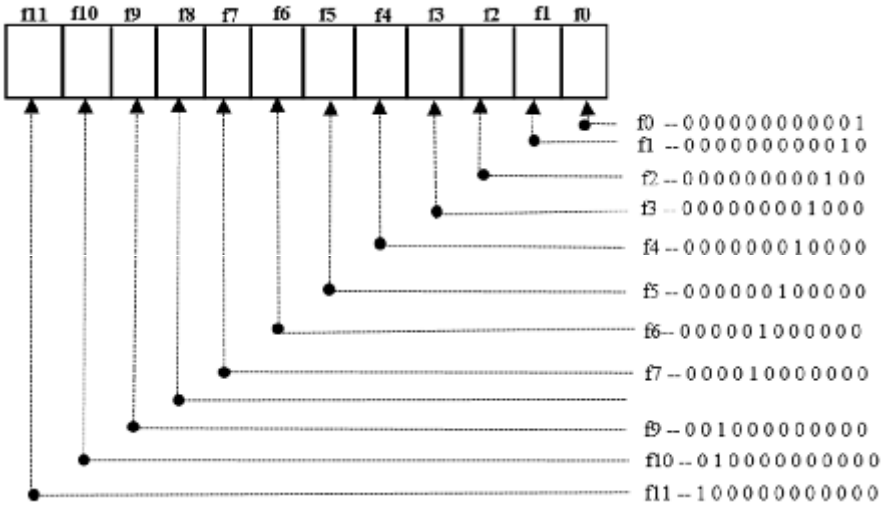

Figure 3.Pseudo test cases for single error bit (Rule 1).

Rule 2.a: With the introduction of error (‘1’) in least significant bit f0, introduce one more error (‘1’) in another single bit, starting from bit f1. Subsequently, this is repeated in all bits from f2 to f11, each at a time.
This rule generates 11 number of pseudo test cases related to double error bits. Fig. 4 shows the step wise introduction of error bits at different positions and the corresponding pseudo test cases.

Rule 2.b: With the introduction of error (‘1’) in bit f1, introduce one more error (‘1’) in another single bit starting from bit f2. Subsequently, this is repeated in all bits from f3 to f11, each at a time.
This rule generates 10 number of pseudo test cases related to double error bits.

Rule 2.c: With the introduction of error (‘1’) in bit f2, introduce one more error (‘1’) in another single bit starting from bit f3. Subsequently, this is repeated in all bits from f4 to f11, each at a time.
This rule generates 9 number of pseudo test cases related to double error bits.

Similar rules are build till the introduction of error in bit f10 (Rule 2.j). These rules from 2.a to 2.j generates 66 number of pseudo test cases related to double error bits comprised as Test set II.

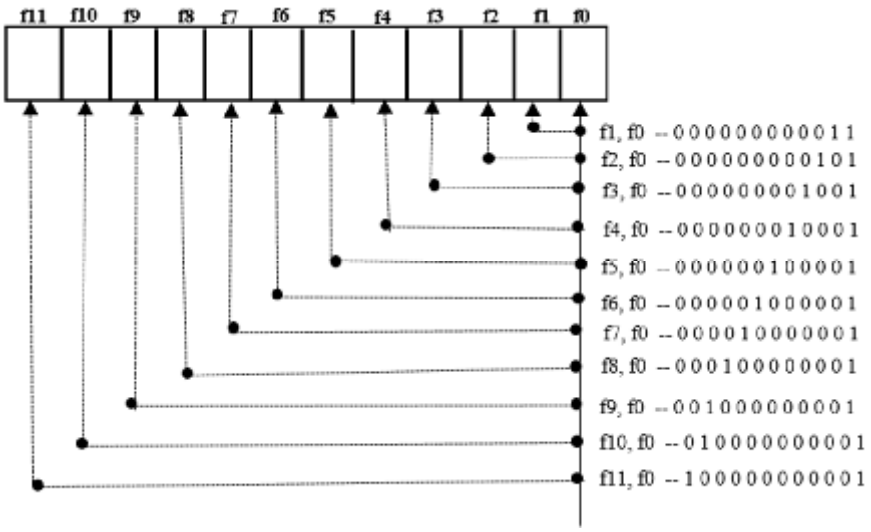

Figure 4. Pseudo test cases for double error bits (Rule 2.a).

Test Set III: Triple error bits

Test subset III. A:

Rule 3.1.a: With the introduction of error (‘1’) in least significant bit f0, introduce two more errors (‘1’) in two adjacent bits, starting from (f1, f2). Subsequently, this is repeated in all bits from (f2, f3), (f3, f4) to (f10, f11), each pair at a time.
This rule generates 10 number of pseudo test cases related to triple error bits. Fig. 5 shows the step wise introduction of error bits at two adjacent positions and the corresponding pseudo test cases.

Rule 3.1.b: With the introduction of error (‘1’) in bit f1, introduce two more errors (‘1’) in two adjacent bits, starting from (f2, f3). Subsequently, this is repeated in all bits from (f3, f4) to (f10, f11), each pair at a time.
This rule generates 9 number of pseudo test cases related to triple error bits.

Rule 3.1.c: With the introduction of error (‘1’) in bit f2, introduce two more errors (‘1’) in two adjacent bits, starting from (f3, f4). Subsequently, this is repeated in all bits from (f4, f 5) to (f10, f11) each pair at a time.
This rule generates 8 number of pseudo test cases related to triple error bits.

Similar rules are build till the introduction of error in bit f9 (Rule 3.3.j). These rules from 3.3.a to 3.3.j generates 55 number of pseudo test cases related to triple error bits comprised as Test subset III.A.

Test subset III. B:

Rule 3.2.a: With the introduction of error (‘1’) in least significant bit f0, introduce two more errors (‘1’) in two bits which are 1 bit distant away, starting from (f1, f3). Subsequently, this is repeated in all bits from (f2, f4), (f3, f 5) to (f9, f11), each pair at a time.
This rule generates 9 number of pseudo test cases related to triple error bits.

Rule 3.2.b: With the introduction of error (‘1’) in bit f1, introduce two more errors (‘1’) in two bits which are 1 bit distant away, starting from (f2, f4). Subsequently, this is repeated in all bits from (f3, f 5), (f4, f6) to (f9, f11), each pair at a time.
This rule generates 8 number of pseudo test cases related to triple error bits.

Rule 3.2.c: With the introduction of error (‘1’) in bit f2, introduce two more errors (‘1’) in two bits which are 1 bit distant away, starting from (f3, f 5). Subsequently, this is repeated in all bits from (f4, f6), (f 5, f7) to (f9, f11), each pair at a time.
This rule generates 7 number of pseudo test cases related to triple error bits.

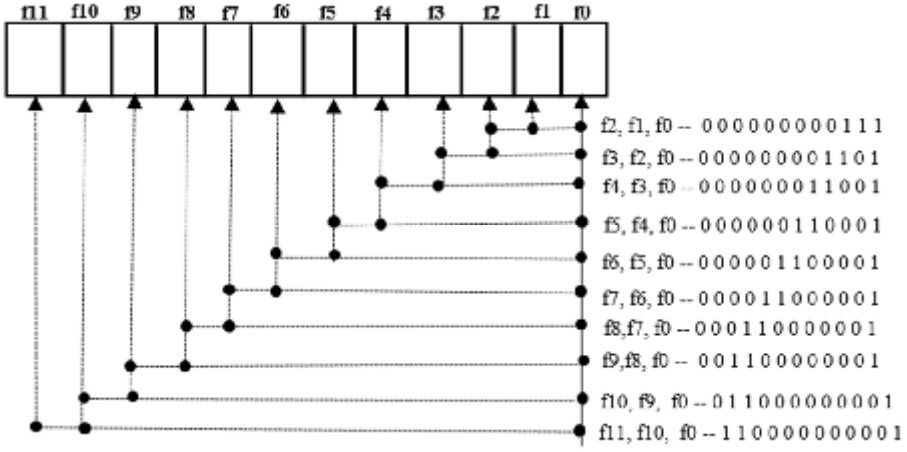

Figure 5. Pseudo test cases for triple error bits (Rule 3.1.a).

Similar rules are build till the introduction of error in bit f8 (Rule 3.2.i). These rules from 3.2.a to 3.2.i generates 45 number of pseudo test cases related to triple error bits comprised as Test subset III.B.

Test subset III. C:

Rule 3.3.a: With the introduction of error (‘1’) in least significant bit f0, introduce two more errors (‘1’) in two bits which are 2 bits distant away, starting from (f1, f4). Subsequently, this is repeated in all bits from (f2, f 5), (f3, f6) to (f8, f11), each pair at a time.
This rule generates 8 number of pseudo test cases related to triple error bits.

Rule 3.3.b: With the introduction of error (‘1’) in bit f1, introduce two more errors (‘1’) in two bits which are 2 bits distant away, starting from (f2, f 5). Subsequently, this is repeated in all bits from (f3, f6), (f4, f7) to (f8, f11), each pair at a time.
This rule generates 7 number of pseudo test cases related to triple error bits.

Rule 3.3.c: With the introduction of error (‘1’) in bit f2, introduce two more errors (‘1’) in two bits which are 2 bits distant away, starting from (f3, f6). Subsequently, this is repeated in all bits from (f4, f7), (f 5, f8) to (f8, f11), each pair at a time.
This rule generates 6 number of pseudo test cases related to triple error bits.

Similar rules are build till the introduction of error in bit f7 (Rule 3.3.h). These rules from 3.3.a to 3.3.h generates 36 number of pseudo test cases related to triple error bits comprised as Test subset III.C.

Test subset III. D:

Rule 3.4.a: With the introduction of error (‘1’) in least significant bit f0, introduce two more errors (‘1’) in two bits which are 3 bits distant away, starting from (f1, f 5). Subsequently, this is repeated in all bits from (f2, f6), (f3, f7) to (f7, f11), each pair at a time.
This rule generates 7 number of pseudo test cases related to triple error bits.

Rule 3.4.b: With the introduction of error (‘1’) in bit f1, introduce two more errors (‘1’) in two bits which are 3 bits distant away, starting from (f2, f 6). Subsequently, this is repeated in all bits from (f3, f7), (f4, f8) to (f7, f11), each pair at a time.
This rule generates 6 number of pseudo test cases related to triple error bits.

Rule 3.4.c: With the introduction of error (‘1’) in bit f2, introduce two more errors (‘1’) in two bits which are 3 bits distant away, starting from (f3, f7). Subsequently, this is repeated in all bits from (f4, f8), (f 5, f9) to (f7, f11), each pair at a time.
This rule generates 5 number of pseudo test cases related to triple error bits.

Similar rules are build till the introduction of error in bit f6 (Rule 3.4.g). These rules from 3.4.a to 3.4.g generates 28 number of pseudo test cases related to triple error bits comprised as Test subset III.D.

Test subset III. E:

Rule 3.5.a: With the introduction of error (‘1’) in least significant bit f0, introduce two more errors (‘1’) in two bits which are 4 bits distant away, starting from (f1, f6). Subsequently, this is repeated in all bits from (f2, f7), (f3, f8) to (f6, f11), each pair at a time.
This rule generates 6 number of pseudo test cases related to triple error bits.

Rule 3.5.b: With the introduction of error (‘1’) in bit f1, introduce two more errors (‘1’) in two bits which are 4 bits distant away, starting from (f2, f7). Subsequently, this is repeated in all bits from (f3, f8), (f4, f9) to (f6, f11), each pair at a time.
This rule generates 5 number of pseudo test cases related to triple error bits.

Rule 3.5.c: With the introduction of error (‘1’) in bit f2, introduce two more errors (‘1’) in two bits which are 4 bits distant away, starting from (f3, f8). Subsequently, this is repeated in all bits from (f4, f9), (f 5, f10) to (f6, f11), each pair at a time.
This rule generates 4 number of pseudo test cases related to triple error bits.

Similar rules are build till the introduction of error in bit f 5 (Rule 3.5.f). These rules from 3.5.a to 3.5.f generates 21 number of pseudo test cases related to triple error bits comprised as Test subset III.E.

Test subset III. F:

Rule 3.6.a: With the introduction of error (‘1’) in least significant bit f0, introduce two more errors (‘1’) in two bits which are 5 bits distant away, starting from (f1, f7). Subsequently, this is repeated in all bits from (f2, f8), (f3, f9) to (f 5, f11), each pair at a time.
This rule generates 5 number of pseudo test cases related to triple error bits.

Rule 3.6.b: With the introduction of error (‘1’) in bit f1, introduce two more errors (‘1’) in two bits which are 5 bits distant away, starting from (f2, f8). Subsequently, this is repeated in all bits from (f3, f9), (f4, f10) to (f5, f11), each pair at a time.
This rule generates 4 number of pseudo test cases related to triple error bits.

Rule 3.6.c: With the introduction of error (‘1’) in bit f2, introduce two more errors (‘1’) in two bits which are 5 bits distant away, starting from (f3, f9). Subsequently, this is repeated in all bits from (f4, f10) to (f 5, f11), each pair at a time.
This rule generates 3 number of pseudo test cases related to triple error bits.

Similar rules are build till the introduction of error in bit f4 (Rule 3.6.e). These rules from 3.6.a to 3.6.e generates 15 number of pseudo test cases related to triple error bits comprised as Test subset III.F.

Test subset III. G:

Rule 3.7.a: With the introduction of error (‘1’) in least significant bit f0, introduce two more errors (‘1’) in two bits which are 6 bits distant away, starting from (f1, f8). Subsequently, this is repeated in all bits from (f2, f9), (f3, f10) to (f4, f11), each pair at a time.
This rule generates 4 number of pseudo test cases related to triple error bits.

Rule 3.7.b: With the introduction of error (‘1’) in bit f1, introduce two more errors (‘1’) in two bits which are 6 bits distant away, starting from (f2, f9). Subsequently, this is repeated in all bits from (f3, f10) to (f4, f11) each pair at a time.
This rule generates 3 number of pseudo test cases related to triple error bits.

Rule 3.7.c: With the introduction of error (‘1’) in bit f2, introduce two more errors (‘1’) in two bits which are 6 bits distant away, starting from (f3, f10). Subsequently, this is repeated for (f4, f11) pair of bits.
This rule generates 2 number of pseudo test cases related to triple error bits.

Similar rule is build till the introduction of error in bit f3 (Rule 3.7.d). These rules from 3.7.a to 3.7.d generates 10 number of pseudo test cases related to triple error bits comprised as Test subset III.G.

Test subset III. H:

Rule 3.8.a: With the introduction of error (‘1’) in least significant bit f0, introduce two more errors (‘1’) in two bits which are 7 bits distant away, starting from (f1, f9). Subsequently, this is repeated in all bits from (f2, f10) to (f3, f11) each pair at a time.
This rule generates 3 number of pseudo test cases related to triple error bits.

Rule 3.8.b: With the introduction of error (‘1’) in bit f1, introduce two more errors (‘1’) in two bits which are 7 bits distant away, starting from (f2, f10). Subsequently, this is repeated for (f3, f11) pair of bits.

This rule generates 2 number of pseudo test cases related to triple error bits.

Rule 3.8.c: With the introduction of error (‘1’) in bit f2, introduce two more errors (‘1’) in two bits which are 7 bits distant away i.e. (f3, f11).
This rule generates 1 pseudo test case related to triple error bits.

These rules from 3.8.a to 3.8.c generates 6 number of pseudo test cases related to triple error bits comprised as Test subset III.H.

Test subset III. I:

Rule 3.9.a: With the introduction of error (‘1’) in least significant bit f0, introduce two more errors (‘1’) in two bits which are 8 bits distant away, starting from (f1, f10). Subsequently, this is repeated in (f2, f11) pair of bits.
This rule generates 2 number of pseudo test cases related to triple error bits.

Rule 3.9.b: With the introduction of error (‘1’) in bit f1, introduce two more errors (‘1’) in two bits which are 8 bits distant away i.e. (f2, f11) pair of bits.
This rule generates 1 pseudo test case related to triple error bits.

These rules from 3.9.a to 3.9.b generates 3 number of pseudo test cases related to triple error bits comprised as Test subset III.I.

Test subset III. J:

Rule 3.10.a: With the introduction of error (‘1’) in bit f0, introduce two more errors (‘1’) in two bits which are 9 bits distant away i.e. (f1, f11).
This rule i.e. 3.10.a generates 1 pseudo test case related to triple error bits comprised as Test subset III.J.

These rules comprising of Test subset III.A to III.J results in generation of 220 pseudo test cases for triple error bits.

2) Test Data Binary Pattern Generator

Once this is accomplished, in this case study, we developed a tool utility named Test data binary pattern generator which generates all possible combinations of Boolean inputs in sequential order. This takes 12 as the size of binary pattern along with the software/system requirements as given below.

- Binary numbers with single 1s.
- Binary numbers with 2 no. of 1’s.
- Binary numbers with 3 no. of 1’s.

Few of the typical patterns generated by Test data binary pattern generator is presented in Table I.

TABLE I
TYPICAL TEST DATA BINARY PATTERN

S. No	Test Data Binary Pattern
1.	000000000000
2.	000000000001
3.	000000000010
4.	000000000011
5.	000000000100
6.	000000000101
7.	000000000110
8.	000000000111
9.	000000001000
10.	000000001001

3) Comparison of Pseudo Test Suite with Test Data Binary Pattern

To check the completeness of our generated pseudo test suite, we compared our pseudo test cases with the test data binary pattern. And rules under subheading 1) in this subsection are reframed till completeness is ensured and adequacy is verified.

4) Generation of Application specific Test Suite

Once the completeness of pseudo test suite is accomplished, pseudo test cases bit pattern is to be mapped or replicated in each of the application specific hexadecimal test input code which satisfy logical specifications.

The matrix covering Rule IDs, Rule Subset IDs, and Test Set IDs is summarized in Table II. This table also shows the relation of test cases to logical specifications and code implementation. This test suite comprises of 298 test cases for each of the hexadecimal code.

For better understanding, consider an example in which pseudo test case F= 0110 1000 0000 (three error bits condition-f7=1, f9=1, f10=1 and rest all bits are 0) and hexadecimal test input code Z=0x456=0100 0101 0110. On applying the pseudo test case bit pattern to Z, i.e. introducing error bits at z7, z9 and z10 positions, final test case for Z results as $Z_{\text{test case}}$=0010 1101 0110=0x2C6.

For each hexadecimal code, this resulted in test suite comprising of a total of 12 single error bit, 66 double error bit and 220 triple error bit test cases. This resulted in test suite of 298 test cases for each of the application specific hexa-decimal code. For four number of application specific hexadecimal codes, this resulted in 1192 test cases to be executed in Code decoder system (target) based dynamic test up for this code decoding module which is module under test.

F. Dynamic Testing

This whole framework for generating and executing test cases is semi-automated as shown in Fig. 2, where in

- Test case generation is done manually
- Verified for completeness using automatic generated test data binary patterns
- Manual test case execution in target based test setup

TABLE II
SIZE OF TEST SETS

S. No	Test Set ID		Rule ID's	No. of test cases for each binary number (hexa decimal code)	No. of test cases for four binary numbers (hexa decimal codes)
1.	Test Set I		1	12	48
2.	Test Set II		2.a, 2.b, 2.c, 2.d	66	264
3.	Test Set III	Test Subset III.A	3.1.a, 3.1.b, 3.1.c, 3.1.d, 3.1.e, 3.1.f, 3.1.g, 3.1.h, 3.1.i, 3.1.j	55	220
		Test Subset III.B	3.2.a, 3.2.b, 3.2.c, 3.2.d, 3.2.e, 3.2.f, 3.2.g, 3.2.h, 3.2.i	45	180
		Test Subset III.C	3.3.a, 3.3.b, 3.3.c, 3.3.d, 3.3.e, 3.3.f, 3.3.g, 3.3.h	36	144
		Test Subset III.D	3.4.a, 3.4.b, 3.4.c, 3.4.d, 3.4.e, 3.4.f, 3.4.g	28	112
		Test Subset III.E	3.5.a, 3.5.b, 3.5.c, 3.5.d, 3.5.e, 3.5.f	21	84
		Test Subset III.F	3.6.a, 3.6.b, 3.6.c, 3.6.d, 3.6.e	15	60
		Test Subset III.G	3.7.a, 3.7.b, 3.7.c, 3.7.d	10	40
		Test Subset III.H	3.8.a, 3.8.b, 3.8.c	6	24
		Test Subset III.I	3.9.a, 3.9.b	3	12
		Test Subset III.J	3.10	1	4
Total No. of Test Cases				298	1192

We evaluated and assessed the effectiveness of the test suite generated in this manner by carrying out dynamic testing in target based test setup. This is done by executing the program under test in target based test setup and monitoring the results by comparing actual results with expected results.

G. Outcome

This technique resulted in failure of 160 test cases out of 1192 designed test cases. On carrying out root cause analysis, it revealed a total of 2 previously unknown bugs which are critical in nature, severe enough to reject a prospective release of Code decoder system software version. Details of these bugs are not presented in this paper.

This unified and novel technique approach is very effective for testing the program whose logical specifications are related to Boolean domain due to the underlying key strengths

- It constructs test suites systematically so as to enable easier identification of failure patterns which resulted in failure of test cases during dynamic testing. Henceforth, this assisted in easier root cause analysis for the failure cases. This is possible due to the fact that test case design criteria is in line with logical and structural requirements.
- Its effectiveness at reducing test suite data combinations. In this case study, for each of the hexa-decimal test input code, test suite comprises of test cases which are very less (298) as shown in Table II. compared to actual theoretical combinations ($2^{12} = 4096$) as per combinations of Boolean input.

IV. CASE STUDY-II

Second case study is dealt with Navigation System software. We identified module related to data acquisition from a package as Module under Test. In this module, Packet and data validity failures are designated as failure modes. Related flags in the software code are identified as test inputs. These are considered as Boolean inputs which impacts the operational flow of the software module. Test case design criteria is identified depending on the logical specifications of these test inputs. Even though the intent of test case generation is to cover different combinations of Boolean inputs in specific sequence, the real aim is to detect software bugs in software implementation to see how well specifications are implemented. Test cases are designed that consists of all combinations of pairs of input values but with specific sequencing. These specific sequence of test cases are used in manual static analysis of module. The effectiveness of the designed test cases is shown by the capability to reveal the bugs in the software. These bugs are related to the states of flags and status word bits that are set during specific sequence of execution which is not visible to external world.

A. Test Inputs

As per module requirement, the failure modes i.e. packet and data validity flags are identified as test inputs.

B. Logical specifications

We identified following logical specifications related to validation of these test inputs.

- If any of the failure modes are detected, previous state for package data is to be retained. Failure is to be indicated in corresponding status word bits and failure flags.
- If no failure is identified, state for data is to be updated.
- Failure is to be handled by fault tolerance, if failure modes persists for n consecutive cycles of execution.

Complete operational modes of the system with respect to these logical specifications is not presented in this paper.

C. Test Case Design Criteria

To test module with two Boolean inputs, four test cases are sufficient for testing all possible combinations. But these are not sufficient enough to cover all logical specifications.

We identified transitions of failure modes across multiple cycles of execution of this module as the basis for test case design criteria. Corresponding logical specifications and their effect on system behavior are taken into consideration while defining this test case design criteria. We identified that the state for this module comprises of data variables, flags and status word bits.

D. Test Case Design

Logical specifications covers only possible values for both the test inputs i.e. ‘1’, ‘0’ as Boolean variables. They do not address their sequence or order of their combinations i.e. 00—11—01—01—11—01—10.

To design the test cases, decision tables [2] can be used. Decision tables addresses the combination of inputs but do not take into account of maintaining previous state. State transition technique [1] covers transitions of states of system but not input transitions [1]. In this context, we propose to design the test cases using novel technique of building rules to generate test cases from all possible combinations of test inputs and their transitions while meeting logical requirements. This technique unifies tree structure and transition concepts.

The notation used for these concepts is discussed. Let 1, 0 and ‘X’ represent the ‘failure’, ‘no failure’, and ‘don’t care’ (i.e. a value that can be either 0 or 1) conditions for these test inputs. The possible combinations for these test inputs is 00, 01, 10 and 11. They represent the combination of present failure modes and control the system behavior accordingly.

Tree structure used in this representation of failure transitions of test inputs as shown in Fig. 6. In this, parent node represents the present failure mode combination and child nodes represents the next possible combination. Transitions from each node to other node is represented by the edge.

Rules are built to incorporate all possible failure transitions of these combinations. The strategy followed is presented as given below.

Rule 1:

Starting from parent node ‘00’, traverse through all the next possible paths in the tree structure.

This generates 4 test case sequences for the next failure mode for ‘00’ failure mode combination i.e. 00-00, 00-01, 00-10 and 00-11.

Rule 2:

Starting from next parent node ‘01’, traverse through all the next possible paths in the tree structure.

This generates 4 test case sequences for the next failure mode for ‘01’ failure mode combination.

This process is to be repeated for all the parent nodes. All paths should be covered to make sure that all transitions in this application are covered. Our adequacy criteria is defined to select all possible transitions from the tree structure and coverage with respect to multiple cycle execution. This adequacy criteria is to assess the comprehensiveness of a given test suite.

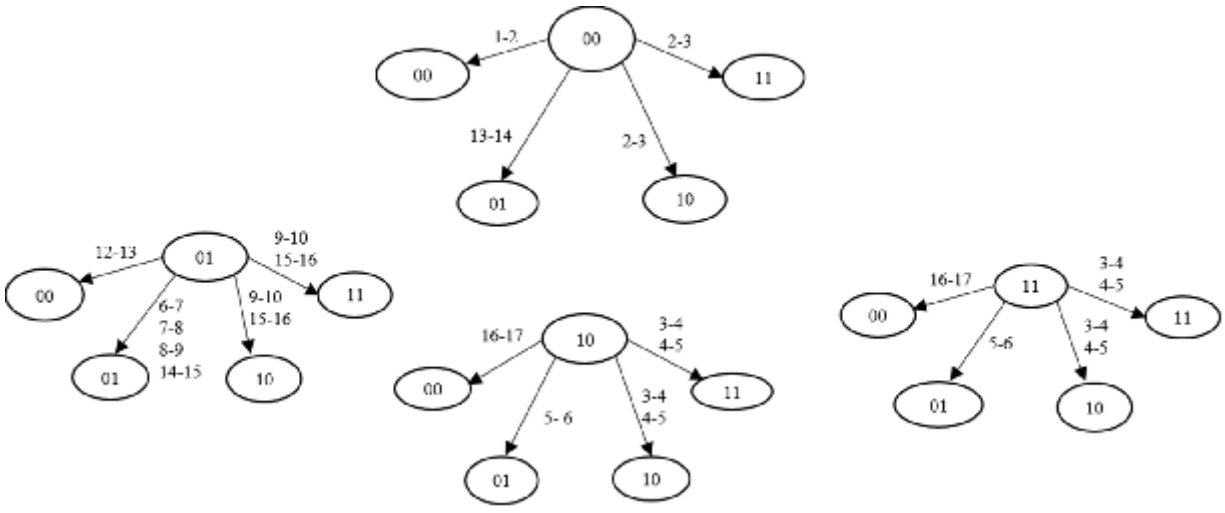

Figure 6. Tree structure with test case sequences.

In the above technique, following points are considered

- Data is to be retained for one cycle if failure is defined in present combination.
- For each parent node, history of the all transitions is not required to be maintained. Henceforth, we restrict the tree structure to only one level. Otherwise the tree structure would have been exploded to multiple levels.

These considerations has reduced drastically the size of test cases that are generated.

The generated test suite comprising of specific sequence of test cases is presented in Table III. The mapping of test case sequences with respect to tree structure transitions are represented on the edges of the tree structure as shown in Fig.6. For better understanding, consider an example in which transition from '00' parent node to '00' child node is covered by test case sequence 1 to 2. Transitions from each test case is depicted on the edge between nodes.

Relationship of test cases to logical specifications is not mentioned in this paper.

E. Manual Static Analysis

The need of carrying out static analysis manually i.e. code walkthrough instead of dynamic testing with the designed test cases sequences arises due to constraints given as below.

- Lack of feasibility of simulating designed sequence of test inputs in target based test setup.
- Code instrumentation based testing is not supportive in the target based test set up

Hence manual static analysis i.e. code walkthrough of module under test with respect to designed test cases sequence has been carried out. Each test case is mentally executed. Test data are walked through the logic of the code. The state of the program i.e. the values of variables is monitored on data variables, flags and status word bits settings. This process is considered manually tedious. Indeed, it is quite challenging to carryout analysis manually for this preliminary module which is consisting of only two test inputs. However, this approach helped to analyze the state of the program under multiple sequence of test data even though the visibility of the program is limited by above mentioned constraints.

F. Outcome

This technique revealed 2 previously unknown bugs. These reflect incorrect states of flags and status word bits that are set during specific sequence of execution which is not visible to external world. Details of these bugs are not presented in this paper.

In an aerial vehicle, central computer is the decision maker for this Navigation System. These status bits and flags reflect the present state of the system and thereby affecting the behavior of central computer.

TABLE III
TEST SUITE: CASE STUDY-II

S.No	Packet Failure	Data Failure
1.	0	0
2.	0	0
3.	1	X
4.	1	X
5.	1	X
6.	0	1
7.	0	1
8.	0	1
9.	0	1
10.	1	X
11.	0	1
12.	0	1
13.	0	0
14.	0	1
15.	0	1
16.	1	X
17.	0	0

The key strength of this novel test case design technique is that it addresses the temporal logic requirements i.e. failure modes transitions across multiple cycles. This technique is not covered by prevalent testing techniques [1], [2] and [3], which are highly essential to any embedded systems software.

V. RESULTS

The effectiveness of the test cases generated is found by main aim to reveal bugs in implementation with respect to logical specifications which are diversified nature for both the cases. Bugs which are mission critical in nature are revealed in first case study after carrying out target based dynamic testing. In second case study, specifications which cannot be tested using traditional or code instrumentation based testing are analyzed using the designed test cases, thereby uncovering the bugs which go unnoticed in system behavior.

VI. CONCLUSION

Novel techniques for test case design are proposed with two contrasting goals, reducing the number of test cases while maximizing the fault detection capability. The fault detection effectiveness of these techniques is investigated through dynamic testing and analysis. The approach presented in this paper is shown to design test cases that are specific to application logical requirements whose inputs are related to Boolean domain and reveal the bugs in implementation/requirements. In contrast to normal binary combinations and text book techniques, our techniques are explicitly designed to be used to generate test pattern to test any implementation of a given Boolean inputs and their related logical specification. This approach was found to be especially advantageous when the number of Boolean variables are high and dependency on these variables is complex. The designed test suite is focused and minimal and not exhaustive, henceforth minimizing the effort, time and cost of given verification and validation activity. This is done manually and is tedious in terms of analysis.

VII. ONGOING AND FUTURE WORK

In current work, manually deriving the test cases and their sequences is labor intensive process. Henceforth, preliminary version of frame work using formal verification methods to generate the test case sequences using the proposed test case design techniques is developed for Case study-II. This methodology enables to apply this approach to verification and validation of software applications, which are high in complexity and number of logical requirements with dramatic time and cost saving.

Directions for future work include the following.

To generate test cases sequence for complex modules wherein logical specifications and test inputs are huge in number and dependencies, analysis for data variables, flags and status words bits becomes cumbersome. This motivated us to carry forward these techniques using full-fledged version of automated framework implemented using formal verification methods. This will enable us to save great amount of time and cost.

ACKNOWLEDGMENT

The authors would like to thank Deblina Das, Prasuna S and Ansuman Banerjee for developing preliminary version of tool for generation of test case sequence as designed in Case study II in this paper and verification of corresponding specifications in code implementation using Formal Methods.

REFERENCES

[1] Lee Copeland, A Practitioner's Guide to Software Test Design.
[2] Glenford J. Myers, The Art of Software Testing, II Edition.
[3] C Kaner, J Falk, H Nguyen , Testing Computer Software.
[4] ISO/IEC/IEEE 29119-4: Test Techniques,2013.
[5] William Alton Ballance, Sergiy Vilkomir and William Jenkins, "Effectiveness of Pair-wise Testing for Software with Boolean inputs", 2012 IEEE Conference on Software Testing, Verification and Validation, doi: 10.1109/ICST.2012.144.
[6] E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a Boolean specification," IEEE Trans. Software Engineering, vol. 20, no. 5, May 1994, pp. 353-363, doi: 10.1109/32.286420.
[7] Usha Badhera, Purohit G.N and S.Taruna, "Boolean Specification Based Testing Techniques:A Survey", ITCS, SIP, JSE-2012, CS & IT 04, pp. 337-346, 2012, doi:10.5121/csit.2012.2131

[8] Richa Sarma and K.K. Biswas, "Automated Generation of Test Cases from Logical Specification of Software Requirements", 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE-2014), pp. 241-248, doi: 10.5220/0004972902410248