Unifying industrial and academic approaches to domain testing

CAST
Grand Rapids
August 3, 2010

Cem Kaner

Sowmya Padmanabhan

Abstract

The most widely used technique in software testing is called Equivalence Class Analysis. Or Category Partitioning. Or Boundary Value Testing. Or Domain Testing.

It's a black-box test technique—except when it's used as a glass-box technique.

Its focus is on input variables. Or output variables. Or variables that hold results, or are impacted by variables that hold results.

I've been trying to muddle through this literature for a decade, coming at it with a practitioner's bias and wondering whether the academics really had anything useful (or comprehensible) to offer.

Sowmya Padmanabhan did her M.Sc. thesis research on this. We are co-authoring the *Domain Testing Workbook*.

This talk will present a worksheet that we've developed for planning and creating domain tests of individual variables or multiple variables that are independent or linearly or nonlinearly related. I'll brush on some of the theory underlying the approach, but mainly I want to present some of the lessons that this work brought home to me about skilled (contrasted with inexperienced or unskilled) domain testing.

Domain Testing

Simple version:

- X is an integer variable. You can input values for X between 0 and 100.

Analysis:

- Test X with
  - -1 (barely too small)
  - 0 (smallest “valid” value)
  - 100 (largest “valid” value)
  - 101 (barely too big)
- We partitioned the values of X into three “equivalence classes:”
  - Too small
  - Just right
  - Too big
- Then we selected boundary values from each

So what did we just do?

The four central questions

- What domain are we testing?
- How do we determine how to group values of a variable(s) into equivalence classes?
- How do we determine which members of each class to test?
- How do we determine whether the program passed or failed the test?

Research

Cem Kaner & Sowmya Padmanabhan, "Practice and transfer of learning in the teaching of software testing," *Conference on Software Engineering Education & Training*, Dublin, July 2007. http://www.kaner.com/pdfs/kanerPadmanabhanPractice.pdf

http://www.kaner.com/pdfs/CSEETdomain2007.pdf (slides)

Padmanabhan, S. (2004). Domain Testing: Divide & Conquer. Unpublished M.Sc. Thesis., Florida Institute of Technology, Melbourne, FL. http://www.testingeducation.org/a/DTD&C.pdf.

- We tried teaching this technique via exemplars but had mixed results.
- We ultimately concluded that we needed to embed the exemplars in a stronger cognitive structure.

Domain

What domain are we testing?

- Input domains (the usual assumption)
- Output domains (typically ignored)

- We rarely buy programs to take advantage of their input-related features
- We buy them to get their *results*, i.e. their output.

Partitioning

How do we determine how to group variables into equivalence classes?

Partitioning a set means splitting it into nonoverlapping subsets.

Disjunction (nonoverlapping) is important for some models, but practitioners often work with overlapping sets (Kaner, et al., 1993; Weyuker & Jeng, 1991). So, for our purposes, in practice, partitioning means dividing a set of possible values of a variable into subsets that either don't overlap at all or don't overlap much.

Partitioning usually splits a set into equivalence classes.

Partitioning

Intuitive or subjective equivalence: Intuitive definitions (e.g., Craig & Jaskiel, 2002; Kit, 1995) appear obvious. For example,

- "Basically we are identifying inputs that are treated the same way by the system and produce the same results" Craig & Jaskiel (2002, p. 162).
- "If you expect the same result from two tests, you consider them equivalent. A group of tests forms an equivalence class if you believe they all test the same thing … Two people analyzing a program will come up with a different list of equivalence classes. This is a subjective process" Kaner (1988, p. 81).

Partitioning

- ***Specified equivalence: Under another common approach, variables' values are valid*** (according to the specification) ***or invalid***. The tester chooses cases from the valid set and each invalid set (Burnstein, 2003; DeMillo, McCracken, Martin, & Passafiume, 1987; Myers, 1979).
- ***Analysis of the code that defines or uses the variables: The intuitive and*** specified-equivalence approaches focus on the program's variables, primarily from a black box perspective. Some authors (e.g., Howden, 1980) suggest ways to partition data or identify boundary values by reading the code.
- ***Same path: Descriptions that appear primarily in the research literature say that two values of a variable are equivalent if they cause the program to take the same branch or the same (sub)path.*** (Beizer, 1995; Clarke, et al., 1982; White, 1981)

Partitioning

- ***Risk-based equivalence: Several early authors pointed out that domain tests*** target specific types of errors (Binder, 2000; Boyer, 1975; Clarke, 1976; White, 1981).
  - Weyuker & Ostrand (1980; *see also Jeng & Weyuker, 1989, p. 38)* proposed that the ideal partitioning should be *revealing*, *meaning that it*

    "divides the domain into disjoint subdomains with the property that within each subdomain, either the program produces the correct answer for every element or the program produces an incorrect answer for every element."

Partitioning

We adopt a subjective, risk-based definition of equivalence:

- Two tests are equivalent, relative to a potential error, if both should be error revealing (both could trigger the error) or neither should be revealing (Weyuker & Ostrand, 1980).
- The same test might be error-revealing relative to one potential error and not revealing relative to another.
- Given the same variable under analysis, two testers are likely to imagine different lists of potential errors because of their different experience history.

Selection

How do we determine which members of each class to test?

Boundaries are representatives of the equivalence classes we sample them from. They're more likely to expose an error than other class members, so they're *better representatives.*

"A *best representative of an equivalence class is a value that is at least as likely as any other value in the class to expose an error."* (Kaner & Bach, 2003, p. 37)

Generalizing from boundaries to ***best representatives*** is useful for analyzing nonordered spaces, such as printers (Kaner, et al., 1993).

Evaluation

How do we determine whether the program passed or failed the test?

- Stunningly little guidance on this in the domain testing literature
- The answer is generally treated as obvious, but it is not
- We did not buy the program in order to segregate 100 from 101 for the variable X. We did not buy the program to obtain an input filter.
- We bought the program to obtain benefits.
- We enter an X value to obtain benefits.
- We test a value of X
  - Against the filter
  - ***Against any use of X later in the program.***
- Failure to consider consequences is the most common failure of our students with this technique. It is widespread among practitioners too.

Our structure for domain testing

Here is a list of several tasks that people often do as part of a domain test. We organized the book's chapters around this list because it seems to us to put the tasks into a logical order.

For any particular product or variable, you might skip several of these tasks. Or, you might do tasks in a different order than we list here. This is an inventory, not a control structure.

A. Identify the potentially interesting variables.
B. Identify the variable(s) you can analyze now.
C. Determine the variable's primary dimension.
D. Determine the variable's type or scale.
E. Determine whether the variable's values can be ordered (smallest to largest)
F. Partition (create equivalence classes):
    - If the dimension is ordered, determine the sub-ranges and transition points.
    - If the dimension is not ordered, determine what "similar" means for this variable, and base partitioning on that.
G. Lay out the analysis in a table that shows the partitions and best representatives (or boundary cases) for each partition.
H. Identify secondary dimensions and analyze each in the classical way.
I. Generalize the analysis to multidimensional variables.
J. Linearize the domain (if possible).
K. Identify constraints among the variables
L. Identify and test variables that hold results (output variables).
M. Evaluate how the program uses the value of this variable.
N. Identify additional potentially-related variables for combination testing.
O. Create combination tests for multidimensional or related variables.
P. Imagine risks that don't necessarily map to an obvious dimension.
Q. Identify and list unanalyzed variables. Gather information for later analysis.
R. Summarize your analysis with a risk/equivalence table.

Our structure for domain testing

A. Identify the potentially interesting variables.
B. Identify the variable(s) you can analyze now.
   - Frequently, we notice more variables than we can analyze today.
   - Identify the ones you're going to work with *now*.
   - Note the others and add details when you get them (or when you go looking for them).

Our structure for domain testing

C. Determine the variable's primary dimension.

- If we're trying to put a number into X between 0 and 100, the primary dimension is a number line that includes [0, 100].
- To figure out the primary dimension of a variable, ask what the variable is for. What range of values will it contain if we only give it useful values? That range lies along the primary dimension.
- Sometimes, this is the *only* dimension
  - If X is stored in a byte and is read by a function that will interpret the bit pattern in that byte as a signed integer, then no matter what we put into X, what gets read is an integer between -128 and 127.
- Other times, there are possibilities outside of this dimension. We'll talk about these as *secondary dimensions*.

Our structure for domain testing

D. Determine the variable's type or scale.

We will analyze

- integers

very differently from

- strings.

Our structure for domain testing

E. Determine whether the variable's values can be ordered (smallest to largest).
   - The classic examples are of ordered sets
   - Domain testing is an instance of stratified sampling
   - In other fields (e.g. polling), we can do stratified sampling without requiring an order
   - Example: stratified sampling
     - Call 15 Jewish Republicans who live in Salt Lake (a sample of a larger set of Jews, a larger set of Republicans, a larger set of people who live in Salt Lake, and a larger set of Republican Jews in Salt Lake)
     - Select 3 Mac-compatible printers that print PostScript version 4 or later
   - Polling looks for typical representatives
   - Testing looks for "best" representatives (most likely to yield failure or other new information)

Our structure for domain testing

F. Partition (create equivalence classes):

- If the dimension is ordered, determine the sub-ranges and transition points.
- If the dimension is not ordered, determine what “similar” means for this variable, and base partitioning on that.
- We partitioned the values of X into three “equivalence classes:”
  - Too small
  - Just right
  - Too big
- For partitioning of non-orderable sets, see the Printers example in Kaner / Falk / Nguyen, Testing Computer Software

Our structure for domain testing

G. Lay out the analysis in a table that shows the partitions and best representatives (or boundary cases) for each partition.

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
X	0 – 100		0	
			100	
		< 0		
		> 100		

Note that this shows only the primary dimension.

Our structure for domain testing

H. Identify secondary dimensions and analyze each in the classical way.

- This is where we consider other data types and other sources of variation that may affect the handling of this variable
  - Non-integer input
  - Non-numeric input
  - Buffer-attackingly large or small values
  - Time
- Weinberg/Myers included these dimensions in their presentations
- Many others (Binder, academics) ignored them completely (e.g. Binder's treatment of Triangle, over 100 tests, all on primary dimension)
- No other discussions that we've seen that distinguish primary from secondary (what the variable/feature is for versus what else might happen to it.)

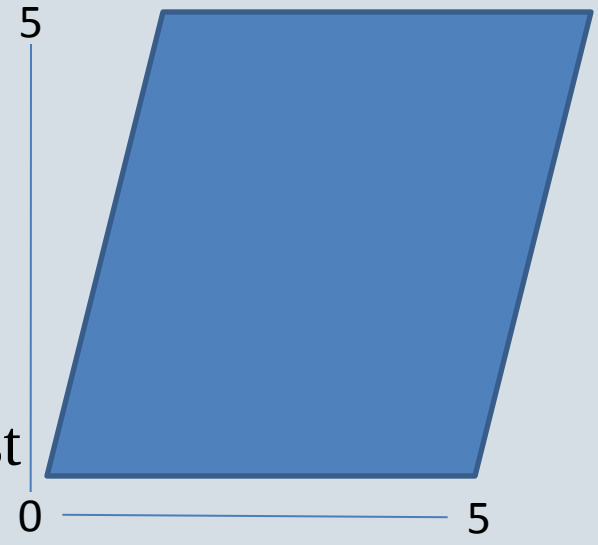
Our structure for domain testing

I. Generalize the analysis to multidimensional variables.
  - Copeland cites this as the breakpoint between equivalence class / boundary analysis & domain analysis because so many academic treatments (I include Beizer in these) present domain testing as a multidimensional (two or three variable) technique
  - The sampling theory is the same: partition & select
  - If variables are *independent*, we can use combination techniques for independent variables (e.g. combinatorial or random)
  - When value of one variable constrains the available values for the other(s), combinatorial (e.g. all-pairs) fails. Instead, we have to map the multidimensional space as it is and sample from the actual boundaries.

Our structure for domain testing

I. Generalize the analysis to multidimensional variables.
   - Here, for example, combinations of
     - X = 0 or 6 (its boundaries)
     - Y = 0 or 5 (its boundaries)

     aren't of much interest because if we test (X,Y) with something like all-pairs:
     - (0, 0) is on boundary
     - (6, 0) is not near any boundary
     - (5, 0) is on boundary
     - (0, 5) is not near any boundary



Our structure for domain testing

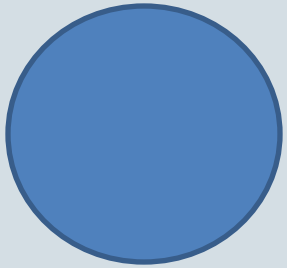
J. Linearize the domain (if possible).

Extensive academic writing about simple linear inequalities:

$$0 \leq X \leq 1$$
$$X \leq Y \leq 5+X$$

- On points, off points, in points, out points
- Special cases that have confused students for years
- The heuristics fall apart for simple nonlinear cases, like:

$$X^2 + Y^2 \leq 10$$



Our structure for domain testing

K. Identify constraints among the variables
  - 30 days hath September … (month constrains days)
  - How much is sales tax here?
  - Oh, *that's* against the law *here*?

Our structure for domain testing

L. Identify and test variables that hold results (output variables).
  - Simple example: Mailing labels
    - Inputs: Your first, middle and last names
    - Output: A label that your name can’t fit on

Our structure for domain testing

M. Evaluate how the program uses the value of this variable.

- Every variable that can be affected by this variable can be tested with boundaries (and other interesting values) of this variable
- 1^{st} through N^{th} order data flows
- Program slicing addresses this in maintenance

Our structure for domain testing

N. Identify additional potentially-related variables for combination testing.

O. Create combination tests for multidimensional or related variables.

- Earlier, we considered an individual variable that was inherently multidimensional (a point)
- Now consider testing a bunch of variables together that are not inherently dimensions of one larger variable.
- We can reduce the enormous set of tests via stratified sampling
  - All-pairs is one stratified sampling approach (sample with a coverage criterion)
  - Cause-effect graphing is similar but for constraining variables
  - Other possibilities …

Our structure for domain testing

P. Imagine risks that don't necessarily map to an obvious dimension.
   - Domain analysis is the start, not the end of test design

Our structure for domain testing

Q. Identify and list unanalyzed variables. Gather information for later analysis.

- You will always have more research to do
- You will never have a complete test plan

Our structure for domain testing

R. Summarize your analysis with a risk/equivalence table.

Variable	Risk (potential failure	Classes that should not trigger the failure	Classes that might trigger the failure	Test cases (best representatives)	Notes
X	Mishandles too-small values	≥ 0	< 0	0	
				-1	
				-999999999etc	overflow
	Mishandles alpha				

The risk-oriented table is more complex to work with when dealing with simple variables. We often prefer the classical table for simple, academic examples.

The weakness of the very simple examples is that they are divorced from real-life software. You analyze a variable, but you don't know why a program needs it, what the program will do with it, what other variables will be used in conjunction with it. As soon as you know the real-life information, many risks (should) become apparent, risks that you can study by testing different values of this variable. The risk-oriented table helps us organize that testing. Any time you are thinking beyond the basic "too big / too small" tests, this style of table might be more helpful than the classical one.