Understanding the Logic of System Testing

Dr. Yuri Chernak
Valley Forge Consulting, Inc.

What do system testing and mathematics have in common? They both deal with proofs. This article discusses the logic of system testing, and the steps to construct valid proofs that testers need to perform for their conclusions about the quality of a software product.

Test case is a fundamental term used by all software testers. Numerous published sources, including the Institute of Electrical and Electronics Engineers (IEEE) Standard 610 (IEEE Std. 610), define what a test case is, techniques to design test cases, and templates to document them. However, testers in the field still find these definitions confusing, and they frequently mean different things when referring to test cases.

A common misunderstanding of test cases can be a symptom of a larger issue – a misunderstanding of the logic of software testing. The main purpose of software testing can be defined as exploring the software product to derive and report valid conclusions about its quality and suitability for use. In this regard, software testing is similar to mathematics – they both need proofs for their conclusions. However, mathematicians surpass software testers in deriving and proving their conclusions thanks to their skill in using a powerful tool called *deductive reasoning*. To construct valid arguments, logicians have developed proof strategies and techniques based on the concepts of symbolic logic and proof theory [1, 2, 3].

On critical software projects, testers have always been required to present valid evidence supporting their conclusions about the product's quality. The recent Sarbanes-Oxley Act[1] makes this requirement much more important going forward. In this article, I discuss the logic of one of the conventional levels of testing – system test [4] – and propose a formal approach to constructing valid arguments supporting testers' conclusions. Finally, understanding the system test logic can help testers better understand the meaning of test cases.

Proofs and Software Testing

Software testers have always dealt with proofs on their projects. One example can be concluding that a system passed testing. As testers can never prove the absence of bugs in a software product, their claim that a system passed testing is conditional upon the evidence and arguments supporting such a claim. On critical projects, either the project's manager, end-users, or a compliance department commonly require documented test cases and test execution logs to be used as grounds for supporting testers' conclusion that a software product passed testing.

Another example is reporting a system failure. Regardless whether it is formal testing or unscripted exploratory testing, testers are required to document and report the defects they find. By reporting a defect, a tester first claims that a certain system feature failed testing and, second, presents an argument in the form of a defect description to support the claim. Such an argument should be logically valid to be sufficiently convincing for developers.

Deriving conclusions and presenting valid proofs, also known in mathematics as logical arguments, is frequently not a trivial matter. That is why mathematicians use *deductive reasoning* as a foundation for their strategies and techniques to derive conclusions and present valid arguments. Deductive reasoning is the type of reasoning used when deriving a conclusion from other sentences that are accepted as true [3]. As I discuss in this article, software testers can also benefit from using deductive reasoning. First, they can better understand the logic of software testing and, second, they can construct valid proofs better supporting their conclusions about product quality.

Applying Deductive Reasoning to Software Testing

In mathematics, the process of deriving a conclusion results in presenting a deductive argument or proof that is defined as a convincing argument that starts from the premises and logically deduces the desired conclusion. The proof theory discusses various logical patterns for deriving conclusions, called *rules of inference*, that are used as a basis for constructing valid arguments [2, 3]. An argument is said to be valid if the conclusion necessarily follows from its premise. Hence, an argument consists of two parts – a conclusion, and premises offered in its support. Premises, in turn, are composed of an implication and evidence. Implications are usually presented as conditional propositions, for example, (*if A, then B*). They serve as a bridge between the conclusion and the evidence from which the conclusion is derived [1]. Thus, the implication is very important for constructing a logical argument as it sets the argument's structure and meaning. In the following, I will apply this concept to software testing and identify the argument components that can be used in testing to construct valid proofs.

In software testing, we derive and report conclusions about the quality of a product under test. In particular, in system testing a common unit of a tester's work is testing a *software feature* (also known in Rational Unified Process as *test requirement*); the objective is to derive a conclusion about the feature's testing status. Hence, the feature status, commonly captured as *pass* or *fail*, can be considered a conclusion of the logical argument. To derive such a conclusion, test cases are designed and executed. By executing test cases, information is gained, i.e., evidence is acquired that will support the conclusion. To derive a valid conclusion, also needed are implications that in system testing are known as a feature's pass/fail criteria. Finally, both the feature's pass/fail criteria and the test case execution results are the premises from which a tester derives a conclusion. The lack of understanding of how system testing logic works can lead to various issues – some of the most common of which I will discuss next.

Common Issues With Testing Logic

Issue 1: Disagreeing About the Meaning of Test Cases

Software testers frequently disagree about the meaning of *test cases*. Many testers would define a test case as the whole set

of information designed for testing the same software feature and presented as a test-case specification. Their argument is that all test inputs and expected results are designed for the same objective, i.e., testing the same feature, and they all are used as supporting evidence that the feature passed testing.

For other testers, a *test case* consists of each pair – input and its expected result – in the same test-case specification. In their view, such a test-case specification presents a set of test cases. To support their point, they refer to various textbooks on test design, for example [4, 5], that teach how to design test cases for boundary conditions, valid and invalid domains, and so on. Commonly, these textbooks focus their discussion on designing test cases that can be effective in finding bugs. Therefore, they call each pair of test input and its expected output a test case because, assuming a bug is in the code, such a pair provides sufficient information to find the bug and conclude that the feature failed testing.

Despite these different views, both groups actually imply the same meaning of the term *test case*: information that provides grounds for deriving a conclusion about the feature's testing status. However, there is an important difference: The first group calls *test case* the information that supports the feature's pass status, while the second group calls *test case* the information that supports the feature's fail status. Such confusion apparently stems from the fact that all known definitions of the term test case do not relate it to a feature's pass/fail criteria. However, as the discussion in the sidebar shows, these criteria are key to understanding the meaning of test cases.

Issue 2: Presenting an Argument Without a Conclusion

This issue is also very common. As discussed earlier, an important part of a logical argument is its conclusion. However, a lack of understanding of this concept can lead to presenting arguments without conclusions. On a number of projects, I have seen testers produce test case documentation in the form of huge tables or Excel spreadsheets listing their test cases. In such tables, each row shows a test case represented by a few columns such as test case number, test input, expected result, and test case execution (pass/fail) status. What is missing in this documentation is a description of what features testers intend to evaluate using these test cases. As a result, it is difficult to judge the validity and verify the completeness of such test cases as the underlying purpose for which they were designed is not known. Such documentation suggests that the testers who designed it do not completely understand the logic of software testing.

Issue 3: Presenting an Argument Without an Implication

This issue also stems from a lack of understanding of the structure of a logical argument, specifically that having an implication is necessary for deriving a valid conclusion. In software testing, such implications are a feature's pass/fail criteria. The issue arises when such criteria are either forgotten or not clearly defined and understood by testers. This can lead to a situation where testers lose sight of what kind of conclusions they need to report. As a result, instead of deriving a conclusion about the feature and then reporting its testing status, they report the status of each executed test case. This situation presents an issue illustrated in the following example.

Let us assume a tester needs to test 10 software features, and he or she designed 10 test cases for each of the features under test. Thus, the entire testing requires executing 100 test cases. Now, while executing test cases, the tester found that one test case failed for each of the features. In our example, the tester did not define and did not think about the feature pass/fail criteria. Instead, the tester reported to a project manager the testing status for each executed test case. Thus, at the end of the testing cycle, the results show that 90 percent of testing was successful. When seeing such results, a manager would be fairly satisfied and could even make a decision about releasing the system.

The project manager would see a completely different picture if the features' pass/fail criteria were not forgotten. In this case, the testers would report the testing status for each feature as

What Do We Call a Test Case?

Most of the published sources defining the term *test case* follow the definitions given in the Institute of Electrical and Electronics Engineers (IEEE) Standard 610 (IEEE Std. 610):

a) **Test Case:** A set of test inputs, execution conditions, and expected results developed for a particular objective such as to exercise a particular program path or to verify compliance with a specific requirement.
b) **Test Case:** Documentation specifying inputs, predicted results, and a set of execution conditions for a test item.

Despite the fact that this standard was published many years ago, testers in the field still do not have a consistent understanding of the meaning of test cases. To better understand this meaning, we can use the concept of deductive reasoning. Following this concept, system testing can be viewed as a process of deducing valid conclusions about the testing status of system features based on evidence acquired by executing test cases. Hence, the main purpose of executing test cases is to gain information about the system implementation. This information can be used together with the feature's pass/fail criteria to derive and support conclusions about the status of feature testing. The feature's pass/fail criteria are important implications in the testing argument that determine the meaning of testers' conclusions. These criteria are a link between a tester's conclusion about the feature status and the test cases used to support the conclusion. *Hence, the meaning of test cases follows from the definition of the feature's pass/fail criteria.*

In system testing, the mission is finding and reporting software defects; the feature fail criterion is commonly defined as, "If any of the feature's test cases fails, then the feature fails testing." What follows from this implication is that a test case is information that is sufficient to identify a software defect by causing a system feature to fail. The feature's pass criterion can further explain the meaning of test cases. It is commonly defined as, "The feature passes the test only if all of its test cases pass testing." According to this definition, we imply that the system feature passed testing only if the whole group of its test cases passed testing. Hence, test cases are used as collective evidence to support the feature's pass status. It should be noted, however, that this interpretation of the test-case meaning refers to the system test only. In contrast, in acceptance testing a testing mission and pass/fail criteria can be defined differently from the system testing. Correspondingly, the meaning of test cases can be different as well.

If the IEEE definitions of test cases are examined again, we can see that these definitions are not specific to a particular testing mission, nor are they explicit about which testing conclusion, i.e., pass or fail, a test case is intended to support. Instead, they focus primarily on the test case structure: test inputs and expected results. As a result, these definitions alone and without the feature's pass/fail criteria lack clarity about the test case purpose and meaning.

opposed to each test case. If the feature fail criterion were defined as, "If any of the feature's test cases fails, then the feature fails testing," then the testing end-result in our example would have been quite the opposite and have shown that none of the software features passed testing; they all should be re-tested when the bugs are fixed.

An Approach to Constructing A Valid Proof

Constructing a valid proof in system testing can be defined as a four-step procedure. The following sections discuss each step in detail, and explain how to construct a valid argument to support a testing conclusion.

Step 1: Define a Conclusion of the Argument

In constructing a proof, always begin by defining what needs to be proven, i.e., the conclusion. In system testing, the ultimate goal is to evaluate a software product. This is achieved by decomposing the entire functional domain into a set of functional features where each feature to be tested is a unit of a tester's work that results in one of the two possible conclusions about a feature's testing status *pass* or *fail*. At any given time, only one of the two conclusions is valid.

The term *software feature* is defined in the IEEE Std. 610 as follows:

a) A distinguished characteristic of a software item.
b) A software characteristic specified or implemented by requirements documentation.

From a tester's perspective, a software feature means any characteristic of a software product that the tester believes might not work as expected and, therefore, should be tested. Deciding what features should be in the scope of testing is done at the test-planning phase and documented in a test plan document. Later, at the test design phase, the list of features is refined and enhanced based on a better understanding of the product's functionality and its quality risks. At this phase, each feature and its testing logic are described in more detail. This information is presented either in test design specifications and/or in test case specifications.

The test design specification commonly covers a set of related features; whereas, the test case specification commonly addresses testing of a single feature. At this point, a tester should already know which quality risks to focus on in feature testing. Understanding the feature's quality risks, i.e., how the feature can fail, is important for designing effective test cases that a tester executes to evaluate the feature's implementation and derive a conclusion about its testing status. Performing Step 1 can help testers avoid Issue No. 2 as discussed earlier.

Step 2: Define an Implication of the Argument

The next important step is to define an implication of an argument. An implication of a logical argument defines an important relation between the conclusion and the premises given in its support. Correspondingly, the feature's pass/fail criteria define the relation between the results of test-case execution and the conclusion about the feature's evaluation status.

According to the IEEE Std. 829, the feature's pass/fail criteria should be defined in the test design specification; this standard provides an example of such a specification. However, it does not provide any guidance on how to define these criteria, apparently assuming this being an obvious matter that testers know how to handle. Neither do the textbooks on software testing methodology and test design techniques. Contrary to this view, I feel that defining these criteria is one of the critical steps in test design that deserves a special consideration. As I discussed earlier and illustrated as Issue No. 3, the lack of understanding of the role and meaning of the feature's pass/fail criteria can lead to logically invalid testing conclusions in system testing. Also, as discussed in the sidebar, from the well-defined implications, i.e., the features' pass/fail criteria, testers can better understand the meaning of test cases and avoid the confusion discussed earlier as Issue No. 1.

Table 1: *Deriving a Feature Fail Conclusion*

Modus Ponens Form	Testing Argument Form
1. If A, then B – *means* ⟶	1. If any test case fails, then a feature fails (*implication*).
2. A is true – *means* ⟶	2. We know that at least one test case failed (*evidence*).
3. Then B is true – *means* ⟶	3. Then the feature fails the test (*conclusion*).

The rationale for defining the feature's pass/fail criteria stems from the system test mission that can be defined as *critically examining the software system under the full range of use to expose defects and to report conditions under which the system is not compliant with its requirements*. As such, the system test mission is driven by the assumption that a system is not yet stable and has bugs; the testers' job is to identify conditions where the system fails. Hence, our primary goal in system testing is to prove that a feature fails the test. To do that, testers develop ideas about how the feature can fail. Then, based on these ideas, testers design various test cases for a feature and execute them to expose defects in the feature implementation. If this happens, each test case failure provides sufficient grounds to conclude that the feature failed testing. Based on this logic, the feature's fail criterion can be defined as, "If any of the feature's test cases fail, then the feature fails testing." In logic, this is known as a sufficient condition (*if A, then B*). The validity of this implication can also be formally proved using the truth-table technique [1]; however, this goes beyond the scope of this article.

Defining the feature's pass criterion is a separate task. In system testing, testers can never prove that a system has no bugs, nor can they test the system forever. However, at some point and under certain conditions they have to make a claim that a feature passed testing. Hence, the supporting evidence, i.e., the test case execution results, can only be a necessary (*C, only if D*), but not a sufficient condition of the feature's pass status. Based on this logic, the feature's pass criterion can be defined as, "The feature passes the test only if all of its test cases pass testing." In this case, the feature's pass criterion means two things:

a) The feature pass conclusion is conditional upon the test execution results presented in its support.
b) Another condition may exist that could cause the feature to fail.

Step 3: Select a Technique to Derive a Conclusion

Once we have defined all components of a testing argument, the next step is to select a technique that can be used to derive a valid conclusion from the premises. The word *valid* is very important at this point as we are concerned with deducing the conclusion that logically follows from its premises. In the proof theory, such techniques are known as *rules of inference* [1, 2]. By using these rules, a valid argument can be constructed and its conclu-

sion deduced through a sequence of statements where each of them is known to be true and valid. In system testing, there are two types of conclusions – a feature fail status and a feature pass status. Correspondingly, for each of these conclusions, a technique to construct a valid argument is discussed. On software projects, testers should discuss and define the logic of constructing valid proofs before they begin their test design. For example, they can present this logic in the *Test Approach* section of a test plan document.

Deriving a Feature Fail Conclusion

I defined the feature's fail criterion as a conditional proposition in the form (*if A, then B)*, which means if any of the test cases fail, then testers can conclude that the feature fails as well. This also means that each failed test case can provide sufficient evidence for the conclusion. In this case, a valid argument can be presented based on the rule of inference, known as *Modus Ponens* [1, 2]. This rule is defined as a sequence of three statements (see Table 1). The first two statements are premises known to be true and lead to the third statement, which is a valid conclusion.

Deriving a Feature Pass Conclusion

The feature's pass criterion was defined as a conditional proposition in the form (*C, only if D)*, which means a feature passes the test only if all of its test cases pass testing. This also means that such a conclusion is derived only when all of the feature's test cases have been executed. At this point, the feature status can be either pass or fail, but not anything else. Hence, the rule of inference can be used, known as *Disjunctive Syllogism* [1, 2], which is presented as three consecutive statements that comprise a valid argument (see Table 2).

Step 4: Present an Argument for a Conclusion

At this point, there is a clear plan on how to construct valid arguments in system testing. The actual process of deriving a testing conclusion begins with executing test cases. By executing test cases, the testers can learn the system's behavior and analyze the feature implementation by comparing it to its requirements captured by expected results of test cases. As a result, testers can acquire evidence from which they can derive and report a valid testing conclusion, i.e., a feature pass or fail testing status.

Concluding a Feature Fail Status

The feature fail criterion is defined as, "If any of the test cases fails, then the feature fails testing." According to the Modus Ponens rule, this means that each failed test case provides grounds for the valid conclusion that the feature has failed testing. As a feature can fail on more than one of its test cases, after finding the first defect a tester should continue feature testing and execute all of its test cases. After that, the tester should report all instances of the feature failure by submitting defect reports, where each defect report should be a valid argument that includes the evidence supporting the feature fail status.

Disjunctive Syllogism Form	Testing Argument Form
1. Either P or Q is true – *means* ⟶	1. After all test cases have been executed, a feature status can be either fail (P) or pass (Q) (*implication*).
2. P is not true – *means* ⟶	2. We know that the feature did not fail the test for all of its test cases (*evidence*).
3. Then Q is true – *means* ⟶	3. Then the feature passes the test (*conclusion*).

Table 2: *Deriving a Feature Pass Conclusion*

On the other hand, if a given test case passed testing, the Modus Ponens rule does not apply, and there are no grounds for any conclusion at this point, i.e., the feature has neither passed nor failed testing. Finally, only when all of the feature's test cases have been executed should it be decided whether there are grounds for the feature pass status as discussed in the next section.

Concluding a Feature Pass Status

Obviously, if the feature has already failed, the pass status cannot have grounds. However, if none of the test cases failed, then the Disjunctive Syllogism rule can be applied. According to this rule, the fact that none of the test cases failed provides grounds for a valid conclusion: the feature passed testing. To support this claim, evidence is provided – test case execution results. However, this conclusion should not be confused with the claim that the feature implementation has no bugs, which we know is impossible to prove. The conclusion means only that the feature did not fail on the executed test cases that were presented as evidence supporting the conclusion.◆

References

1. Copi, I., and C. Cohen. Introduction to Logic. 11th ed. Prentice-Hall, 2002.
2. Bloch, E. Proof and Fundamentals. Boston: Birkhauser, 2000.
3. Rodgers, N. Learning to Reason. An Introduction to Logic, Sets, and Relations. John Willey & Sons, 2000.
4. Meyers, G. The Art of Software Testing. John Wiley & Sons, 1979.
5. Kit, E. Software Testing In the Real World. Addison-Wesley, 1995.

Note

1. The Sarbanes-Oxley Act <http://news.findlaw.com/hdocs/docs/gwbush/sarbanesoxley072302.pdf>.

Acknowledgements

I am grateful to the CROSSTALK reviewers, to the distinguished professor Sergei Artemov at the graduate center of the City University of New York, and to Robin Goldsmith at GoPro Management for their feedback and comments that helped me improve this article.

About the Author

Yuri Chernak, Ph.D., is the president and principal consultant of Valley Forge Consulting, Inc. As a consultant, Chernak has worked for a number of major financial firms in New York helping senior management improve the software testing process. Currently, his research focuses on aspect-oriented requirements engineering, use-case-driven testing, and test process assessment and improvement. Chernak is a member of the Institute of Electrical and Electronics Engineers (IEEE) Computer Society. He has been a speaker at several international conferences, and has published papers on software testing in IEEE publications and other professional journals. Chernak has a doctorate in computer science.

Valley Forge Consulting, Inc.
233 Cambridge Oaks ST
Park Ridge, NJ 07656
Phone: (201) 307-4802
E-mail: ychernak@yahoo.com