# A Review of Boundary Value Analysis Techniques

Dr. David J. Coe
*The University of Alabama in Huntsville*

*Software testing is an essential element of any software development effort. Developers must have some means of selecting tests to evaluate the completeness and quality of product produced. This article reviews Boundary Value Analysis (BVA), a functional testing methodology that can assist in the identification of an effective set of tests.*

Software testing is a fundamental software engineering activity critical to a successful development effort. In fact, an increasingly popular approach to software development is that of *test-driven development* in which tests are identified and documented prior to implementation of the code [1]. The test-driven approach to development places an emphasis on the *quality* of the resulting product by establishing completeness and correctness criteria early. A major challenge to any testing effort is that one must identify a set of tests that are effective at finding defects while keeping the resources associated with applying those tests within project cost and schedule constraints.

The following is an overview of BVA, a systematic methodology for identifying tests to apply. In the following discussion, the term *test case* refers to "a set of 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." A *test* is defined as either "a set of one or more test cases" or as "the execution of the test cases." A *fault* is "an incorrect step, process, or data definition in a computer program," and a *failure* is the "inability of a system or component to perform its required functions within specified performance requirements [2]." Thus, a primary goal of software testing is to identify failures, which indicate the presence of one or more faults [3].

## Overview of BVA

BVA is a black-box approach to identifying test cases. In black-box testing, test cases are selected based upon the desired product functionality as documented in the specifications without consideration of the actual internal structure of the program logic [4]. A fundamental assumption in BVA is that the majority of program errors will occur at critical input (or output) boundaries, places where the mechanics of a calculation or data manipulation must change in order for the program to produce a correct result [3].

An example that illustrates the general concept of boundary values would be a program that calculates income tax for a given income. In a progressive income tax scheme, the tax rate applied increases from low-income brackets to high-income brackets. In this case, the critical input boundaries would be the set of incomes at which the applied tax rate should change along with any minimum or maximum extremes of the income value. Thus, the set of boundary incomes defines the limits of each tax bracket.

## Test Case Selection Using BVA

The set of test cases identified by BVA depends upon both the reliability requirements of the software under test and the underlying assumptions on the likelihood of single versus multiple range checking faults. The following discussions of *single-variable* and *multi-variable BVA* are derived from the BVA taxonomy and discussion presented in [3].

### *Single-Variable BVA*

The baseline procedure for BVA begins by identifying the boundary values, typically from the input point of view. All of these boundary values will be incorporated into the set of test cases. In addition to those values, values near the boundaries will be tested. These boundary-adjacent values will help to exercise the program's bounds-checking logic. For example, when testing the range of a value in a branching or looping statement, the developer may use a less-than operator, '<,' when the correct operator should have been a less-than-or-equal to operator, '<=,' or a greater-than operator, '>,' which is adjacent to the less-than operator on most keyboard layouts. Such errors would

Figure 1: *Baseline BVA Test Cases Identified for Single-Variable, Single-Range Example (Shaded area indicates valid values of the variable N)*

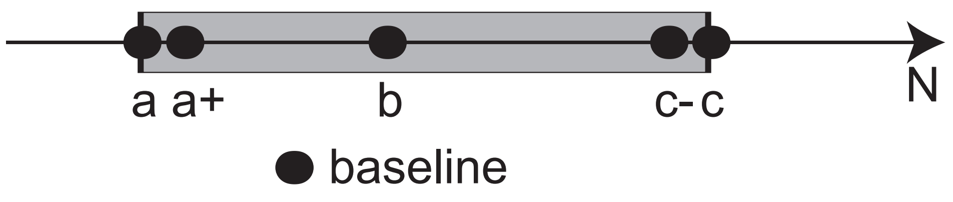


Figure 2: *Single-Variable, Single-Range Baseline Test Cases Augmented With Robustness Tests (shaded area indicates valid values of the variable N)*

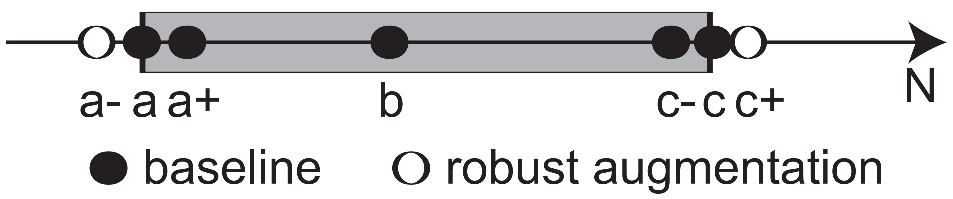
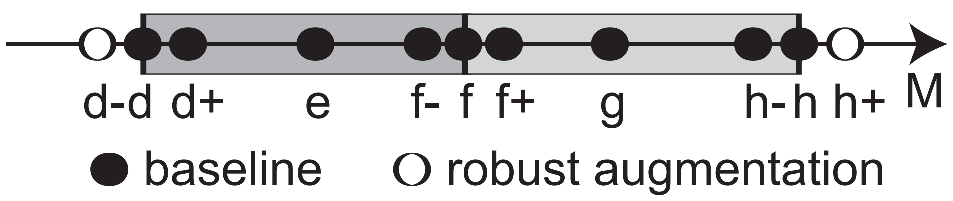


Figure 3: *Single-Variable, Two-Range Test Cases Identified by Robust BVA (Highlighted areas indicate the two subranges of valid values of M)*



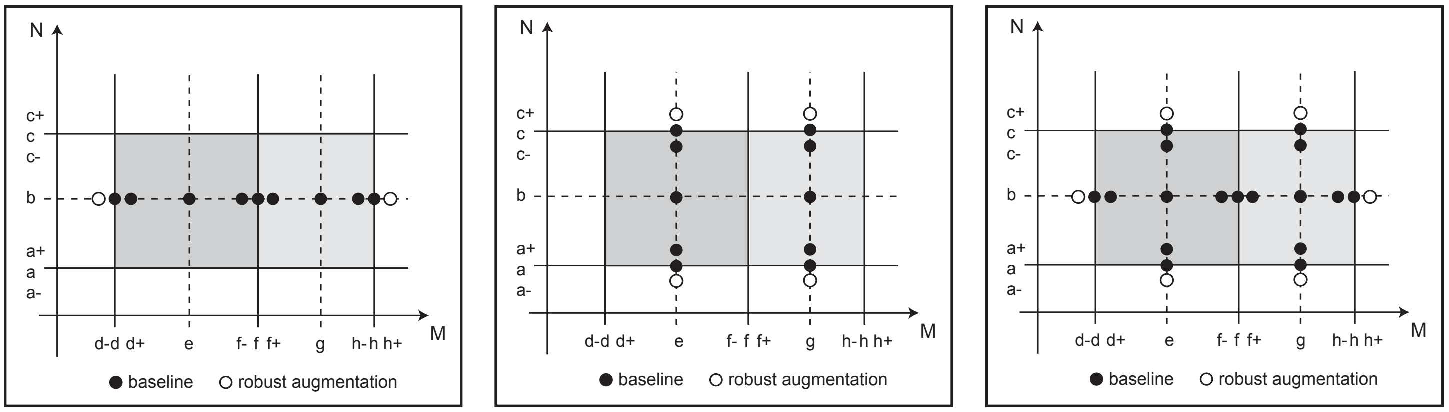


Figure 4 A,B,C: *Single-Fault, Baseline, and Robust Test Cases (A) assuming N is at its nominal value, (B) assuming M is at its nominal values for each subrange, and (C) the set of all test cases identified (derived from* [3]*)*

result in code that compiles but executes incorrectly under certain conditions. To test for these types of errors, values adjacent to the boundary values must be included in the set of test cases. In addition to the boundary and boundary-adjacent values, the baseline BVA procedure includes some nominal value of input (or output) in the set of test cases. The baseline BVA procedure is best illustrated by the following example.

Consider a program with a single input variable $N$ that has an output defined only for values of $N$ in the range $a \le N \le c$. The set of test cases selected would be at *minimum* the set of values $N_{baseline} = \{a, a+, b, c-, c\}$ where $a+$ is a value just greater than $a$, $c-$ is a value just less than $c$, and the value $b$ is some nominal value that lies between $a+$ and $c-$. In this example, the baseline BVA procedure identifies five test cases. As graphed on the number line in Figure 1 (see page 19), the test cases selected under the baseline procedure do not exceed the allowable range of inputs for the variable $N$.

If error handling is critical to the software under test, then one augments the set of test cases identified by the baseline BVA procedure to include robustness tests, that is, values outside the allowable range. The baseline tests identified above are augmented with the values $\{a-, c+\}$ where $a-$ is a value just below the minimum acceptable value $a$ and $c+$ is a value just above the maximum acceptable value $c$. The inclusion of the values $\{a-, c+\}$ in the set of test cases should force execution of any exception handler or defensive code. In this single-input, single-range example, robust BVA identifies a total of seven test cases as shown in Figure 2 (see previous page) where $N_{robust} = \{a-, a, a+, b, c-, c, c+\}$.

The baseline BVA or robust BVA procedures may also be applied in situations where an input may have multiple subranges. Consider a single input $M$ with two adjacent subranges where range #1 is given by $d \le M < f$ and range #2 is given by $f \le M \le h$. The set of test cases would be the union of the test cases identified by applying the BVA procedure to each individual subrange. So, the *union* of test cases resulting from the application of baseline BVA to each subrange individually is given by the following:

$$\begin{aligned} \mathbf{M_{baseline}} &= \{d, d+, e, f-, f\} \cup \{f, f+, g, h-, h\} \\ &= \{d, d+, e, f-, f, f+, g, h-, h\} \end{aligned}$$

Application of robust BVA augments $M_{baseline}$ with the extreme values $\{d-, h+\}$ to yield $M_{robust} = \{d-, d, d+, e, f-, f, f+, g, h-, h, h+\}$ as illustrated in Figure 3. The addition of multiple subranges clearly increases the total number of test cases identified. For two adjacent subranges of a single variable, baseline BVA identified nine test cases and robust BVA identified 11 test cases total.

## Multi-Variable BVA

The BVA test case selection procedure for multi-variable problems also requires consideration of fault likelihood, what I refer to as a fault model. Under the single-fault model, it is assumed that a failure is the result of a single fault due to the low probability of two or more faults occurring simultaneously [3]. For the multiple-fault model, one assumes that the likelihood of multiple simultaneous faults is no longer insignificant, and thus additional test cases must be selected to address situations such as erroneous range checking on multiple variables simultaneously.

Drawing from our previous single variable examples, assume the single-fault model for a problem that has two inputs, $N$ and $M$, with values of $N$ in the allowable range $a \le N \le c$ and where allowable values of M span range #1, given by $d \le M < f$, and range #2, given by $f \le M \le h$. From our previous discussion, the baseline single variable test cases identified for N and M respectively are the following:

$$\mathbf{N_{baseline}} = \{a, a+, b, c-, c\}$$

and

$$\mathbf{M_{baseline}} = \{d, d+, e, f-, f, f+, g, h-, h\}$$

Under the single-fault assumption, multi-variable BVA test cases are selected that exercise the boundaries of one variable while the other variables are held at a nominal value. The final set of test cases selected is the union of all test cases identified as this procedure is applied to each individual input in turn. In the following example, I have chosen to apply this procedure to each subrange of each variable in turn to produce a symmetric solution.

Since we have assumed that there are two inputs in this problem, the set of test cases will consist of ordered pairs of inputs $(m,n)$ such that $n$ is a member of $N_{baseline}$ and $m$ is a member of $M_{baseline}$. Figure 4A shows a graph of the nine test cases identified assuming that $n$ is held to its nominal value $b$ while $m$ varies across the members of $M_{baseline}$. The graph in Figure 4B shows the 10 test cases identified in which $m$ is held to its nominal value, $e$ or g, while $n$ varies across the members of $N_{baseline}$. Figure 4C illustrates the union of these sets of test cases. Note that due to the selection of $(e,b)$ and $(g,b)$ twice, a total of 17 test cases have been identified instead of 19.

For robustness testing, one applies the same procedure starting with the values previously identified in the sets $M_{robust}$ and $N_{robust}$. Note that under the single-fault model, robustness testing adds only six additional test cases to the 17 baseline test cases for a total of 23 test cases. These additional tests are identified in Figure 4C.

Under the multiple-fault assumption, additional test cases must be selected to detect multiple, simultaneous faults such as erroneous range checking on two variables at the same time. The multiple-fault BVA procedure again starts with the sets $M_{baseline}$ and $N_{baseline}$ if bounds checking is not critical or $M_{robust}$ and $N_{robust}$ if bounds checking is a high priority. To select BVA test cases assuming that multiple simultaneous faults are likely, one computes the Cartesian product $N_{baseline}$ x $M_{baseline}$ for the baseline multiple-fault test cases or $M_{robust}$ x $N_{robust}$ for the multiple-fault, i.e., worst-case test cases [3].

Given two sets M and N, the *Cartesian product* of M and N is defined as follows:

$$M \times N = \{(m,n) \mid m \in M \wedge n \in N\}$$

where (m,n) denotes an ordered pair [5]. In other words, M x N is the set that consists of all possible ordered pairings of an element from set M with an element of set N. So, if set M contains x elements and set N contains y elements, the resulting set M x N will contain a total x*y total elements.

Figure 5 depicts the baseline and robust BVA test cases identified for our sample problem assuming the multiple-fault model. Note the significant increase in the total number of tests identified. Forty-five baseline test cases were identified for this problem plus an additional 32 for worst-case robustness testing.

Table 1 summarizes the number of test cases identified versus various reliability requirements and fault model assumptions. The multiple-fault assumption significantly increases the total number of tests required, especially in situations where a variable of interest has multiple ranges. Under the single-fault assumption, the incorporation of robustness tests, even in the situation where a variable has multiple ranges, results in a modest increase in the total number of test cases required.

## Discussion

From the previous review it is clear that BVA has several advantages: The mechanical nature of the procedure and the symmetry of the tests identified make the BVA procedure easy to remember and use, especially given that critical input boundaries are often already explicitly identified in the requirements. With BVA, one can adjust the number of test cases identified and, thus, the resources expended on testing effort, depending upon the robustness demands of the product.

BVA also serves as an introduction to other test techniques. Discussions of BVA in the literature are often intermingled with a related black-box technique known as Equivalence Partitioning (EP), which utilizes the boundary values in an attempt to define partitions or sets of test cases that are *equivalent* in the sense that all test cases grouped within a particular partition would reveal the presence of the same set of defects and likewise fail to detect other defects. In its simplest form, once the partitions are identified, the set of test cases selected is one representative test case from each partition. A distinct advantage of the EP technique is that the total number of test cases is significantly smaller than the set of test cases identified through BVA. In fact, the set of test cases identified by EP can be a subset of those identified by BVA, and researchers have exploited this fact to reduce the total number of test cases identified in merged BVA-EP schemes.

Studies show, however, that BVA can be effective at identifying failures. In [6], Reid investigated the effectiveness of random testing, equivalence partitioning, and boundary value analysis techniques.

***"With BVA, one can adjust the number of test cases identified and, thus, the resources expended on testing effort ..."***

According to his results, the probability that BVA would detect a fault was more than six times higher than random testing and more than twice as high as equivalence partitioning. The cost for this increased effectiveness was additional test cases, on the order of two to three times the number of test cases as equivalence partitioning depending upon the particular variations of the techniques employed.

Other studies have compared functional, structural, and code reading test methodologies. In structural testing, test cases are selected to exercise specific program elements such as statements, branches, or paths through a code segment. For example, to achieve 100 percent statement coverage, the set of test cases identified must force execution of each program statement at least once. For 100 percent branch coverage, the set of test cases identified must force each branch option to execute at least once. For code reading, individuals were given the source code and asked to work backwards towards a specification for that program by successively grouping subprograms into logical modules until an understanding of the overall functionality was achieved. Failures were detected by comparing the actual specification to that derived by the code reader.

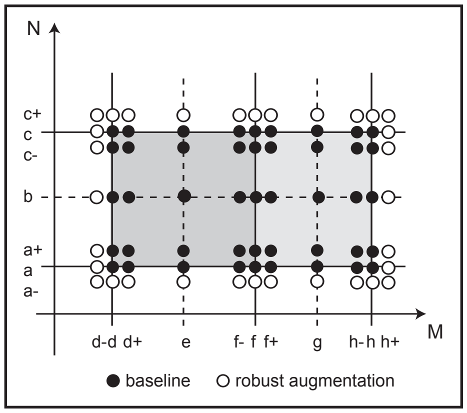


Figure 5: *Multiple-Fault, Baseline, and Robust Tests (derived from* [3]*)*

Basili and Selby [7] studied the relative effectiveness of a combined BVA and EP functional testing approach against 100% statement coverage structural testing and code reading. Among professional programmers, they found that code reading detected the most faults followed by functional testing and then structural testing. The average maximum statement coverage achieved by both the functional and structural testers was 97 percent yet the functional testing approach detected more faults than did structural testing in this study. It was also noted that the number of faults detected varied with the type of software tested, and that the testing techniques tended to detect different types of faults.

The relative effectiveness of a combined BVA and EP functional testing technique, 100% branch coverage structural testing, and code reading were compared in [8]. This study also observed that the effectiveness of the techniques varied with both the nature of the programs and of the faults themselves. Most importantly, this study determined that the use of two or more test techniques together, such as functional testing and code reading, was more effective in general than any single methodology since the techniques were

Table 1: *Number of Test Cases Identified for Two-Variable BVA Problem*

| Number of Tests Identified | | Assumed Fault Model | |
|---|---|---|---|
| | | Single-Fault | Multiple-Fault |
| Reliability Requirement | Baseline | 17 | 45 |
| | Robust | 23 | 77 |

essentially complementary [8].

## Conclusions

The BVA technique provides a systematic procedure for evaluating the completeness and quality of a software product. While some may find excessive redundancy in the set of test cases generated by boundary value analysis, I have found that the symmetry and mechanical nature of BVA help to make the procedure both easier to teach at the undergraduate level and easy to remember and apply in practice. BVA also provides a basis for learning other techniques, in particular, equivalence partitioning, and it is effective as a functional testing technique for identifying failures. Empirical studies show, however, that a combination of functional, structural, and/or code reading techniques is generally more effective than relying upon any single methodology since the effectiveness of the techniques vary with both the type of code being tested and the nature of the faults.◆

## References

1. Schach, Stephen R. Object-Oriented and Classical Software Engineering. 7th ed., McGraw Hill, 2007.
2. IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard 610.12-1990.
3. Jorgensen, Paul C. Software Testing: A Craftman's Approach. 2nd ed. CRC Press, 2002.
4. Perry, William E. Effective Methods for Software Testing. 3rd ed. Wiley Publishing, 2006.
5. Beyer, William H. CRC Standard Mathematical Tables. 25th ed. CRC Press, 1981.
6. Reid, S.C. 1997. An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing. Proc. of the 4th International Symposium on Software Metrics 05-07 Nov. 1997, Washington, D.C.: IEEE Computer Society, 1997.
7. Basili, Victor R., and Richard W Selby. "Comparing the Effectiveness of Software Testing Strategies." IEEE Trans. on Software Engineering 13.12 (1987): 1278-1296.
8. Wood, M., M. Roper, A. Brooks, and J. Miller. Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study. ACM SIGSOFT Software Engineering Notes, Proc. of the 6th European conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering ESEC '97/FSE-5, 22.6 (1997): 262-277.

## About the Author

**David J. Coe, Ph.D.,** is an assistant professor in the department of electrical and computer engineering at the University of Alabama in Huntsville where he teaches undergraduate and graduate courses in C++ programming, data structures, and software engineering. He has consulted locally in the areas of software engineering and software process. Coe has an undergraduate degree in computer science from Duke University, and a master of science degree in electrical engineering and doctorate degree in electrical engineering from the Georgia Institute of Technology.

**The University of Alabama in Huntsville**
**Department of Electrical and Computer Engineering**
**217-F Engineering BLDG**
**Huntsville, AL 35899**
**Phone: (256) 824-3583**
**E-mail: coe@ece.uah.edu**