APPLICATIONS OF SPECIFICATION-BASED TESTING IN FLIGHT SOFTWARE DEVELOPMENT FOR HUBBLE SPACE TELESCOPE MISSION OPERATIONS

Nzinga Tull
December 13, 2005

ENSE 623: Systems Engineering Validations and Verification
Dr. Mark Austin, Professor

OUTLINE

I. Introduction and Purpose

Specification-based testing is performed on software and other types of systems to improve the quality of the system by verifying the intended functionality of the system. The goal of this paper was to explore the benefits and uses of specification-based testing, particularly in application to software development for Hubble Space Telescope (HST) Mission Operations. The advantages and limitations of different specification-based testing techniques are discussed. A case study will be performed to illustrate use the most applicable technique for HST Mission Operations.

II. Overview of HST Mission Operations

Since its April 1990 launch, the Hubble Space Telescope has been orbiting 600 kilometers above Earth every 96 minutes. The length of the primary structure of the spacecraft is 13.2 meters (43.5 ft.) and the maximum diameter is 4.2 meters (14 ft.) It is about the size of a large school bus and weighs over 10 tons. Since launch, engineers at National Aeronautics and Space Administration's (NASA's) Goddard Space Flight System (GSFC) have been responsible for the successful maintenance and operation of NASA's most prolific orbiting observatory. System engineers (SEs) are tasked with monitoring and trending spacecraft data to characterize performance, address anomalies as they occur, and design and implement configuration changes that would optimize performance and extend mission life. Distinct groups of SEs are responsible for all of the telescope's primary subsystems:

- Data Management Subsystem (DMS): Includes the flight computer and spacecraft health and safety data management and interface hardware).
- Electrical Power Subsystem (EPS): Include batteries, solar arrays, power control unit, and other hardware required to generate and distribute power for all operating spacecraft components.
- Instrumentation and Communication Subsystem (I&C): Includes antennae, transmitters and other hardware used for data signal acquisition and processing.
- Optical Telescope Assembly Subsystem (OTA): Includes the primary and secondary mirrors, fine guidance sensors and other hardware associated with the optics.
- Pointing and Control Subsystem (PCS): Includes gyroscopes, reaction wheels, and other sensors and actuators used to control spacecraft attitude.
- Safing: Includes a series of computers and software that is used to monitor mission-critical data and initiate autonomous actions to protect hardware when health and safety thresholds are breached.
- Science Instruments (SIs): Includes hardware used to expedite science.
- Thermal Control Subsystem (TCS): Includes a series of heaters, insulators and other elements designed to keep all instruments and equipment with a safe operating temperature range.

While some functions, like data dumps and vehicle configuration changes, require ground commanding, there are millions of lines of code that have been developed for autonomous monitoring and operations of all vehicle subsystems from HST's flight computer. The group of flight software (FSW) code designers and testers are a distinct group for the SE shop. SEs identify autonomous operational scheme that either needs fixing or enhancement based on some subsystem's on-orbit performance data. The SEs give a set of functional specifications to the FSW team, who then develops the code and keeps track of all FSW changes that are made. The SEs and the FSW team execute various levels of unit-, system- and acceptance-level testing prior to installing the new code on-orbit.

III. Why Do We Test?

The primary reason we test flight software is to make sure that the software performs the functions we want it to perform without any unexpected consequences due to "bugs." While the effects of a bug may appear to be innocuous at first, there is usually some remote scenario where the bug can propagate and result in a nuisance situation at best or a catastrophic failure at worst. While no testing suite can guarantee that a piece of software written by a human is completely free of bugs, a sound testing program provides engineers and decision-makers with sufficient information to make reasonably assess that a piece of software will me technical objectives without harming existing hardware or software subsystems or otherwise jeopardizing the mission.

A standard industry definition of a software "bug" (or "fault") can be found in IEEE Software Engineering Standards: "a design flaw that will result in symptoms exhibited by some object (the object under test or some other object) when an object is subjected to an appropriate test."[3] Further, a "symptom" (or "failure") is defined as "any observable misbehavior of any object (not just the object under test), such as the falsification of a requirement or an unexpected processing by-product.[3] These definitions address some key concepts with regard to testing software for HST mission operations applications:

1. Software bugs should be addressed in code design in all possible cases. Many modern systems are like HST in that they are operated by a combination of autonomous processes and active management and intervention by human users (system engineers, flight operations personnel). It would, of course, be impossible to predict and design for every inadvertent human error. However, the existence of operational policies and procedures illustrate that there are certain anticipated scenarios and predictable human behaviors that can result in software system failures. So, the best software designers will attempt to address those human behaviors in addition to errors in actual code development during the coding phase.

   HST example: The Background Memory Integrity Check (BMIC) is a software routine that continuously monitors critical memory locations and autonomously shuts the flight computer off if commands are erroneously made to those areas. If a software development or enhancement requires intentional adjustments to

memory monitored by BMIC, then the BMIC must first be disabled. A flight computer operations engineer issued a ground command to dump the contents of an area of a critical area of memory. The engineer was unaware that the area was being continuously monitored by the BMIC. Once the command was processed, the flight computer recognized the command as an attempt at an illegal function and triggered the powering of the flight computer as part of a hard-wired autonomous safety mode. This caused an interruption of a scientific observation that had to be rescheduled. Performing memory dumps are a routine part of monitoring and trending the on-board data management subsystem. If software designers had anticipated this activity, a software block of dump commands sent to areas of memory being actively protected by the BMIC could have prevented the untimely shut off of the flight computer and the interruption of the science mission.

2. A system context must be applied to software testing to fully verify functionality. While certain types of unit-level testing can verify that a section of code functions as designed, it is insufficient to verify interfaces with existing code. While these interfaces are checked out in full-up flight hardware and software system level testing (Operations Acceptance Testing, or OAT), these larger tests are usually performed just before planned on-orbit execution so the effects on schedules and resources are significant. Applying a system-level context early in the code development process provides a greater confidence in functionality of the code and can prevent costly re-tests.

   HST Example: EPS SEs wanted to enhance the software scheme used to control how the flight batteries are charged during the sunlight portion of the vehicle orbit (orbit day). The legacy scheme began opening charge-control relays to reduce charge current on the batteries only after a certain number of batteries reached their respective "charge-off" levels. The enhancement was designed to improve the state-of-charge balance of the batteries by opening relays as each battery reached its individual "charge-off" level. The flight software designers modified the code and it passed unit-level functional testing. However, the enhancement caused charge control relays to open 10-15 minutes earlier in orbit day. When the code enhancement was subjected to system-level acceptance testing, a vehicle "Cold Solar Array Protection" safing test failed. This safing test was designed to prevent the flight computer from opening relays early in orbit day when the solar array temperatures were low enough to cause relay damage due to electrical arcing. Engineers and managers had to stand down and assess whether the safing test needed to be adjusted or if the software charge-control enhancement had to be modified to accommodate the safing test. While unit testing showed the charge-control enhancement worked as planned, it could not illuminate the important safing test interaction at the system level.

IV. What is Specification-based Testing?

Specification-based testing, also known as "black-box" or "functional" testing, is a classification of testing methods whereby the tester is only knowledgeable of the inputs to the system, the system environment, and the anticipated system output. As the term "black-box" implies, the tester can see or understand internal workings of the item being tested. For example, a black-box tester of a hand-held video game would know how objects of the game's graphical user interface would respond and shift to different control inputs from a gamer. But the tester wouldn't know how the game was designed. For specification-based testing of any software, this means that the tester selects of test data and interprets test results based on the tester's understanding of the functional properties (or "specifications") of the software as opposed to the tester's understanding of the software code.

Because black-box testing focuses on function, it is often considered a useful way of verifying requirements and specifications. These specifications can either be formal (using mathematical notation) or informal (using natural, descriptive language text to describe functions). Some testers and developers argue that true black-box testing should not be performed by the designer of the software because their intimate knowledge of the program internals would prevent separation of what the program is required to do and what it is designed to do. While it would be ideal for it system users to conduct black-box testing, anyone thoroughly knowledgeable with system functionality can be a tester. Users are usually involved in field and laboratory tests. But black-box tests like volume tests, stress tests, and recovery tests often do not involve testers.

V. Advantages and Disadvantages of Specification-Based Testing

As previously discussed, HST SEs develop operational concepts for new or enhanced spacecraft functions to improve the performance of the spacecraft and/or to extend mission life. FSW engineers a primarily responsible for developing software implementations for these functions. Specification-based testing is particularly useful for the development of test cases and requirements for software applications in mission operations for HST for several reasons:

1. *The test is unbiased because the designer and the tester are independent of each other*. SEs and FSW personnel are distinct groups and both provide critical input to code development. But since the groups are distinct, SEs can objectively conduct black-box testing and verify functionality without bias of a programmer. Black-box testing can also be done within the FSW team since code designers are distinct from code testers. However, the FSE testers must be sufficiently knowledgeable of subsystem functions.

2. *The tester does not need knowledge of any specific programming languages.* Most HST SEs are not programmers. So specification-based testing techniques allow them to use their subsystem-level technical expertise in hardware design and

operation for software development without needing to learn programming languages.

3. *The test is done from the point of view of the user, not the designer.* This keeps the focus of testing on achieving the results the SEs desired within the greater context of spacecraft operations. Black-box testing can also be done within the FSW team

4. *Test cases can be designed at any point in the software development process.* As soon as SEs release the first draft of specifications, FSW testers and SEs can start developing test cases to identify inconsistencies or omissions in functional requirements and specifications. Once all iterations of requirement development are complete, the SEs can also use test cases they generated from black box testing to verify that the FSW test plan is adequate. This reduces the lag time between development of requirements and code development and testing, reduces the overall development time for code, and reduces the risk of finding costly errors or omissions in the software late in the development cycle when changes are very costly.

Despite its benefits, there are some limitations to a specification-based testing approach [2]:

1. *Testing techniques are limited in number, less systematic*. There are simply not as many black-box testing techniques that focus on software functions as there are In comparison with white-box testing techniques that are concerned with source code design. And, most black-box testing techniques rely heavily (if not solely) on the intuition and subjective system comprehension of the tester.

2. *The more prominent black-box testing techniques require automated tools*. As previously discussed, a primary benefit of black-box testing is that it can be executed by SEs with no programming experience. But to use techniques like the category-partition method (CPM), resources must be spent to acquire automated tools and the SE must spend time to learn the tool.

3. *Specifications may not be formal*. Formal specifications that express functional requirements in mathematical form would be easier to apply to most black-box testing methods. But unless the user (SE) has some programming back-ground, the specifications are likely to be in the form of descriptive text.

VI. Summary of Techniques

The applicability of different specification-based testing techniques to system testing should be assessed based on how well the technique can address certain key issues:

- *Data*: What test data should be used? How should test cases be generated from the test data to avoid redundant tests?
- *Coverage:* What are the maximum and minimum number of tests that should be run? How do you know when it is safe to stop testing? How can you make sure that important test cases are included?
- *Functional constraints*: How well can tests flush out constraints or errors?

A discussion of some specification-based testing techniques follows:

1. *Random (or "ad-hoc") testing.* This is by far the simplest (and probably most-widely used) approach. Once different types of system input are identified, the tester (or a software tool) selects and combines test data at random to form test cases. There can be no guarantees that minimal test cases cover key functions. As such, this method really can not compete with some of the more formalized approaches.

2. *Decision-table Method.* A decision table describes in a precise manner how an application behaves. It is composed of rows and columns that depict a relationship between system conditions and actions, which are obtained from the specification. For each combination of conditions, a rule exists. Each rule comprises a response that is often binary (e.g. Yes/No, True/False) and an associated list of actions. Then, for each action, an actions sequence number specifies the order in which an action should be performed if this set of conditions is true. The columns in the decision table form test cases. The format of the table is simple and the method is systematic. But important one limitation is that the decision table method can not be used to depict constraints among conditions which can result in the generation of invalid test cases. Figures 1a and 1b illustrate the decision-table method.

Figure 1a: Specification for Store Software that Provides Customer Benefits

CUSTOMER

If the customer is a new customer, offer 20% discount on next order

If the customer is a repeat customer, offer free shipping

RISK LEVEL OF GOODS

If the risk level of goods is high, then

If the customer is a new customer, check their credit record

If the customer is a repeat customer, then:

If the past orders total > £500, fine

Otherwise check their credit record

Figure 1b: Decision Table for Store Software that Provides Customer Benefits

Customer type	New customer				Repeat customer			
Risk level of goods	High risk		Low risk		High risk		Low risk	
Past orders total	>500	<500	>500	<500	>500	<500	>500	<500
20% discount next order								
Free shipping								
Check credit history								

New customer	T	T	T	T	F	F	F	F
High risk level of goods	T	T	F	F	T	T	F	F
Past orders total > 500	-	-	-	-	T	F	-	-
20% discount next order								
Free shipping								
Check credit history								

3. *Cause-Effect Graphing (CEG).* This method is actually a hardware-testing technique that was adapted to software testing by W.R. Elmendorf and others. From a natural language (informal) specification, the tester identifies causes (system inputs) and the associated effects (system outputs). Then the graph itself is constructed as a combinational logic network that connects cause and effect nodes with Boolean operators based on certain constraints as stated or implied in the specification. The graph can be traced to a decision table, which can then be used to generate test cases. The systematic approach is an advance over ad-hoc methods and provides consideration of constraints that application of the decision-tree alone does not provide. However, creation of the boolean graph can be a major draw back for larger or more complicated specifications – identifying causes and effects would be very tedious and the graphical depiction could be overwhelming. Figures 2a – 2c illustrate the CEG method [5].

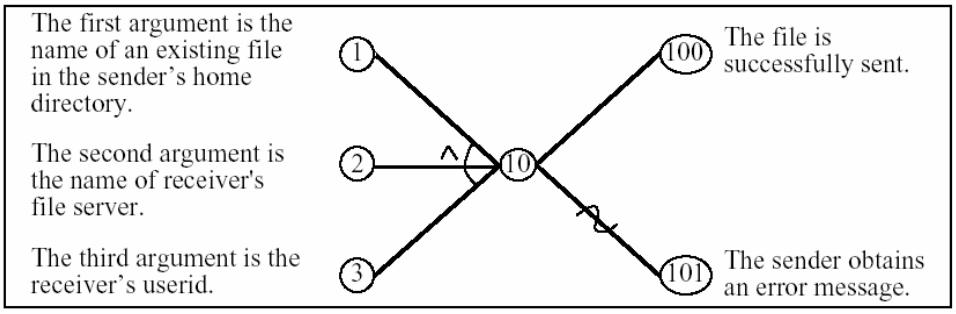
Figure 2a: Specification for the "Sendfile" Command

In a given network, the sendfile command is used to send a file to a user on a different file server. The sendfile command takes three arguments: The first argument should be an existing file in the sender's home directory. The second argument should be the name of the receiver's file server. The third argument should be the receiver's userid. If all the arguments are correct, then the file is successfully sent; otherwise the sender obtains an error message.

Figure 2b: Cause-Effect Table for the "Sendfile" Command

Causes	Effects
1. The first argument is the name of an existing file in the sender's home directory. 2. The second argument is the name of receiver's file server. 3. The third argument is the receiver's userid.	100. The file is successfully sent. 101. The sender obtains an error message.

Figure 2b: CEG for the "Sendfile" Command



4. *Category Partition Method (CPM).* The standard approach for this method is to identify categories (or *classifications*) of system input parameters and their associated choices (or *classes*) which are the different values each category could contain. The tester must also determine constraints based on the specification: how the choices interact, how the occurrence of one choice can affect the existence of another, and what special restrictions might affect any choice. Once categories, choices and constraints are identified, the information is written in a formal test specification language (TSL) which is processed by generator tool to produce test cases. While this method provides considerable coverage and adequate consideration of constraints, its major drawback is the reliance on the automatic generation tool for generation of test cases. Figures 3a – 3c illustrate the CPM method [6].

Figure 3a: Specification for the "Find" Command

Command:
find

Syntax:
find <pattern> <file>

Function:
The find command is used to locate one or more instances of a given pattern in a text file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the number of times the pattern occurs in it.

The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes ("). To include a quotation mark in the pattern, two quotes in a row ("") must be used.

Examples:
find john myfile
displays lines in the file **myfile** which contain **john**

find "john smith" myfile
displays lines in the file **myfile** which contain **john smith**

find "john"" smith" myfile
displays lines in the file **myfile** which contain **john" smith**

Figure 3b: Category Partition for the "Find" Command

```
# Unrestricted Test Specification for FIND command

Parameters:
    Pattern size:
        empty
        single character
        many character
        longer than any line in the file

    Quoting:
        pattern is quoted
        pattern is not quoted
        pattern is improperly quoted

    Embedded blanks:
        no embedded blank
        one embedded blank
        several embedded blanks

    Embedded quotes:
        no embedded quotes
        one embedded quote
        several embedded quotes

    File name:
        good file name
        no file with this name
        omitted


Environments:
    Number of occurrences of pattern in file:
        none
        exactly one
        more than one

    Pattern occurrences on target line:
    #  assumes line contains the pattern
        one
        more than one
```

Figure 3c: Final Specification for the "Find" Command

```
# Test Specification for find command
# Modified: property lists and selector expressions added
# Modified: error and single annotations added

Parameters:
    Pattern size:
        empty                                [property Empty]
        single character                     [property NonEmpty]
        many character                       [property NonEmpty]
        longer than any line in the file     [error]

    Quoting:
        pattern is quoted                    [property Quoted]
        pattern is not quoted                [if NonEmpty]
        pattern is improperly quoted         [error]

    Embedded blanks:
        no embedded blank                    [if NonEmpty]
        one embedded blank                   [if NonEmpty and Quoted]
        several embedded blanks              [if NonEmpty and Quoted]

    Embedded quotes:
        no embedded quotes                   [if NonEmpty]
        one embedded quote                   [if NonEmpty]
        several embedded quotes              [if NonEmpty] [single]

    File name:
        good file name
        no file with this name               [error]
        omitted                              [error]


Environments:
    Number of occurrences of pattern in file:
        none                                 [if NonEmpty] [single]
        exactly one                          [if NonEmpty] [property Match]
        more than one                        [if NonEmpty] [property Match]

    Pattern occurrences on target line:
    #  assumes line contains the pattern
        one                                  [if Match]
        more than one                        [if Match] [single]
```

5. *Classification Tree Method (CTM).* CTM takes a similar approach to test case generation for black box testing as CPM in that it identifies classifications and associated classes and shows relationships between them based on constraints. Benefits of CTM include that the constraints among classifications are captured in a graphical tree representation and that automation tools are helpful but no necessary for CTM. This user-friendly and easy-to-learn method encompasses all of the desirable properties discussed at the beginning of this section. A more detailed discussion of this method is provided in Sections VII and VIII.

VII. Classification Tree Method

CTM was developed by Grotchamann and Grimm in 1993. CTM is executed via the following steps:

1. Use the informal specification to identify classifications and classes. *Classifications* are the different criteria for partitioning the input of the software to be tested. *Classes* are the subsets of values for each classification. Each subset is disjoint and, by definition, classifications are also disjoint.

2. Construct a classification tree. At the top of the tree is the *general root note* which represents the unification of all inputs. The next row of figures are the top-level classifications which represent the highest-level categories of inputs and are the "children" of the general root node. Each classification has two or more classes which are also depicted in a parent-child relationship: a parent has

connector lines that fan out to associated children classes. Similarly, classes can parent classifications. Construction of the tree is complete when all identified classifications and classes are illustrated. The lowest level classes depicted are called *terminal nodes* or *leaves*.

3. Construct the test-case table. A grid is drawn below the tree. The columns of the grid result from vertical lines that correspond to the leaves of the classification tree. A tester can construct a test case via inspection and by hand and by selecting a single child class of each top-level classification. Then, following vertical lines from every child classification selected, the tester should recursively select a single child class. The combinations of marks on any row of the grid indicate distinct test cases.

4. Determine feasible test cases. The tester can identify all feasible or legal test cases from the test-case table by referring to the constraints stated or implied by the specification.

CTM stands apart from the other black-box testing techniques discussed in Section VI because it encompasses all of the properties of an ideal specification-based testing technique:

a) CTM offers a systematic approach with well-defined steps that clearly illustrate how many values of a test-relevant aspect have to be used. As opposed to simple identifying all combinations and permutations of terminal nodes, the CTM process identifies the test case cut sets and weeds out the test cases that don't add value.

b) The graphical description of the test case specifications permits easy visualization of test ideas. These ideas can then be easily reviewed by a user (SE), a developer, or a tester. Ease and clarity in communicating the test ideas build confidence in appraisals of the test and test designer and assures that no relevant test case has been overlooked.

c) CTM exhaustively checks a formal or informal specification for consistency, coverage, and omissions. That is, if an expected behavior for a given test case cannot be determined from the specification, then there is obviously something missing in the specification. As a result, CTM can be very useful in developing well-defined, well-written, testable requirements early in the software development process.

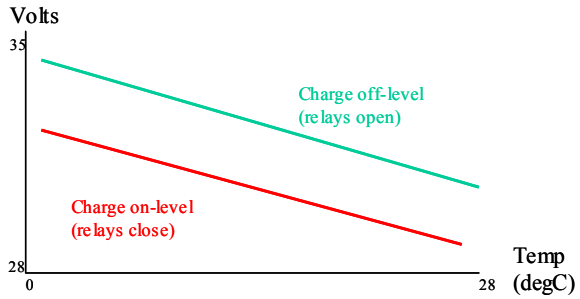
The entire CTM process can be informed by hand. However, there is a user-friendly tool that supports CTM and it is particularly useful for large systems. **Classification Tree Editor (CTE)** tool was first developed in 1996 by engineers at Daimler-Benz AG and featured an easy-to-use graphical user interface (GUI) for constructing the classification tree and the test case tables. Later models were renamed **Classification Tree Editor-eXtended Logics (CTE-XL)** added features such as automatic generation of test cases.

The current release of CTE-XL (version 1.6.1,last update 08/09/2005) can be downloaded free-of-charge at www.systematic-testing.com. More details and uses of CTE-XL are provided in Section VIII, where CTE-XL is applied in a HST mission operations software case study.

VIII. Case Study: CTM and HST Charge Control Software Testing

HST power system is supported by six, rechargeable, nickel-hydrogen batteries. Battery voltage and temperature is continuously monitored and compared to voltage-temperature (V/T) curves. Figure 5 illustrates the V/T curves used for each battery to determine charge control. When the HST's solar arrays are exposed to the Sun's energy during the HST orbital day, the batteries are being charged. Once the battery voltage at a given temperature exceeds the "charge-off" level, batteries are considered fully charged. A subset of charge-control relays are commanded open interrupt charge current from the solar arrays, scaling back the battery charge rate and preventing over-charge the batteries. When HST passes behind the earth, the sun is occulted and HST is in orbital night. During night, the batteries discharge to support the vehicle load. Once the battery voltage at a given temperature drops below the "charge-on" level, relays are closed so that the power system will be configured to resume charging when HST passes back into sunlight.

Figure 4: V-T Levels for HST Software Charge Control Scheme



Battery charge control in can either be conducted using hardware charge-current controllers (CCCs) or using by a software algorithm appropriately called Software Charge Control (SWCC).

Now an application of CTM will be applied to help develop test cases for the SWCC algorithm. For ease of illustration, we will consider the power system as a three-battery system (as opposed to the nominal six-battery system). The informal specification below is very similar to the "operations concept" description that was provided by EPS SEs when the SWCC scheme was first proposed:

> SWCC monitors voltage and temperature of batteries 1-3 and compares them to the V/T curves programmed into the Voltage/Temperature Front End (VTFE), a software emulation of the CCCs. The trickle charge initiation parameter (TCINIT) is used to verify if the system is in full charge or trickle charge. TCINIT is set to 2 batteries. If two or more batteries are above the charge-off levels, the system should go into trickle charge and open relays. When at most one battery is above the charge-on level, then the system exits trickle charge and closes relays. Regardless of the charge phase, safing tests are active to continuously monitor battery state-of-charge (SOC) and rate-of-charge (ROC). The SOC test checks to make sure that system capacity stays at or above 200 Ampere-hours (Ah). The ROC test will check to verify that total battery charge current is at adequate levels through out the orbit: $\geq$ 40A for full charge in orbit day, $\geq$ 12A for trickle charge in orbit day, and $\geq$ -90A for discharge in orbit night (negative charge current in vehicle telemetry indicates the battery is discharging). If any SOC or ROC safing test threshold is crossed, then the flight computer will activate an automatic vehicle power-down sequence.

From this informal specification, a series of classifications and associated classes can be identified (see Figure 5).

Figure 5: Classifications and Classes for HST Software Charge Control Scheme

Classification	**Class**
BAT 1	Charging, Discharging
BAT 2	Charging, Discharging
BAT 3	Charging, Discharging
VTFE-Off	AboveOff, Belowff
VTFE-On	AboveOn, BelowOn
TCINIT	Above_2batts, Below_2batts
TC_Rate	>12A, <12A
SOC Test	$\geq$200Ah, <200Ah
ROC Test	Day, Night
FullChg	$\geq$40A, <40A
TrickleChg	$\geq$-10A, <-10A
Dischg	$\geq$-90A, <-90A

The following constraints must also be generated based on nominal power system operations:

1. Batteries do not charge in orbit night. But batteries can discharge in orbit day. For example, large vehicle maneuvers can significantly increase the vehicle load and cause batteries to discharge if they are at a low, trickle ROC in orbit day.
2. If the batteries are in full-charge during orbit day, the trickle-charge initiation parameter (TCINIT=2) has not been met.
3. If the batteries are in trickle-charge during orbit day, the trickle-charge initiation parameter (TCINIT=2) has been met.
4. If the batteries are discharging in orbit night, the trickle-charge initiation parameter (TCINIT=2) has not been met.

With the classifications, classes and constraints clearly identified, the classification tree can be constructed. The CTE-XL tool was used (see Figure 6 on the following page). While there is an easy-to-read help section and tutorial, the icons and functions of the tool are pretty intuitive to anyone with fundamental computer skills (e.g. Microsoft applications) so a tester can jump right into to using the tool. The CTE-XL main window GUI is separated into four quadrants. Here are some important features:

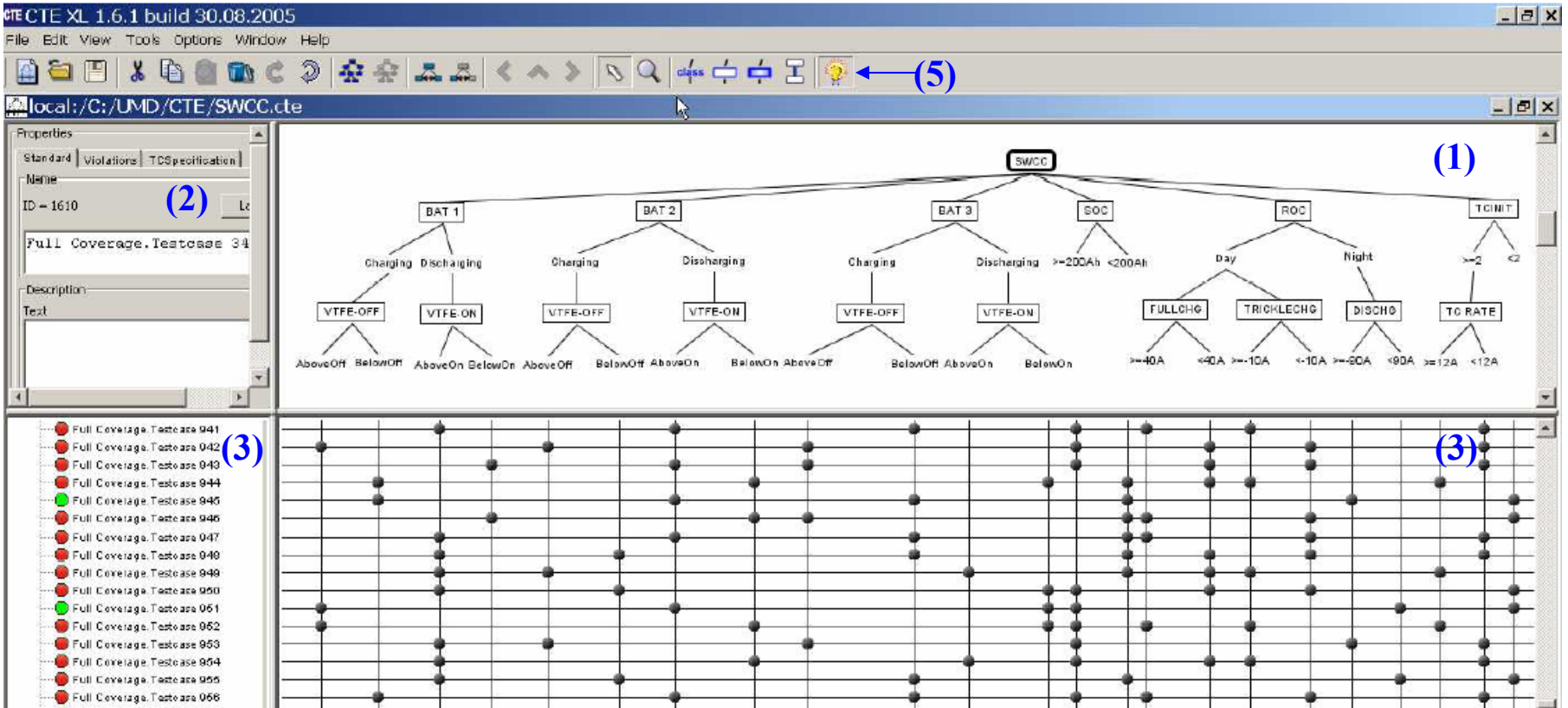
1. The top-right quadrant (labeled (1) in Figure 6) is used to construct the classification tree. The nodes of the tree are created using the icons or by right-clicking on the blank space and scrolling through a menu. Lines connecting parent and child elements are automatically drawn but can be moved or altered manually.

2. The top-left quadrant (labeled (2) in Figure 6) is the "Properties" window. The "Standard" tab can be used to enter descriptive text about any selected element in the classification tree or test-case table. The "Violations" tab will list any defined constraints that a selected test case violates. The "TCSpecifications" tab lists the specifications that apply to a selected test case.

3. The bottom-left quadrant (labeled (3) in Figure 6) has the test-case labels. These can be created manually by right-clicking on the white space in the quadrant or by using the automatic test case generator. Names for the test cases are automatically generated by the tool but they can be changed by the tester. If the classes selected to form a test case are combined in compliance with an active rules (constraints), then the circle preceding the test case name is green. If the classes selected to form a test case are not combined in compliance with an active rules, then the circle preceding the test case name is red.

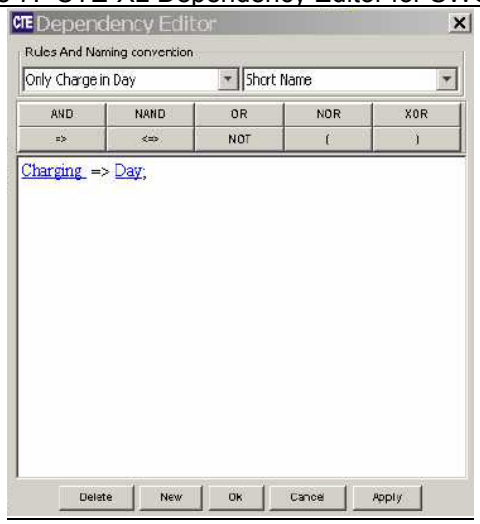
4. The bottom-right quadrant (labeled (3) in Figure 6) has the actual test-case table. The vertical lines (associated with leaves from the classification tree) appear as soon as the first test case is generated. If the test cases are being created manually (one-by-one), the horizontal lines appear when the test case is labeled. A tester can move the cursor over points in the grid and select intersection points to select a leaf. Since classes are disjoint, the tool will prevent the tested from selecting two leaves from the same parent classification and this prevents invalid test cases from being formed.

Figure 6: SWCC Classification Tree Analysis using CTE-XL



5. There is a light bulb icon (labeled (5) in Figure 6) near the top of the CTE-XL main window. When this icon is highlighted, it means that there are software rules actively being processed by the tools. These rules inform the green and red indicators on the test case labels to let the tester know hat test cases are valid. Rules are entered into the "Dependency Editor" (see Figure 7) and are expressed as boolean relationships between classes.

Figure 7: CTE-XL Dependency Editor for SWCC

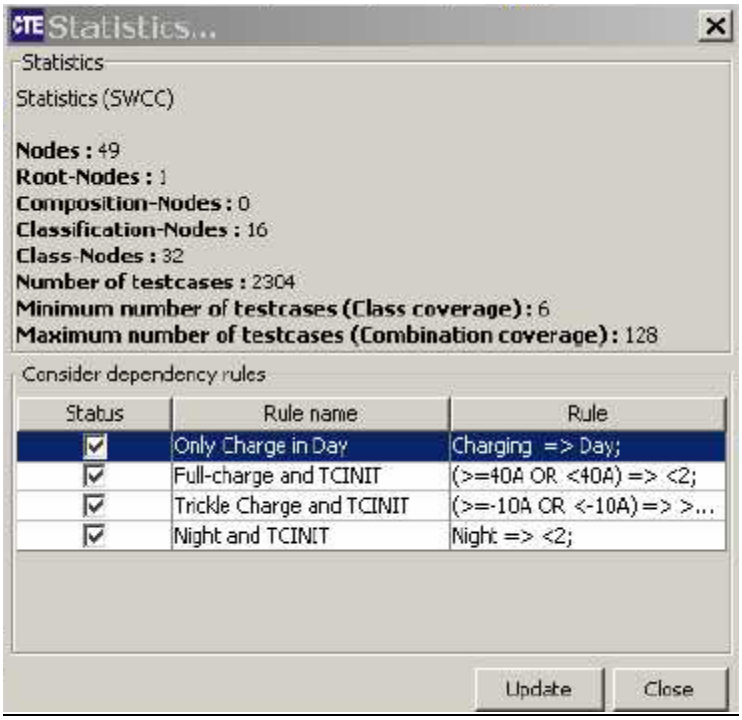


Like most software problems that would be applied to complex space mission operations, the SWCC has a lot of distinct leaves that could be combined to create an extensive list of test cases. So, it the automatic test-case generation function of CTE-XL was used. The "Test Case Generator Editor" (see Figure 8) can be used to identify how top-level classifications should be combined to create a set of test cases. For SWCC, all top-level classifications were combined ("AND") to yield a total of 2,304 test cases!

Figure 7: CTE-XL Dependency Editor for SWCC

However, the tester can easily tell that many of the test cases identified are invalid due to rule violations when she scrolls through the test case label quadrant and sees all the red circles. CTE-XL has a statistics function that generates useful metrics about the test cases that have been generated (see Figure 8).

Figure 8: CTE-XL Statistics Generator for SWCC



The Statistics tool shows that when the dependency rules are applied, only 128 of the 2,304 test cases generated are required to provide the maximum combination coverage. The tester can use this inform to identify the distinct combinations of classes that for test purposes.

IX. Conclusion

This project has discussed how specification-based testing can be a useful tool in engaging users and non-programmers in software testing in ways that improve the process of generating requirements and increases the confidence level that software meets specifications. While several different specification-based testing methods were discussed, the most effective method was classification tree method. The CTE-XL tool proved to be a powerful but easy-to-use tool for software programmers and non-programmers alike to apply the classification tree method.

References

1. Beizer, B. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. New York: John Wiley & Sons, Inc., 1995.

2. Chen, T. Y. and Poon, P. L. "Experience With Teaching Black-Box Testing in a Computer Science/Software Engineering Curriculum." *IEEE Transactions on Education* 47, #1 (2004)

3. Grochtmann, M., Grimm, K., Wegener, J. "Tool-Supported test Case Design for Black-Box Testing by Means of the Classification-Tree Editor." *EuroSTAR'93*—1st European International Conference on Soaftware Testing Analysis and Review, 25-28 (October 1993): 169-176.

4. Mogyorodi, G. "What is Requirements-Based Testing?" *Crosstalk: The Journal of Defense Software Engineering* (March 2003): 12-15

5. Nursimulu, K. and Probert, R. L. "Cause-Effect Graphing Analysis and Validation of Requirements." Bell-Northern Reaseach and telecommunications Software Engineering Research Group, Department of Computer Science, University of Ottawa, Canad.

6. Ostrand, T. J., and Balcer, M. J. "The Category-Partition Method for Specifying and Generating Functional Tests." *Communications of the ACM* ,31, #6 (June 1998):676-686.

7. Pettichord, B. "Five Ways to Think about Black Box Testing."

ACRONYM LIST

A	Ampere
Ah	Ampere-hour
BMIC	Background Memory Integrity Check
CCC	Charge-Current Controller
CEG	Cause-Effect Graphing
CPM	Category-Partition Method
CTE-XL	Classification Tree Editor – eXtended Logics
CTM	Classification Tree Method
DMS	Data Management Subsystem
EPS	Electrical Power Subsystem
FSW	Flight Software
GSFC	Goddard Space Flight Center
GUI	Graphical User Interface
HST	Hubble Space Telescope
I&C	Instrumentation and Communication Subsystem
NASA	National Aeronautics and Space Administration's
OAT	Operations Acceptance Testing
OTA	Optical Telescope Assembly Subsystem
PCS	Pointing and Control Subsystem
ROC	Rate-of-Charge
SI	Science Instrument Subsystem
SE	System Engineer
SOC	State-of-Charge
SWCC	Software Charge Control
TCINIT	Trickle-Charge Initiation Parameter
TCS	Thermal Control Subsystem
TSL	Test Specification Language
V/T	Voltage/Temperature (Curve)
VTFE	Voltage/Temperature Front End