Formal Functional Test Designs With a Test Representation Language

J. M. Hops
Radio Frequency and Microwave Subsystems Section

This article discusses the application of the category-partition method to the test design phase of hardware, software, or system test development. The method provides a formal framework for reducing the total number of possible test cases to a minimum logical subset for effective testing. An automatic tool and a formal language have been developed to implement the method and produce the specification of test cases.

I. Introduction

The focus of this article is on how the category-partition method, a method for specifying functional tests [4], can be applied to the test design phase of the testing life cycle, a required part of any DSN implementation task. Before describing the method itself, the requirements for the test design phase need to be clearly defined, as well as how they fit into the *JPL Software Management Standard*.[1] This discussion can be found below, in Section II.

Section III centers on some methods often used in the test design phase. The category-partition method is described in detail. Included in this section are the background of the method, a step-by-step description of how to implement it, and a demonstration of the method applied to a simple example.

The subsequent section, Section IV, introduces the Test Representation Language (TRL). TRL is a formal language for specifying test designs that have been created with the category-partition method and a computer tool for automatically generating test cases from the formal specification. The example from Section III is presented using the TRL format.

The conclusions in Section V provide some insight into the results that have been achieved and offer some suggestions for further study and data collection that may be necessary to assess the contribution of the TRL tool developed and the category-partition method used.

II. Problem Definition

A. Testing Life Cycle

In the field of software engineering, one is often faced with the challenge of creating an integrated, working system based on inadequate and meager requirements. The waterfall life cycle for software development, wherein requirements are systematically refined, architectural and detail design established, code written, and then the system tested, has become one of the accepted methods for dealing with the ambiguities and vagueness of the original requirements. The testing portion of this development life cycle, however, is not so clearly defined or widely accepted.

[1] *JPL Software Management Standards Package, Version 3.0*, JPL D-4000 (internal document), Jet Propulsion Laboratory, Pasadena, California, December 1988.

Figure 1 depicts the testing life cycle used in the development projects for the DSN. This life cycle is described in the *JPL Software Management Standard*[2] and is similar to the software development standard adopted by the Department of Defense, DOD-STD-2167A [2]. Table 1 defines the acronyms used in Fig. 1, along with the title of the document each acronym stands for, the phase in the testing life cycle during which the document is used or produced, and finally, whether a description of the document's contents is in the *JPL Software Management Standard*[3] and [2].

As noted in both Fig. 1 and Table 1, a key component is missing from the standards—a definition of the input, output, and purpose of the test design phase. This gap between the requirements for testing, produced in the test requirements analysis phase, and the detailed test procedures, produced in the test specification phase, is the phase during which the category-partition method can be most useful. The test design phase is explored in detail in the following section.

B. Test Design Phase

The dictionary definition of the word "design" is to conceive and to devise for a specific purpose. During the test design phase, the "specific purpose" that the test engineer is concerned with is meeting the test objectives and requirements determined in the test requirements analysis phase; what the test engineer is trying "to conceive and to devise" are the necessary and sufficient ways of validating the functional and performance requirements of the entire system. Therefore, the purpose of the test design phase is *to conceive and specify the environmental and system attributes that verify requirements and meet test objectives for each test requirement in the test plan and for each requirement in the functional and software requirements documents.*

Based on this definition of the purpose, the input to this phase is relatively simple to identify. It is

(1) The test objectives as documented in the subsystem integration and test plan and/or the software test plan

(2) The functional and performance requirements and system design as documented in the software specification documents or the functional and software requirements documents

(3) Any other pertinent design or requirements information that may be available, such as interface agreements and the preliminary software operator's manual

The output from the test design phase to the next phase, however, is not so easy to identify. The products to be developed are ways to validate requirements, which will be referred to as test designs. These designs are not expected to be test procedures specified to enough detail to be run by an operations engineer or possibly a quality assurance engineer; the test procedures written to that detail will eventually be written in the subsequent phase of the testing life cycle, the test specification phase. The test designs can have some ambiguity in the sequence of steps, the testing range of certain parameters, or the actual testing steps themselves.

Additionally, each test design should directly imply or specify a group of test cases. The test cases should have specific values for environmental and/or system parameters that have an effect on how the system under test will behave. Each of the test cases should also include the expected response or behavior of the system.

With this in mind, the output of the test design phase can be stated as follows:

(1) Test design specifications that

  (a) Are traceable to test objectives and functional and/or software requirements

  (b) Directly imply or specify a group of test cases that can be individually executed but share the same setup procedures

  (c) Identify the environmental and system features that are to be set or observed to control and determine the behavior of the system

  (d) Pass criteria for the group of test cases, and

(2) Test cases that specify

  (a) The environmental and/or system parameters and system states that should exist before the test case is executed

  (b) The test action or step to be taken to initiate the system behavior

  (c) The expected system behavior after the action has been taken

[2] Ibid.

[3] Ibid.

In the following sections, methods for determining the test designs and for automatically producing the documentation for the test cases are presented.

III. Method of Solution

A. Test Design Methods

There are many ways to create test designs that meet the needs of a certain project. Four of these methods are discussed below: the representative set method, the ad hoc method, the all-permutations method, and finally the category-partition method.

A common method for determining the number and contents of the test designs and test cases that should be transformed into test procedures is selecting a representative set of normal conditions and parameters that prove that the system works and meets requirements. On a project using this method, the emphasis will be on demonstrating that the system works rather than testing the system to detect failures, but the repeatability of the test procedures and the traceability to the requirements being tested is generally good.

On projects that are particularly short of time, money, and personnel, the test design phase is almost totally skipped. In this case, the test design method can be characterized as ad hoc. The ad hoc test case selection process is particularly prone to missing important aspects of the system behavior that could help determine where the problems are. The emphasis on a project using this method is almost always on getting the system out the door. Traceability to requirements is often poor. And most devastating of all, test repeatability is sacrificed; when a failure eventually occurs and the problem solved, it is very difficult to verify that the fix was correct because the conditions that caused the failure cannot be repeated.

Though not often seen, another method for selecting test designs and cases is a brute force method of analyzing all permutations of system parameter values. With this method, the test designs and cases are easily traced to requirements and test objectives, but it takes a lot of time and effort to analyze each permutation and decide which ones are valid and which ones are meaningless. This method allows the test engineer to find test cases that lie on the extreme boundary of the valid input space and therefore is good for error detection.

A recommended method for determining test designs is the category-partition method [4]. This method combines the benefits of choosing normal cases with the error exposing properties of the all-permutations method. Traceability can be maintained quite easily by creating a test design for each test objective in the test plan. By using an automatic tool to create the test cases based on the test design, the subsequent effort to transform the test cases into test procedures is simplified. The method allows the rapid elimination of undesired test cases from consideration and easy review of test designs by peer groups.

Section III.B discusses the category-partition method in general and is followed by Section IV, which presents the Test Representation Language (TRL) that can be used to implement the method and produce the test cases using the TRL tool.

B. Category-Partition Method

1. Background. The category-partition method was first presented by Ostrand and Balcer in 1988 [4]. A follow-on article in 1989 [1] discussed a test specification language and a tool for the automatic generation of test scripts that could be compiled and executed in the test environment that they had set up at Siemens Corporate Research. As pointed out in these two articles, the category-partition method is a way of analyzing the functional and software requirements of a system in order to determine test cases to be run. The method relies exclusively on the test engineers' reading of the requirements and design documents and their judgment of exactly which test cases should be selected for procedure development. If a formal requirements specification language is used to document the requirements and design, other methods may be more useful, such as the ones described in the article by Richardson et al. [5]. However, it is not often that the test engineer is presented with a functional requirements document or a software requirements document that is written this formally. Therefore, a structured method, such as the category-partition method, is needed to provide a systematic approach to developing test specifications from informal representations of the required system behavior.

The following sections discuss the steps in the category-partition method. The steps have been organized differently from the procedure discussed in the primary references, [4] and [1]. The organization of steps presented below has proven useful in communicating the method to the test engineers on JPL projects.

2. Steps in the Category-Partition Method. The category-partition method consists of the following four steps:

(1) Functional decomposition

(2) Category analysis

(3) Partition value analysis

(4) Partition constraint analysis

Each of these steps is discussed in the following sections.

a. Step 1: Functional Decomposition. The first step in the category-partition method is functional decomposition. The purpose of this step is to decompose the specification and/or requirements into functional units that can be tested independently. A secondary purpose of this step is to identify the parameters that affect the behavior of the system for each functional unit.

The requirement space is subdivided into subgroups, which may or may not overlap in some aspect. Each subgroup clearly identifies the requirements being tested and the input, output, and environmental parameters that affect how the system meets the requirements. The types of parameters that should be considered are user input, input from external interfaces, environmental input, output to another (observable) portion of the system, output to a user or external interface, output to the environment or state of the system, or maybe even the sequence of events. Note that there will be times when some of the parameters are not explicitly stated in the requirements specification, and therefore implicit parameters will have to be determined.

For an example, assume the following requirement specification has been decomposed from the requirement space: Sort an integer array either in ascending or descending order. The parameters mentioned explicitly in this requirements statement are *the array* and *an indication of sort order*. Implicitly, however, *the result of the sort operation* is also a parameter for this requirement.

The next step of the procedure is to further analyze the parameters identified and determine the characteristics, or categories, of the parameters that affect program or system execution.

b. Step 2: Category Analysis. The second step in the category-partition method is category analysis. The work done in the previous step, identifying functional units and explicit and implicit parameters, is carried further by determining the properties or subproperties of the parameters that would make the system behave in different ways. The test engineer should analyze the requirements and determine the features or categories of each parameter and how the system may behave if the category were to vary its value. If the parameter undergoing refinement were a data item, then categories of this data item may be any of its attributes, such as type, size, value, units, frequency of change, or source.

Choosing *the array* from the example in step 1 for further refinement, the categories that may be derived from the specification are *array size*, *the values in the array*, and, because the functional unit is a sorting function, *the arrangement of the values in the array*.

As can be seen, the original requirement statement said nothing about the valid range of *array size*. This step, along with the next one, tends to point out deficiencies in the requirements specification. The test engineer will have to work closely with the author of the requirements and the designers in order to resolve the ambiguities and uncertainties that surface from this analysis.

c. Step 3: Partition Value Analysis. After all the categories for the parameters of the functional unit have been determined, the next step is to partition each category's range space into mutually exclusive values that the category can assume. In choosing partition values, the focus should be on error-exposing values. The discussion on boundary value testing in Myers' book [3] and revealing subdomains in the article by Weyuker and Ostrand [6] should prove useful as references.

The partition values should include all possible kinds of values, especially the ones that will maximize error detection. Important values to look for are boundary values, extremal and nonextremal values, values that represent special cases or interactions, and valid and invalid values.

Returning to the example and using the category *array size* for illustration, the five partition values are

(1) 0

(2) 1

(3) 2 to the Upper bound minus 1

(4) Upper bound

(5) Greater than the Upper bound

It can be seen that *0* and *Greater than the Upper bound* represent error conditions that the sort function will have to process, while *1* and *Upper bound* represent special cases or boundary values. All the values between *2* and the *Upper bound minus 1* (inclusive) have been grouped together because the sorting function is expected to behave the same in this range; an error in processing that occurs for a particular value in this range should occur for all the

values in this range. It is left up to the test specification phase of the testing life cycle to determine the exact, or random, values that should be used to verify this partition in the test procedure.

The fact that two of the five values in this example have already been identified as being representative of error conditions gives one a head start on the next step of the category-partition method.

d. Step 4: Partition Constraint Analysis. The purpose of this final step is to refine the test design specification so that only the technically effective and economically feasible test cases are implied. There are three types of constraints defined in the category-partition method as described in [4]: errors, limits, and conditions.

An error constraint applied to a partition value is used to indicate that the partition value represents an exception state that the system under test should note and report without processing any further. Partition values of this type need to be tested in one test case but no more, due to the way exceptions are usually handled. Examples of partition values that should have error constraints are *0* and *Greater than the Upper bound* in the category of *array size.*

A limit constraint is for limiting the number of times a partition value will be used in the resulting test cases. Limit constraints can be applied to a test design in order to control the actual number of test cases implied. When economic feasibility, as in restricted time and resources, is a factor in the test execution, the limit constraint will help the test engineer to eliminate some of the test cases that seem redundant. In the above example, the test engineer may want to limit the number of times that an *array size* of *Upper bound* is used.

The remaining type of constraint is the conditional constraint. Determining these types of constraints is where the majority of the intellectual effort is spent. This part of the analysis specifies which partition values from one category can be used with the partition values of another category. Conditional constraints are specified in pairs: preconditions and postconditions. Preconditions are states or conditions that must co-occur for a particular partition value to be used in a test case; postconditions are the states or conditions that are set when a partition value is used. To illustrate their use, a slightly more involved example is discussed.

Starting with the category of *array size* and the partitions determined in the previous step, the types of conditions that are expressed by each partition value are analyzed. It can be seen that the values represent three separate conditions:

(1) "Error occurs" (for partition values of *0* and *Greater than Upper bound*)

(2) "Size is normal" (for partition values of *2 to Upper bound minus 1* and for *Upper bound*)

(3) "Size represents a degenerate array" (for an array size of *1*)

Clearly, if everything else is set appropriately, the valid partition values of the category *result* will be dependent on these conditions. Assume the following four partition values were identified in step 3 for the *result* category: *error notification, array unchanged, array in ascending order*, and *array in descending order.* A precondition for the result *error notification* is that the postcondition "Error occurs" has been set. For the values of *array in ascending order* and *array in descending order*, the postcondition of "size is normal" must have been set before these values could be used in a valid test case. The *result* of *array unchanged* could possibly be a result of many conditions, one of which is that the *array size* is *1*, where the "size represents a degenerate array."

3. Example Application of Category-Partition Method. Table 2 provides the results of the method applied to the example that has been discussed throughout the previous sections of this article.

IV. Test Representation Language (TRL)

The TRL was developed to implement the category-partition method. When used during the test design phase of the testing life cycle, the TRL files will form concise and uniform representations of the test designs for the functional testing of the system.

The TRL tool that implements the TRL language processes the ASCII formatted TRL files and produces ASCII formatted result files that document the individual test cases implied by the test design. The TRL tool documents the description, categories, and partition values to be used in each test case as they were documented in the input file. Each TRL file is created and changed with an ASCII editor and therefore can be easily modified to adapt to changes in functional specifications. The resulting test cases can be used during engineering tests of the system under test to verify preliminary procedures and functions while work continues in the test specification phase on transforming the test cases into formal detailed test procedures.

The TRL tool was written in the C programming language and can be ported to any platform; the SUN/SPARC and DOS environments are the computer platforms on which it currently runs. This tool differs from the one described in [1] in that the TRL tool is a general permutation control language that can be used in any environment; the output of the TRL tool is ASCII files that can be used for documentation rather than an executable test script, as in [1].

A. TRL Language Definition

The TRL provides a way to describe many test cases with one TRL file. The language consists of 1 comment character, 11 key words, 2 field demarcation characters, a logical AND character, and a logical NOT character. The processing rules for the key words, comments, and fields appear in the following sections, and a summary of the Test Representation Language appears in the Appendix.

1. Special Characters. There are five special characters in the TRL character set.

(1) Comment character = asterisk (*)

(2) Start field character = open bracket ([)

(3) End field character = close bracket (])

(4) Logical AND character = comma (,)

(5) Logical NOT character = exclamation point (!)

The asterisk is for initiating a comment line, which is a line defined by the comment character appearing as the first non-white-space character on a line in the TRL file. The start field character and end field character are for specifying the beginning and ending of a partition constraint field. Partition constraint fields are discussed in Section IV.A.3.

The logical AND character and the logical NOT character are for specifying a logical relation inside a partition value constraint field that is used for setting conditional constraints.

2. Line Key Words. There are two types of key words in the TRL: line key words and field key words. To be recognized as valid, the line key word should be the first word on a line. These key words are used to initiate a description of the test designs (DESCRIPTION), indicate the beginning of the categories and partitions (PARAMETERS), indicate a certain type of category (TYPE), specify the name of a category (NAME), set error message text (MESSAGE), and indicate the start of the block that describes the partition value and constraints of each category (SAMPLES). The line key words are, respectively: DESCRIPTION, PARAMETERS, TYPE, NAME, MESSAGE, and SAMPLES.

3. Field Key Words. The field key words are used in the partition value constraint fields to either describe a partition value (LABEL) or to specify the constraints determined during step 4 of the category-partition method. The field key words for setting labels and constraints are: SET, IF, LIMIT, ERROR, and LABEL.

A partition value constraint field is associated with a particular partition value by its physical location in the partition value block. A line in this block consists of the partition value text followed by zero or more constraint fields. The constraint fields can extend beyond the physical line of the TRL file, but the partition value text cannot. Partition value constraint fields are started by the start field character ([) and ended by the end field character (]).

As previously mentioned, partition value text cannot start with the comment character (*) and cannot contain any start or end field characters, ([) or (]).

B. Example Application of Category-Partition Method with TRL

In this section, the same example from Section III.B.2 will be discussed, but this time TRL will be used. To avoid confusion, the procedures for creating a test design using TRL are referred to as stages, and the procedures for implementing the category-partition method are referred to as steps. These stages will be performed for each functional unit and/or test objective in the system under test.

1. TRL Stage 1: Unconstrained Representation. The first stage in the TRL procedure is to create an unconstrained representation of the test design. This is accomplished by performing the first three steps in the category-partition method.

(1) Step 1: functional decomposition (Section III.B.2.a)

(2) Step 2: category analysis (Section III.B.2.b)

(3) Step 3: partition value analysis (Section III.B.2.c)

As for creating a TRL file, the following TRL key words and information should be created:

(1) DESCRIPTION key word and the description block. Create a description block that contains the requirements to be tested, the pass criteria to be used, and any other information pertinent to the test design.

(2) PARAMETERS key word. Start the parameter specification block.

(3) TYPE key words and NAME key words. For each type of parameter and category identified in step 2 of the category-partition method, create a TYPE and NAME specification in the TRL file.

(4) SAMPLES key words and the partition values. For each category, add in the unconstrained partition values that the category can assume during a test.

Note that the example in Fig. 2 with the unconstrained representation would produce 1440 test cases.

2. TRL Stage 2: Error Constrained Representation. The second stage of this process is to add in the error indicators and the message descriptions. This corresponds to a portion of the fourth step, partition constraint analysis, in the category-partition method.

The following key words and information should be added to the TRL file:

(1) ERROR field key words. For each partition value that should raise an exception during testing, create an [ERROR] field and add it to the test design.

(2) MESSAGE key word and error message list block. For each ERROR field, make sure there is a corresponding error message in a message list block.

See the example for TRL stage 4 for an illustration. When the error indicators are added to the three partition values as indicated below, 651 test cases result (Table 3).

3. TRL Stage 3: Condition Constrained Representation. The third stage of test design creation using TRL is probably the most difficult and time consuming. Adding in the conditional statements to make sure that only the technically feasible combinations of partition values get produced in the resulting test cases often takes many iterations. Investigating exactly which combinations are valid when used together, and what the expected output of the system should be, can expose many inconsistencies and undocumented requirements.

This stage, similar to the previous one, corresponds to the fourth step in the category-partition method. The purpose of this stage is to determine the precondition and postcondition pairs that describe the behavior of the system under test.

To modify the existing TRL file so that the conditions are expressed, the SET and IF field key words must be added. There will be some occasions where the addition of "don't care" partition values, or even the addition of repeat partition values with different conditional fields attached, will be necessary in order to produce the optimum set of resulting test cases.

The following key words and information should be added to the TRL file:

(1) SET field key words and postconditions. For each partition value that should cause a postcondition to exist if it is used in a test case, create a postcondition value and append it to the inside of the [SET] field. Use a logical AND character (,) to separate multiple postconditions being set for the same partition value.

(2) IF field key words and preconditions. For each partition value that is valid only when combined with a particular partition value in another category, append the condition value to the inside of the [IF] field. Use a logical AND character (,) to separate multiple preconditions to be applied to the same partition value. A logical NOT character (!) in front of a condition expresses that a condition should NOT exist in order for the particular partition value to be used in a resulting test case.

Again, the reader should refer to the stage 4 discussion in Section IV.B.4 for an example that has preconditions and postconditions. Before the LIMIT fields are added to the TRL file in stage 4, the TRL results file contains 32 test cases, which together represent the complete functionality of the requirement being tested in this functional unit. The purpose of the fourth stage is to reduce the number of test cases even further so that testing of this functional unit takes less resources.

4. TRL Stage 4: Limit Constraint Representation. This final stage of TRL file development produces the limit constrained representation of the test design. The purpose of the LIMIT field is to specify how many times a partition value can be used in the resulting set of test cases. Setting these limit values corresponds to the last step, or substep, of the category-partition method, where the remaining partition value constraints are determined.

Also included in this stage is the labeling of the partition values. The purpose of the labels is to provide the test engineer, who is performing the tests or transforming the test cases into detailed procedures, as much information about the test case as possible. The labels recommended are ones that describe the partition value in terms of its range, such as "normal," "low boundary," "high out-of-bounds," etc.

Therefore, the following key words and information should be added to the test design:

(1) LIMIT field key words. For each partition value that should only be used a certain number of times, n, in the resulting test cases, create a [LIMIT n] field. Note that partition values with an [ERROR] field are automatically limited to one test case.

(2) LABEL key words and label text. For some or all of the partition values in the TRL file, add a [LABEL label_text] field such that the "label_text" provides a description of the partition value that will be useful to the other test engineers.

The example given in Fig. 3 produces 24 test cases when processed by the TRL tool. Figure 4 gives an excerpt of the first two test cases from the resulting test cases produced by the TRL tool from the TRL test design documented in Fig. 3.

V. Conclusion

The purpose of the test design phase is to determine a set of technically feasible and resource-frugal test cases that meet the test objectives of the test plans and that verify the functional requirements of the system under test. The category-partition method can be used to determine test designs that meet this goal.

The Test Representation Language (TRL) and the TRL computer tool, used to process files written in the language, have proven very useful and efficient in implementing the category-partition method. For one task in particular at JPL, the Block V Receiver Task, the test cases that result from the TRL tool are being used to verify the system requirements in the engineering testing stage. Detailed test procedures are being developed based on the output of the tool. The TRL tool was also used on the Microwave Generic Controller Task to help develop the system and software acceptance test procedures.

As of yet, no objective data have been collected that can be used to compare the results of the testing process changes introduced by the use of the TRL tool. However, the qualitative feedback received from both test engineers and software designers is that the category-partition method and the TRL tool help them engineer tests rather than just perform tests. The effects of the method and the tool may be hard to quantify on an ongoing project. A way could be found to determine these effects if a small, controlled case study were to be initiated where two groups perform the same job—one using TRL and the category-partition method and the other using neither.

Work is continuing on enhancing the TRL tool to meet the needs of the test engineers using it. Some key words are being added to allow some very fine-tuned control over which test cases get included in the results.

In summary, the purpose and requirements of the test design phase of the testing life cycle have been explored and defined. The category-partition method and the TRL tool are efficient ways to produce the test designs and resulting test cases needed as input to the following phase of the testing life cycle. The Test Representation Language and the TRL tool can be of use to the test engineer or programmer no matter what level of testing is being performed. More effort in gathering the necessary metrics would be useful to be able to quantify the benefits received from implementing this process. If qualitative results are enough, however, most organizations could profit from an implementation similar to the TRL tool and the category-partition method for bridging the gap between test requirements and test specifications.

References

[1] M. J. Balcer, W. M. Hasling, and T. J. Ostrand, "Automatic Generation of Test Scripts From Formal Test Specifications," *SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 210–218, December 1989.

[2] *Military Standard Defense System Software Development*, DOD-STD-2167A, Washington, DC: U.S. Government Printing Office, February 1988.

[3] G. J. Myers, *The Art of Software Testing*, Wiley Series in Business and Data Processing, New York: John Wiley and Sons, 1979.

[4] T. J. Ostrand and M. J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, June 1988.

[5] D. J. Richardson, O. O'Malley, and C. Tittle, "Approaches to Specification-Based Testing," *SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 86–96, December 1989.

[6] E. J. Weyuker and T. J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, pp. 236–246, May 1980.

Appendix

Test Representation Language (TRL) Summary

Character or Key Word	Purpose and/or Usage
*	Indicates a comment line.
DESCRIPTION	Indicates the start of a description block that will be included in test cases.
PARAMETERS	Indicates the beginning of parameter specifications.
NAME	Specifies the name of a parameter or category.
TYPE	Indicates the type of category.
SAMPLES	Indicates the beginning of a samples block defining the partition values and constraints. Comma (,) is used for logical AND; exclamation point (!) for logical NOT.
[	Beginning of sample value constraint field.
]	End of sample value constraint field.
IF	Field identifier indicating that postcondition constraints are listed in the current field. Comma (,) is used for logical AND; exclamation point (!) for logical NOT.
LIMIT *m*	Field identifier indicating that the number of test cases involving this partition value should be limited to *m*. If *m* is unspecified, the limit is one test case.
LABEL	Field identifier indicating that the specified label should be listed for this partition value.
ERROR *n*	Field identifier indicating that the sample value is an error exit. The error can be specified using the optional *n*.
MESSAGE *n*	Indicates that a message block follows corresponding to the errors in the partition values. The message number can be specified using the optional *n*.
Command line options	For performing "count only" (-c), writing results into separate files (-s), including preconditions/postconditions in output (-p), and including the partition label text in the output (-l).

Table 1. Testing life cycle input and output.

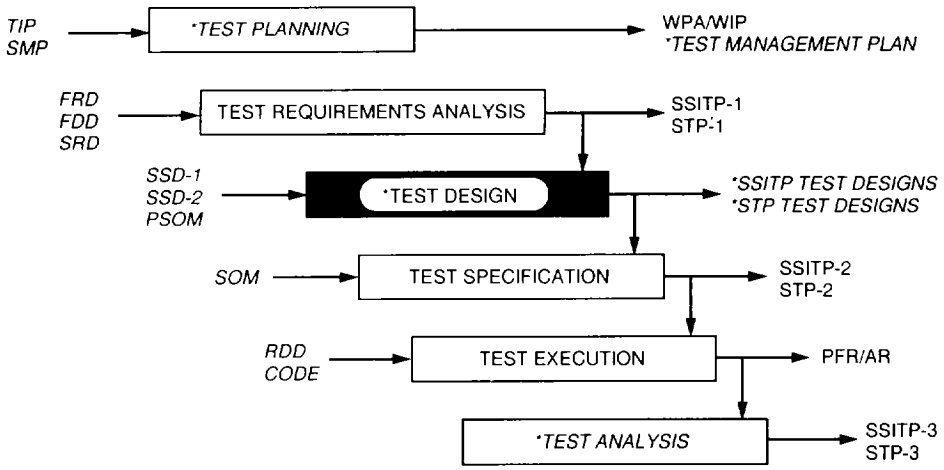
Acronym	Document title	Testing life cycle phase	Sufficiently described in standards?
D-4000	JPL Software Management Standard	All	Yes
TIP	Task implementation plan	Test planning	Yes
SMP	Software management plan	Test planning	Yes
WPA/WIP	Work package agreement/work implementation plan	Test planning	Yes
Test management	Test management plan for defining procedures of complete testing cycle	Test planning	No
FRD	Functional requirements document	Test requirements analysis	Yes
FDD	Functional design document	Test requirements analysis	Yes
SRD	Software requirements document	Test requirements analysis	Yes
SSITP-1	Subsystem integration and test plan-1, requirements	Test requirements analysis	Yes
STP-1	Software test plan-1, requirements	Test requirements analysis	Yes
SSD-1	Software specification document-1, architecture	Test design	Yes
SSD-2	Software specification document-2, detail design	Test design	Yes
PSOM	Preliminary software operator's manual	Test design	Yes
SSITP test designs	Subsystem integration and test plan—test designs	Test design	No
STP test designs	Software test plan—test designs	Test design	No
SOM	Software operator's manual	Test specification	Yes
SSITP-2	Subsystem integration and test plan-2, procedures	Test specification	Yes
STP-2	Software test plan-2, procedures	Test specification	Yes
RDD	Release description document	Test execution	Yes
PFR/AR	Problem failure report/ anomaly report	Test execution	Yes
SSITP-3	Subsystem integration and test plan-3, report	Test analysis	No (phase) Yes (report)
STP-3	Software test plan-3, report	Test analysis	No (phase) Yes (report)

Table 2. Example of formal functional test design: application of category-partition method (functional unit: sort an integer array either in ascending or descending order).

Categories	Partition values	Partition constraints: postconditions	Partition constraints: preconditions
Array size	0 (array unspecified)	SET "error occurs"	
	1 (degenerate array)	SET "size represents a degenerate array"	
	2 to upper bound minus 1	SET "size is normal"	
	Upper bound	SET "size is normal"	
	Greater than upper bound	SET "error occurs"	
Array values	All zero	SET "values identical"	IF "size is normal"
	All the same but nonzero	SET "values identical"	IF "size is normal"
	All negative values	SET "not identical"	IF "size is normal"
	All positive values	SET "not identical"	IF "size is normal"
	Mixed positive, negative, and zeros	SET "not identical"	IF "size is normal"
	Don't care	SET "values identical"	IF "error occurs" or IF "size represents a degenerate array"
Value arrangement	Minimum value before maximum value		IF "not identical"
	Maximum value before minimum value		IF "not identical"
	Don't care		IF "values identical" or IF "error occurs"
Sort order	Unspecified	SET "error occurs"	IF "size is normal"
	Ascending order	SET "ascending order"	IF "not identical"
	Descending order	SET "descending order"	IF "not identical"
	Don't care		IF "values identical" or IF "error occurs"
Result	Error notification		IF "error occurs"
	Array unchanged		IF "values identical"
	Array in ascending order		IF "ascending order"
	Array in descending order		IF "descending order"

Table 3. Stage 2 example of error indicators added to partition values.

Category	Partition value	Fields
Array size	0	[ERROR] ...
Array size	Greater than Upper bound	[ERROR] ...
Sort order	Unspecified	[ERROR] ...



LEGEND: *D-4000 INPUT* D-4000 PHASE OR OUTPUT **PHASE OR OUTPUT NOT DEFINED IN D-4000*

Fig. 1. Testing life cycle.

```
Example:
 *
 DESCRIPTION
 *
 Functional Unit:  Sort an integer array either in ascending or
                   descending order.
 *
 PARAMETERS
 *
 TYPE     Input-Categories for Parameter: Array

    NAME array size
    SAMPLES
         0
         1
         2 to Upper Bound minus 1
         Upper Bound
         greater than Upper Bound

    NAME array values
    SAMPLES
         all 0's
         all the same but not 0
         all negative
         all positive
         mixed +/-/0
         don't care

    NAME value arrangement
    SAMPLES
         minimum before maximum
         maximum before minimum
         don't care
 *
 TYPE     Input-Parameter: Sort Order

    NAME sort order
    SAMPLES
         ascending
         descending
         unspecified
         don't care
 *
 TYPE     Output to program or change in state

    NAME result
    SAMPLES
         error notification
         array unchanged
         array in ascending order
         array in descending order
 *
 * end of file
 *
```

Fig. 2. Stage 1 example of an unconstrained representation of a test design.

```
*************************************************************************
*        STAGE 4.  ADD [LIMIT ] AND [LABEL ] FIELDS to the TRL File     *
*************************************************************************
*
DESCRIPTION
*
        Test Representation for SORT requirement.
        File Name:       SORT.TRL
        Version:          1.5  Errors/Messages/Conditions/Limits/Labels
        Last Modified:   9/4/91
        Modified By:     J. Hops
*
PARAMETERS
        TYPE     Input-Categories for Parameter: Array
                 NAME    array size
                 SAMPLES
                 *
                 *  5 partitions
                 *
                    0 (array unspecified)        [ERROR 1]
                                                 [ SET error, dont_care]
                                                 [ LABEL error condition ]
                    1 (degenerate array)         [ SET size_1, dont_care ]
                                                 [ LABEL degenerate array]
                    2 to Upper Bound minus 1     [ SET size_ok  ]
                                                 [ LABEL valid]
                    Upper Bound                  [ SET size_ok  ]
                                                 [ LABEL valid upper bound]
                    greater than Upper Bound     [ERROR 2]
                                                 [ SET error, dont_care]
                                                 [ LABEL invalid array size]
                 *
                 MESSAGE 1
                     Array size of 0 is invalid or array size is unspecified.
                     Array size is greater than the Upper Bound of sizes
                 *
                 NAME    array values
                 SAMPLES
                 *
                 *  5 partitions, 1 don't care
                 *
                    all 0's                      [IF size_ok]
                                                 [SET all_same, dont_care]
                    all the same but not 0       [IF size_ok]
                                                 [SET all_same, dont_care]
                    all negative                 [IF size_ok] [SET not_identical]
                                                 [LIMIT 4]
                    all positive                 [IF size_ok] [SET not_identical]
                                                 [LIMIT 4]
                    mixed +/-/0                  [IF size_ok] [SET not_identical]
                    don't care                   [IF !size_ok]
                 *
                 NAME     value arrangement
                 SAMPLES
                 *
                 *     2 partitions, 1 don't care
                 *
                      minimum before max   [IF   size_ok, not_identical]
                      maximum before min   [IF   size_ok, not_identical]
                      don't care           [IF   !not_identical]
                 *
        TYPE     Input-Parameter: Sort Order
                 NAME     sort order
                 SAMPLES
                 *
                 *     3 partitions, 1 don't care
                      ascending               [IF size_ok, not_identical] [ SET ascend]
                      descending              [IF size_ok, not_identical] [ SET descend]
                      unspecified     [ERROR 3][IF size_ok]
                                               [SET error, dont_care]
                      don't care              [IF  dont_care]
                 *
                 MESSAGE
                      Sort order is not specified
                 *
        TYPE     Output to program or change in state
                 NAME    result
                 SAMPLES
                 *     4 partition values
                      error notification         [IF error]
                      array unchanged            [IF dont_care, !not_identical]
                      array in ascending order   [IF ascend, not_identical]
                      array in descending order  [IF descend, not_identical]
                 *
```

Fig. 3. Stage 4 example of a TRL test design.

```
Description:

         Test Representation for SORT requirement.
        File Name:       SORT.TRL
        Version:          1.5  Errors/Messages/Conditions/Limits/Labels
        Last Modified:   9/4/91
        Modified By:     J. Hops


**********************************
Case #        1
Label:        1.6.3.4.1

PARAMETERS:

Type:   Input-Categories for Parameter: Array

        Category Name: array size
            Partition Value:      0 (array unspecified)
            Partition Label:      error condition
            Iteration number: 1

        Category Name: array values
            Partition Value:      don't care
            Partition Label:      instance value needed to pass error

        Category Name: value arrangement
            Partition Value:      don't care
            Partition Label:      instance value needed to pass error

Type:   Input-Parameter: Sort Order

        Category Name: sort order
            Partition Value:      don't care
            Partition Label:      instance value needed to pass error

Type:   Output to program or change in state

        Category Name: result
            Partition Value:      error notification
            Partition Label:      instance value needed to pass error

Error #1: Array size of 0 is invalid or array size is unspecified.


**********************************
Case #        2
Label:        2.6.3.4.2

PARAMETERS:

Type:   Input-Categories for Parameter: Array
        Category Name: array size
            Partition Value:      1 (degenerate array)
            Partition Label:      degenerate array

        Category Name: array values
            Partition Value:      don't care
            Partition Label:      valid

        Category Name: value arrangement
            Partition Value:      don't care
            Partition Label:      valid

Type:   Input-Parameter: Sort Order

        Category Name: sort order
            Partition Value:      don't care
            Partition Label:      valid

Type:   Output to program or change in state

        Category Name: result
            Partition Value:      array unchanged
            Partition Label:      valid

No error conditions exist.


**********************************
```

Fig. 4. Test case results of a stage 4 example of a TRL test design.