
ISLANDIA

How to Avoid Letting The Unknown Burn Your Software Project

Rout Out the Dangers Lurking in Program Boundaries With Exploratory Testing

By Rob Sabourin

On ancient navigation charts, dangerous or unexplored territories were often indicated with images of mythical creatures and warnings to mariners. The Lenox Globe from the early 1500s, the oldest known terrestrial globe, used the Latin phrase "HC SVNT DRACONES (hic sunt dracones, "here are dragons") on the east coast of Asia. But the true dangers were not mythical creatures, but some very real concerns about the risks of the unknown.

In many software testing projects, I've been challenged to discover a product's true boundaries. As explained in last month's installment, "On the Field of Finite Boundaries," a good many bugs can be found at the edges of your applications. Boundaries exist in all variables and variable combinations, which influence the behavior of the software we test.

Rob Sabourin is president of software consultancy Amibug.com.

Why look for boundaries? Their locations and sources can help us to gain confidence in the behavior of software, as well as help us focus testing. In my experience, whenever I identify a boundary in a variable, I can immediately define important tests of the application's behavior by attempting to process values on either side of the boundary as well as on and around the boundary.

The Four Steps

I vary my approach to finding an application's boundaries depending on the blend of the project's technical and business contexts.

Sometimes I have access to requirements; sometimes I have access to designs, and on occasion, I can even chat with developers. I've even had the good fortune of accessing database schemas and source code to help me find boundaries.

While uncovering bugs as part of the process is certainly a useful side effect, understanding boundaries can also help us to define related tests and discover strategies to access correctness. Understanding boundaries also gives us key points at which to validate the behavior and requirements of the software being tested.

The four basic steps for identifying boundaries are:

- Identify the testing objective
- Find variables related to the testing objective
- Determine which variables contribute to the processing or behavior of the application being tested
- Use experimentation and analysis to isolate boundary values of contributing variables

Identify Testing Objectives

When I test software, I don't begin by looking for boundaries. Instead, I make sure that I'm finding boundaries related to a specific testing objective. A testing objective could be a goal from a test plan, confirmation of a software require-

ment, a charter of exploratory testing, or confirmation of a usage scenario or a system failure mode.

Focus on one test objective at a time. Try to identify boundaries of the application to help fulfill the test objective. Test objectives concerned with software behavior invariably involve finding boundaries.

Find Variables

Variables in requirement-based testing. Requirement-based test objectives can be related to explicit, documented requirements; implicit, undocumented requirements; product constraints; product environments or statements of business rules. Boundaries can be found based on the requirement or the implementation of the requirement.

Requirement-based test objectives are used to confirm that the software conforms to the requirements. *Variables* are generally the parameters, options and conditions that influence the application's behavior in fulfilling the requirement.

All variables related to the requirement are considered. In addition, it's critical at times to identify variables not explicitly mentioned in the requirement document, but which would affect the behavior of the software related to the requirement. For example, a report-generation requirement may describe the data and structure required on a month-end report. Report generation is also influenced by the printer page setup, margins and paper type. These variables influence the implementation of the capability, but aren't explicitly referenced in the requirement.

Variables in exploratory testing. When implementing exploratory testing, I generally divide the project into a series of testing charters, each designed to concurrently explicate and test the product. Charters help to focus testing.

In exploratory testing, boundaries related to the charter are discovered. Sometimes I even define the charter of an exploratory testing session to explicitly discover the variables and associated boundaries related to some feature or characteristic of the application being tested.

Discovering and identifying the variables depends on the tester's experience, subject-matter expertise and observation skills. Some variables, especially data entered on menus, dialogs and user interface controls, are obvious. Variables related to the system under test are also relatively straightforward to isolate. But the elusive ones are those that influence the behavior of the system but aren't readily visible to the tester.

Try to identify variables that the requirements may not mention, but which affect the related software.

Experimental approaches can be used to confirm hypotheses about the nature of possible variables related to the testing objective. For example, in an insurance application. many attributes, including age, can define a customer, some of them influencing the rate computation. Experiments can be defined to confirm the hypothesis that age influences the rate computation.

First, set all attributes to fixed values, and then compute rate several times varying the age value. In each trial, vary the age without changing other attributes. If the rate varies, then you've confirmed the hypothesis that age is a variable in the rate computation. The opposite is not true—an invariant rate tells us nothing about the relationship between rate computations and age.

The principle of concomitant variations, conceived by 19th century English philosopher John Stuart Mill, can be used to discover variables that relate to the testing charter (see "Mill's Methods" sidebar).

Variables in usage scenarios. When testing a usage scenario, I'm concerned with the user's ability to perform a task using the software. Usage scenarios are defined in terms of the work the user must do to perform the task—not on the capabilities of the software under test.

In usage scenarios, I generally walk through the scenario from beginning to end, once for each possible alternate path. For example, to test the purchase of a book at Amazon.com, I check paths for each alternate flow by varying the type of books purchased and the payment terms. In the walkthrough, I take note of every time a user enters data or makes a choice. Each instance noted is considered a variable, and only those variables related to the scenario qualify. If there are fields of dialogs and menu items the user hasn't employed to fulfill the scenario, they aren't considered relevant in my hunt for boundaries.

Variables in failure-mode testing. In failure-mode testing, we study the behavior of the system in response to invalid data, unexpected conditions and a harshly constrained environment. Variables are chosen in advance, and testing observes and analyzes the application's behavior as these variables are controlled in an invalid way. Generally, failure-mode testing offers an indication of the robustness or fault tolerance of the system under test. Often, the failure-mode test aims to discover at what value a variable will cause system behavior to degenerate.

Determine Contributing Variables

Variables can be classified into two broad categories: contributing variables are those that influence the behavior of the application, and independent variables are those that don't. Although all vari-

ILL'S METHODS

From www.wikipedia.org:
"Whatever phenomenon varies in any manner whenever another phenomenon varies in some particular manner, is either a cause or an effect of that phenomenon, or is connected with it through some fact of causation."

If, across a range of circumstances leading to a phenomenon, some property of the phenomenon varies in tandem with some factor existing in the circumstances, the phenomenon can be attributed to that factor. For instance, suppose that various samples of water, each containing both salt and lead, were found to be toxic. If the level of toxicity varied in tandem with the level of lead, one could attribute the toxicity to the presence of lead.

ables influence the behavior of the software being tested in some way, I'm concerned with variables that add optimal value to the process.

If I were testing the capabilities of a clothes dryer, I'd identify several variables: material type, temperature, weight, humidity, speed, color and the light switch setting. To focus on the variables more directly influencing the ability to dry clothing, I'd probably define material color and light switch settings as independent variables.

I'd be tempted to consider material type as independent as well, but I'd probably consult with subject-matter experts first. A textile specialist could help me better understand the relationship among materials, temperature and humidity in the drying process. Variables of temperature, weight, humidity and speed would certainly be contributing variables.

Finding Boundaries

Now that we've identified the relevant variables, it's time to roll up our sleeves and discover the boundaries. I've used many different approaches, but I've found that the most effective are black-, white- and gray-box testing.

Black-box testing is testing an application from the outside. We generally lack specific knowledge of the design or implementation of the software. We control the application by varying the environment and controlling external parameters. And we assess correctness by observing the results, outputs and outcomes of the processing done by the application.

Very often, black-box approaches are the only ones readily available to testers and should be considered key skills. If I have requirement documents available, I research the possible ranges of values for business logic or data processing. Each range of values defines boundaries. I then confirm that these boundaries really existed in the application by trying values above, below and immediately on each value.

Next, I make sure that I confirm processing; that persistent stored data and reports are consistent for each variable. In my experience, boundary bugs are often due to inconsistent development or design for different aspects of the application related to the variable.

To explore for boundaries of a variable, I often try to vary data entered or processed. If the data entered is a string, I may try to find boundaries related to the length of the string and the amount of the string that's processed.

Testing guru James Bach's Web site, www.satisfice.com, offers Perlclip, a wonderful free tool that allows you to enter a string of different lengths into the Windows clipboard. These strings can then be pasted into data fields of dialogs. Perlclip lets you create *counterstrings* of different lengths. If an application truncates a counterstring, you can easily determine the number of characters processed, and thus identify a boundary.

Table 1 illustrates how a binary search can be used to identify boundaries of a variable. The software being investigated computes the tax rate based on income level; different tax rates are used for different ranges of income. The system precision is assumed to be in units of $1.00.

TABLE 1: BLACKBEARD'S BLACK BOX

STEP	MIN	MAX	NEW	Result	Comment
1			0	0	Low Tax Rate
1		0	1,000,000	50	High Tax Rate
2	0	1,000,0000	500,000	50	High Tax Rate
3	0	500,000	250,000	50	High Tax Rate
4	0	250,000	125,000	50	High Tax Rate
5	0	125,000	62,500	25	Medium Tax Rate
6	62,500	125,000	93,750	25	Medium Tax Rate
7	93,750	125,000	109,375	50	High Tax Rate
8	93,750	109,375	101,562.5	50	High Tax Rate
9	93,750	101,562	97,656	25	Medium Tax Rate
10	97,656	101,562	99,609	25	Medium Tax Rate
11	99,609	101,562	100,585.5	50	High Tax Rate
12	99,609	100,585	100,097	50	High Tax Rate
13	99,609	100,097	99,853	25	Medium Tax Rate
14	99,853	100,097	99,975	25	Medium Tax Rate
15	99,975	100,097	100,036	50	High Tax Rate
16	99,975	100,036	100,005.5	50	High Tax Rate
17	99,975	100,005	99,990	25	Medium Tax Rate
18	99,990	100,005	99,997.5	25	Medium Tax Rate
19	99,997	100,005	100,001	50	High Tax Rate
20	99,997	100,001	99,999	25	Medium Tax Rate
21	99,999	100,001	100,000	50	High Tax Rate/ Med.Tax Rate Boundary

Boundaries of numeric variables can be determined in many ways. For example, a binary search can be useful. Start by choosing two possible values: MIN and MAX. If the behavior observed is different at MIN and MAX, identify a NEW value that is halfway between.

If behavior at NEW is the same as MIN, replace MIN with the NEW value. If behavior at NEW is the same as MAX, replace MAX with the NEW value. Continue this process until the difference between MIN and MAX is within the precision of the variable being explored. Caveat: This approach has some risks, since you may not converge on all of the boundaries.

To identify boundaries based on applied differential calculus, you can try some other analytic techniques that can be useful when the application computes different values in a continuous range. I urge you to study Runge-Kutta, Newton and Newton-Raphson numerical methods to identify boundaries of continuous functions. Detailed descriptions of these approaches can be found at www.wikipedia.com.

When you're identifying boundaries, the application's behavior may vary depending on the order or progression of values tested. The application may have a bug in which memory is accidentally overwritten by data. I've tried to find boundaries by approaching them from large to small, or independently from small to large. In a number line, this would be considered approaching the value from the left or right. The application's behavior may be different depending on the approach.

It's also important to make sure each trial of the application begins at a controlled starting point, since the application's behavior varies based on the sequence of events, not just the specific values used in testing. I've occasionally had to restart from scratch each trial to help isolate a buffer overflow–related boundary.

It's possible to simulate changes to environment variables, such as the

amount of free disk space, memory or network resources. There are several commercial tools that allow you to simulate varying many system parameters without modifying the actual operating environment.

White-box testing is based on studying the detailed design, code and data structures used to implement the application. When I have access to such information, I use it to complement black-box approaches. I never rely exclusively on white-box approaches since they're based on the code and tend to confirm that the code does only what it *says* it does. I prefer to confirm that the code does what it's *expected* or *required* to do.

I use three approaches to white-box testing to help identify boundaries: structured walkthroughs, static analysis and data-flow analysis.

In the *structured walkthrough,* I ask the developer to walk through and explain to me all of the code used to implement the part of the application I'm testing. I focus on how data is processed and stored in memory. I don't generally follow all of the paths through the code. Instead, I walk through the normal cases and exceptional paths that commonly occur.

Whenever I identify processing or storage of data, I take note of the types of data and valid ranges. Decisions and case statements in the code are indicators of potential boundaries. Confirm boundaries found in the code by experimenting with a running application after the walkthrough. Look at database schemas to understand the types and relationships among records kept in persistent storage. Ranges, rules and triggers associated with elements of the database are all potential boundary conditions. Experiment with the running application to explore behavior around these boundaries.

In my white-box toolkit, I also use *static analysis* tools to help identify potential boundary conditions in code. A static analysis tool helps identify type *casting,* the cases in which the data type is changed at runtime by storing data of one type in another. For example, an integer is commonly cast into a character variable or a byte variable. Casting exposes potential boundary conditions related to the ranges of values.

I also identify boundaries related to the size of buffers used in memory to process data. Buffer sizes may be computed dynamically or assigned statically. Studying buffers in code helps identify potential boundaries.

The third white-box testing approach I use is *data-flow analysis.* In this approach, you examine all the places that variables are created, modified or destroyed. Each point where a data element is processed may expose a potential boundary.

Gray-box testing is a combination of black- and white-box approaches. In this method, I study the design to understand how data is processed and stored. Based on understanding the algorithms and data structures used, I identify potential boundaries. I then explore the behavior of the application around these boundaries.

In white-box testing, studying buffers in code can help to identify potential boundaries.

I use interviews with developers, architects and technical analysts to understand how multiple variables can combine to influence the behavior of the application being tested. Boundaries may depend on the relationship between variables as well as the specific values of each variable.

For example, consider the task of computing valid postal rates in Canada. The rate depends on the height, width and length of a package on an individual basis (for example, a maximum and minimum boundary for each), but the validation depends on the sum of all three values. If the sum of height, width and length exceeds some value, the postal rate computation changes.

Algorithms used to solve complex problems using numeric methods often have values that lead to unstable computations. Take, for example, a result that ends up with values that are divided by zero. Dividing by zero leads to an invalid number. Any number divided by zero that isn't represented by floating-point numbers in a digital computer is said to be infinite. By studying the numeric methods used in computation, we can also identify boundaries.

A *computational boundary,* therefore, is a value or combination of values that lead to unstable computations.

Any Means Justify the End

Different techniques can be used to identify boundaries in a software system. The critical task is to identify which variables influence the behavior of the application being tested. Once this is done, you can determine how they contribute to the processing of the application related to the test objective. For each contributing variable, different black-, white- or gray-box approaches can be used to explore boundaries and study the application's behavior as variables or combinations of variables approach their boundary conditions.

Next month, the concluding article in this series will investigate boundaries related to the behavior of the system and aspects of quality factors, as well as the limits of system behavior under load and in harshly constrained environments. I'll also examine boundaries of load, performance, capacity, environment and stress. Finally, I'll share some techniques to identify risks associated with multiple boundaries and experiments that push code to the limits. ☒