<ch5 toc ch7>

Chapter 6
DOMAIN TESTING

1. SYNOPSIS

Programs as input data classifiers: domain testing attempts to determine whether the classification is or is not correct. Application of domain testing to interfaces and integration, domain design, software design for testability, and limitations.

2. DOMAINS AND PATHS

2.1. The Model

Domain testing can be based on specifications and/or equivalent implementation information. If domain testing is based on specifications, it is a functional test technique; if based on implementations, it is a structural technique. Domain testing, as practiced, is usually applied to one input variable or to simple combinations of two variables, based on specifications. For example, you're doing domain testing when you check extreme values of an input variable. Domain testing as a theory, however, has been primarily structural.

All inputs to a program can be considered as if they are numbers. For example, a character string can be treated as a number by concatenating bits and looking at them as if they were a binary integer. This is the view in domain testing, which is why this strategy has a mathematical flavor.

Before doing whatever it does, a routine must classify the input and set it moving on the right path. Figure 6.1 is a schematic representation of this notion. Processing begins with a classifier section that partitions the input vector into cases. An invalid input (e.g., value too big) is just a special processing case called "reject," say. The input then passes to a hypothetical subroutine or path that does the processing. In domain testing we focus on the classification aspect of the routine rather than on the calculations.

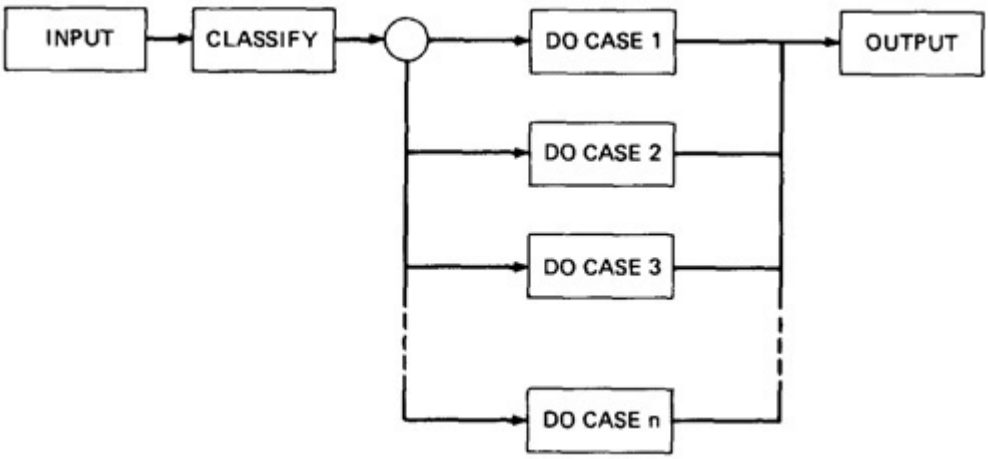


Figure 6.1. Schematic Representation of Domain Testing.

Structural knowledge is not needed for this model—only a consistent, complete specification of input values for each case. Even though we might not use structural information, we can infer that for each

case there must be at least one path to process that case. That path need not be obvious and at the level of the tested routine. For example, case classification and/or case processing could be buried in a low-level subroutine. Because domain testing can be a functional test technique, not only do we need not know anything about the hypothetical path, but usually we don't care. When we talk about "paths through the routine" in this chapter, it is understood that we mean actual paths in the routine, paths in subroutines, table entries, or whatever else is needed to process each case. If we base domain testing on structure rather than on specifications, the hypothetical paths will be actual paths.

2.2. A Domain Is a Set

An input domain is a set. If the source language supports set definitions (e.g., Pascal set types, C enumerated types) less testing is needed because the compiler (compile-time and run-time) does much of it for us. Domain testing doesn't work well with arbitrary discrete sets of data objects because there are no simple, general strategies. In Section 3 below we discuss domain problems (ugly domains): arbitrary, discrete sets suffer from many of those problems. Our discussion will focus on domains that either are numerical or can be thought of as numerical.

The language of set theory is natural for domain testing. We speak of connected and disconnected domains, closure properties, boundaries, intersections and unions, and so on. Readers who were victimized by the "new math" may remember these concepts from elementary school and may have wondered what use they were; they will be useful now. Readers unfamiliar with set theory should read an introduction such as a high school text. Computer science graduates and software engineers usually know some set theory but probably never used it for anything practical. We won't be using deep concepts, but a terminology review might be in order.

2.3. Domains, Paths, and Predicates

In domain testing, predicates are assumed to be interpreted in terms of input vector variables. If domain testing is applied to structure, then predicate interpretation must be based on actual paths through the routine—that is, based on the implementation control flowgraph. Conversely, if domain testing is applied to specifications, interpretation is based on a specified data flowgraph for the routine; but usually, as is the nature of specifications, no interpretation is needed because the domains are specified directly. I will be deliberately vague about whether we're dealing with implementations or specifications when the discussion applies equally well to either. For the most part, though, there will be a specification bias because that's where I believe domain testing has the best payoff.

For every domain there is at least one path through the routine. There may be more than one path if the domain consists of disconnected parts or if the domain is defined by the union of two or more domains. Unless stated otherwise, we'll assume that domains consist of a single, connected part. We'll also assume (for now) that the routine has no loops. Domains are defined by their boundaries. Domain boundaries are also where most domain bugs occur. For every boundary there is at least one predicate that specifies what numbers belong to the domain and what numbers don't. For example, in the statement IF x>0 THEN ALPHA ELSE BETA we know that numbers greater than zero belong to ALPHA processing domain(s) while zero and smaller numbers belong to BETA domain(s). A domain may have one or more boundaries—no matter how many variables define it. For example, if the predicate is $x^2 + y^2 < 16$, the domain is the inside of a circle of radius 4 about the origin. Similarly, we could define a spherical domain with one boundary but in three variables. Domains are usually defined by many boundary segments and therefore by many predicates.

In typical programs, domain boundary predicates alternate with processing. A domain might be defined

by a sequence of predicates—say A, B, and C. First evaluate A and process the A cases, then evaluate B and do B processing, and then evaluate C and finish up. With three binary predicates, there are up to eight (2^3) domains, corresponding to eight possible combinations of TRUE/FALSE outcomes of the three predicates. It could be as few as two domains if, for example, the ABC (boolean) case is processed one way and the rest of the cases $(\overline{A} + \overline{B} + \overline{C})$ another way.

To review:

1. A domain for a loop-free program corresponds to a set of numbers defined over the input vector.
2. For every domain there is at least one path through the routine, along which that domain's processing is done.
3. The set of interpreted predicates traversed on that path (i.e., the path's predicate expression) defines the domain's boundaries.

2.4. Domain Closure

Figure 6.2 shows three situations for a one-dimensional domain—i.e., a domain defined over one input variable; call it *x*. As in set theory, a domain boundary is **closed** with respect to a domain if the points on the boundary belong to the domain. If the boundary points belong to some other domain, the boundary is said to be **open**. Figure 6.2 shows three domains called D1, D2, and D3. In Figure 6.2a, D2's boundaries are closed both at the minimum and maximum values; that is, both points belong to D2. If D2 is closed with respect to *x*, then the adjacent domains (D1 and D3) must both be open with respect to *x*. Similarly, in Figure 6.2b, D2 is closed on the minimum side and open on the maximum side, meaning that D1 is open at the minimum (of 132) and D3 is closed at D2's maximum. Figure 6.2c shows D2 open on both sides, which means that those boundaries are closed for D1 and D2. The importance of domain closure is that incorrect closure bugs are frequent domain bugs. For example, x >= 0 when x > 0 was intended.

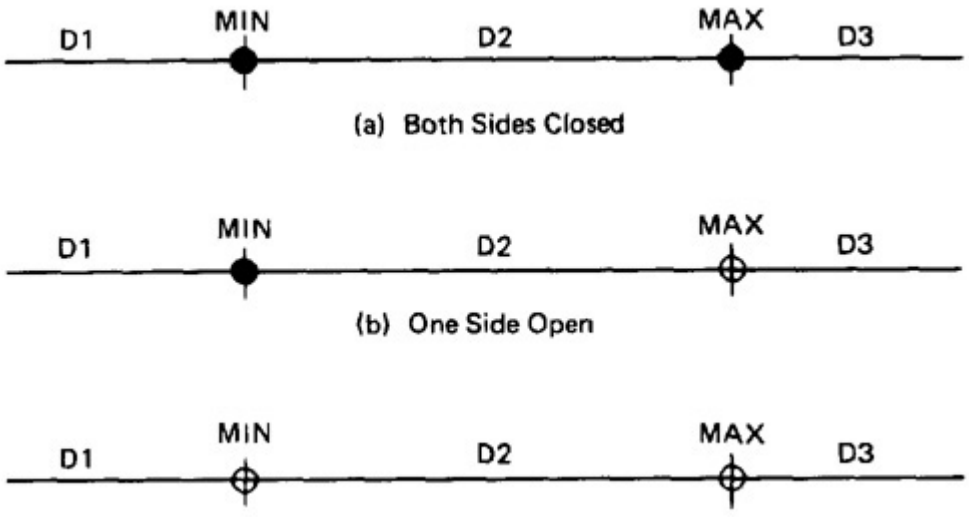


Figure 6.2. Open and Closed Domains.

2.5. Domain Dimensionality

Every input variable adds one dimension to the domain. One variable defines domains on a number line, two variables define planar domains, three variables define solid domains, and so on. Don't confuse the domain's dimensionality with the number of boundary predicates. There must be at least one boundary predicate or there's nothing to test from the point of view of domain testing; however, there's no limit to

the number of boundary predicates. Every new predicate slices through previously defined domains and cuts (at least one of) them in half. A piece of salami (the input vector space) is a three-dimensional object (three variables), but a food processor can create far more than three slices or chunks—but no less than two or else it isn't slicing.

Every boundary slices through the input vector space with a dimensionality which is less than the dimensionality of the space. Thus, planes are cut by lines and points, volumes by planes, lines and points, and ***n*-spaces** by **hyperplanes**. Because a domain could consist of only points along a line, or for that matter a single point, the dimensionality of the predicate can be any value less than the dimensionality of the space. Spaces of more than three dimensions are called ***n*-spaces**. Things that cut through *n*-spaces are called **hyperplanes** (not starships). An input array with dimension 100 is not unusual, but it is a 100-dimensional space. Input spaces of dozens of dimensions are common for even modest routines. It's obvious that pictures, intuition, and visualization rapidly become useless. Domain testing can be done manually for one or two dimensions, but it is tool-intensive in general.

2.6. The Bug Assumptions

The bug assumption for domain testing is that processing is okay but the domain definition is wrong. An incorrectly implemented domain means that boundaries are wrong, which may in turn mean that control-flow predicates are wrong. Although we can infer that an incorrect boundary results in a control-flow bug, we can't assume that the bug is at the level of the routine we're testing. The faulty boundary could be decided by a lower-level subroutine, by a faulty call to an otherwise correct subroutine, or by a faulty table entry. We also assume that once the input vector is set on the right path, it will be correctly processed. This last assumption implies that domain testing should be augmented by other testing to verify that processing is correct: for example, at least one case within the domain. Many different bugs can result in domain errors. Here is a sample of more common ones:

1. *Double-Zero Representation*—In computers or languages that (unfortunately) have a distinct positive and negative zero, boundary errors for negative zero are common.
2. *Floating-Point Zero Check*—A floating-point number can equal zero only if the previous definition of that number set it to zero or if it is subtracted from itself, multiplied by zero, or created by some operation that forces a zero value. Floating-point zero checks should always be done about a small interval, typically called "epsilon," which for that application has been defined as "close enough to zero for all practical purposes."
3. *Contradictory Domains*—An implemented domain can never be ambiguous or contradictory, but a specified domain can. A contradictory domain specification means that at least two supposedly distinct domains overlap. Programmers resolve contradictions by assigning overlapped regions to one or the other domain for a 50-50 chance of error.
4. *Ambiguous Domains*—Ambiguous domains means that the union of the specified domains is incomplete; that is, there are either missing domains or holes in the specified domains. Not specifying what happens to points on the domain boundary is a common ambiguity.
5. *Overspecified Domains*—The domain can be overloaded with so many conditions that the result is a null domain. Another way to put it is to say that the domain's path is unachievable.
6. *Boundary Errors*—Domain boundary bugs are discussed in further detail in Section 4, below, but here's a few: boundary closure bug, shifted, tilted, missing, extra boundary.
7. *Closure Reversal*—A common bug. The predicate is defined in terms of >=. The programmer chooses to implement the logical complement and incorrectly uses <= for the new predicate; i.e., x >= 0 is incorrectly negated as x <= 0, thereby shifting boundary values to adjacent domains.
8. *Faulty Logic*—Compound predicates (especially) are subject to faulty logic transformations and improper simplification. If the predicates define domain boundaries, all kinds of domain bugs can result from faulty logic manipulations.

2.7. Restrictions

2.7.1. General

Domain testing has restrictions, as do other testing techniques. They aren't restrictions in the sense that you can't use domain testing if they're violated—but in the sense that if you apply domain testing to such cases, tests are unlikely to be productive because they may not reveal bugs or because the number of tests required to satisfy the criterion is greater than practicality warrants. In testing (other than faulty outcome prediction, improper execution, or other test design and execution bugs), there are no invalid tests—only unproductive tests.

2.7.2. Coincidental Correctness

Coincidental correctness is assumed not to occur. Domain testing isn't good at finding bugs for which the outcome is correct for the wrong reasons. Although we're not focusing on outcome in domain testing, we still have to look at outcomes to confirm that we're in the domain we think we're in. If we're plagued by coincidental correctness we may misjudge an incorrect boundary. Note that this implies weakness for domain testing when dealing with routines that have binary outcomes (i.e., TRUE/FALSE). If the binary outcome is INSIDE/OUTSIDE (the domain), as in data-validation routines, domain testing will probably be effective. But if the domain is a disconnected mess of small subdomains, each of which has a binary outcome or if outcomes are restricted to a few discrete values, then coincidental correctness is likely and domain testing may not be revealing.

2.7.3. Representative Outcome

Domain testing is an example of **partition testing. Partition-testing** strategies divide the program's input space into domains such that all inputs within a domain are equivalent (not equal, but equivalent) in the sense that any input represents all inputs in that domain. If the selected input is shown to be correct by a test, then processing is presumed correct, and therefore all inputs within that domain are expected (perhaps unjustifiably) to be correct. Most test techniques, functional or structural, fall under partition testing and therefore make this **representative outcome** assumption. Another way to say it is that only one function is calculated in a domain and that adjacent domains calculate different functions. This is not equivalent to barring coincidental correctness—coincidental correctness concerns the value of functions, whereas a representative outcome concerns the functions themselves. For example, x^2 and 2^x are equal for $x = 2$, but the functions are different. The functional differences between adjacent domains are usually simple, such as $x + 7$ versus $x + 9$, rather than x^2 versus 2^x.

2.7.4. Simple Domain Boundaries and Compound Predicates

Each boundary is defined by a simple predicate rather than by a compound predicate. We want to avoid, among other things, inconsistent boundary closures. Compound predicates in which each part of the predicate specifies a different boundary are not a problem: for example, x >= 0 .AND. x < 17, just specifies two domain boundaries by one compound predicate. As an example of a compound predicate that specifies one boundary, consider: x = 0 .AND. y >= 7 .AND. y <= 14. This predicate specifies one boundary equation ($x = 0$) but alternates closure, putting it in one or the other domain depending on whether $y < 7$ or $y > 14$.

Compound predicates that include ORs can create concave domains (see Section 3.3.7), domains that are not simply connected (see Section 3.3.8) and violate the requirement that adjacent domains compute different functions. A domain defined by a predicate such as ABC + DEF defines two subdomains (by three boundaries corresponding to ABC or three boundaries corresponding to DEF). These subdomains

can be separated, adjacent but not overlapped, partially overlapped, or one can be contained within the other; all of these are problematic.

Eliminating compound predicates is usually enough to guarantee simple boundary closures and connected, convex domains, but it is probably too strong a requirement. The predicates usually employ numerical relational operators: >, >=, =, <=, <>, and <. Although elaborate boundary closures can be constructed using these operators and compound predicates, it's not likely, except in mathematical software. In real situations, exotic boundaries don't come up often, but multiply connected and/or overlapped domains can and do occur with compound predicates. Note also that if one leg of a binary predicate is compound with only ANDs, then the other leg, because it is a boolean negation, *must* contain ORs and therefore can create disconnected, overlapped, or concave domains. Treat compound predicates with respect because they're more complicated than they seem.

2.7.5. Functional Homogeneity of Bugs

Whatever the bug is, it will not change the functional form of the boundary predicate. For **linear predicates** (i.e., boundary predicates that are linear functions) the bug is such that the resulting predicate will still be linear. For example, if the predicate is $ax >= b$, the bug will be in the value of a or b but it will not change the predicate to $a^x >= b$, say.

2.7.6. Linear Vector Space

A **linear (boundary) predicate** is defined by a linear inequality (*after interpretation in terms of input variables*),* using only the simple relational operators >, >=, =, <=, <>, and <. Most papers on domain testing, such as WHIT81, assume linear boundaries—not a bad assumption because in practice most boundary predicates *are* linear. A more general assumption is that boundaries can be embedded in a linear vector space. For example, the predicate $x^2 + y^2 > a^2$ is not linear in rectangular coordinates, but by transforming to polar coordinates we obtain the equivalent linear predicate $r > a$. Similarly, a polynomial boundary can be transformed into a linear vector space by $y_1 = x, y_2 = x^2, y_3 = x^3, \ldots$. Polynomial and other nonlinear boundaries have been examined by Zeil and White (ZEIL81, ZEIL84). The difficulties with nonlinear boundaries are practical and theoretical. The practical problems are twofold: significant increase in the number of tests needed to confirm boundaries and major escalation in calculations to determine test points. In real testing, asking for a linear vector space boils down to simple linear predicates.

---

*Emphasize linearity "after interpretation in terms of input variables." A predicate could *appear* to be linear, say x + y > 0, but not be linear after interpretation because there had been, say, a previous assignment such as $x := y^2$. Similarly, a nonlinear predicate could become linear after interpretation. To make matters worse, the interpretation could depend on the path—linear for some paths and nonlinear for others. Despite these potential (but rare) difficulties, it's a useful technique.

2.7.7. Loop-free Software

Loops are problematic for domain testing. The trouble with loops is that each iteration can result in a different predicate expression (after interpretation), which means a possible domain boundary change. If a loop is an overall control loop on transactions, say, there's no problem. We "break" the loop and use domain testing for the transaction process. If the loop is **definite** (that is, if we know on entry exactly how many times it will loop), then domain testing may be useful for the processing within the loop, and

loop testing discussed in Chapter 3 can be applied to the looping values. The really troublesome loops for domain testing are indefinite loops, such as in iterative procedures.

Despite the theoretical difficulties of applying domain testing to software with loops, White and Wiszniewski (WHIT88, WISZ85, WISZ87) have investigated the problem, especially testing complexity. Early research indications are that domain-testing tools for programs with loops may emerge, but it will take time to determine whether the loop methodology is cost-effective in practice.

3. NICE DOMAINS AND UGLY DOMAINS

3.1. Where Do Domains Come From?

Domains are often created by salesmen or politicians. I shouldn't knock salesmen and politicians, but we know that they aren't as careful about "minor" issues of consistency as we'd like them to be—especially if by ignoring such problems they befriend the customer and make the sale. There's always a salesman at the top of the design process—a person who negotiates functionality with a buyer in the loosest possible way while providing the greatest possible appearance of consistency and completeness. It's an emotional appeal rather than a rational process that submits to logic. There's always a salesman in the act—be it a retailer who sells software to a user or a manager who "sells" the project up the line (or up the creek). Don't fault salesmen for their (typical) shortsighted view and desire for instant gratification—it's an essential aspect of a personality needed to get a job done. And if they didn't sell the "impossible" once in a while, where would progress be? I don't fault them—I don't even want to change them—but as a designer and tester I've learned to be cynical about their domain "definitions." As for politicians? The difficulties you have with your income tax returns were caused by politically defined domains.

Domains are and will be defined by an imperfect iterative process aimed at achieving (user, buyer, voter) satisfaction. We saw in Chapter 2 that requirement bugs are among the most troublesome of all. Many requirement bugs can be viewed as ill-defined domains—meaning ill-defined domain boundaries. The first step in applying domain testing should be to analyze domain definitions in order to get (at least) consistent and complete domain specifications. As designers and testers, we have been far too timid about challenging specified domains. Most domain problems discussed below (Section 3.4) can be avoided.

3.2. Specified Versus Implemented Domains

Implemented domains can't be incomplete or inconsistent. Every input will be processed (rejection is a process), possibly forever. Inconsistent domains will be made consistent. Conversely, specified domains can be incomplete and/or inconsistent. **Incomplete** in this context means that there are input vectors for which no path is specified, and **inconsistent** means that there are at least two contradictory specifications over the same segment of the input space. If a program is based on such specified domains, because the program can't be either incomplete or inconsistent (in the above sense), there must be a bug.

3.3. Nice Domains

3.3.1. General

Figure 6.3 shows some nice, typical, two-dimensional domains. The boundaries have several important properties discussed below: they are linear, complete, systematic, orthogonal, consistently closed,

simply connected, and convex. To the extent that domains have these properties, domain testing is as easy as testing gets. To the extent that these properties don't apply, testing is as tough as it gets. What's more important is that bug frequencies are much lower for nice domains than for ugly domains.

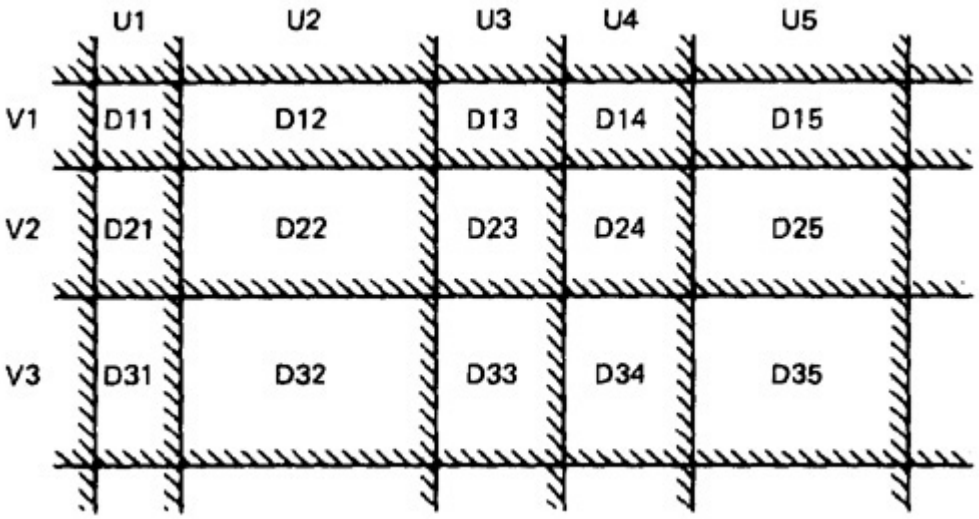


Figure 6.3. Nice Two-Dimensional Domains.

3.3.2. Linear and Nonlinear Boundaries

Nice domain boundaries are defined by linear inequalities or equations—*after interpretation in terms of input variables*. The impact on testing stems from the fact that it takes only two (test) points to determine a straight line, three points to determine a plane, and in general $n + 1$ points to determine an n-dimensional hyperplane. Add one point to test boundary closure correctness and you have it all.

The restriction to linear boundaries might seem to severely restrict the applicability of domain testing, but that's not so. Cohen (COHE78) studied 50 COBOL programs and found only one nonlinear predicate out of 1070. Studies by Knuth (FORTRAN) and Elshoff (PL/1) reported in WHIT80A support the observation that almost all boundary predicates met in practice are linear. There are no published data on how many nonlinear predicates could be linearized by simple transformations (see Section 6.2 below). My guess is that, in practice, more than 99.99% of all boundary predicates are either directly linear or can be linearized by simple variable transformations.

3.3.3. Complete Boundaries

Nice domain boundaries are complete in that they span the number space from plus to minus infinity in all dimensions. Figure 6.4 shows some incomplete boundaries. Boundaries A and E have gaps. Such boundaries can come about because the path that hypothetically corresponds to them is unachievable, because inputs are constrained in such a way that such values can't exist, because of compound predicates that define a single boundary, or because redundant predicates convert such boundary values into a null set. The advantage of complete boundaries is that one set of tests is needed to confirm the boundary no matter how many domains it bounds. If the boundary is chopped up and has holes in it, then every segment of that boundary must be tested for every domain it bounds.

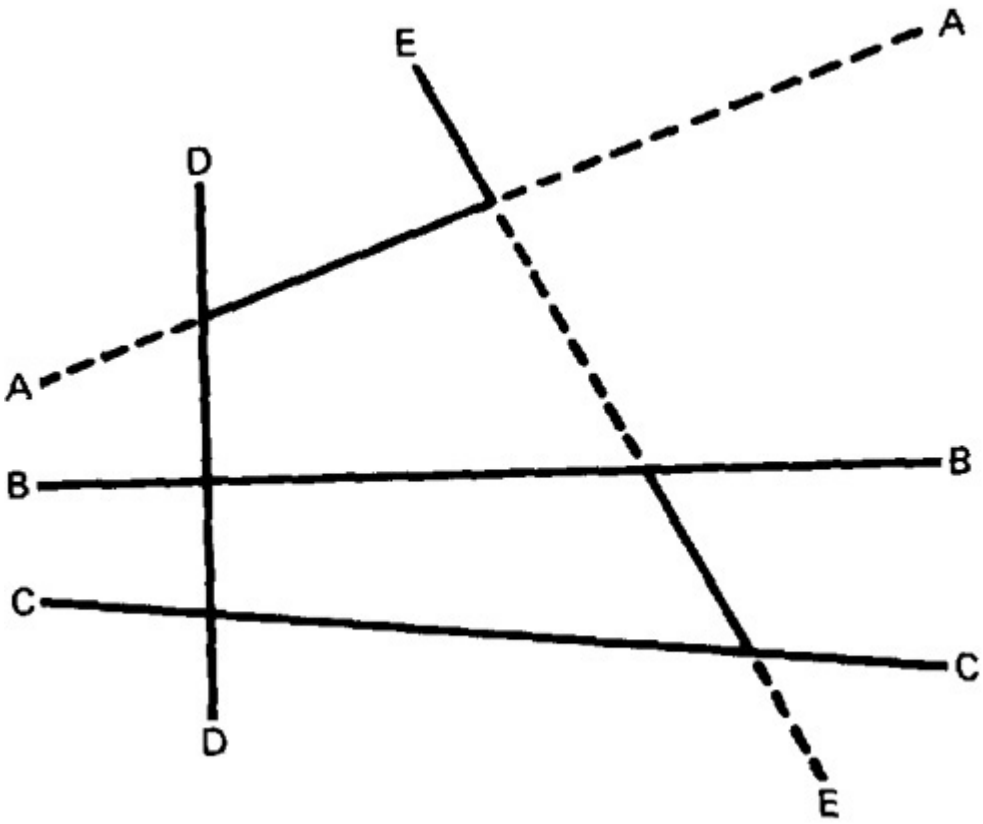


Figure 6.4. Incomplete Domain Boundaries.

3.3.4. Systematic Boundaries

By **systematic boundaries** I mean boundary inequalities related by a Simple function such as a constant. In Figure 6.3 for example, the domain boundaries for u and v differ only by a constant. We want relations such as

$$f_1(X) >= k_1 \text{ or } f_1(X) >= g(1,c)$$
$$f_1(X) >= k_2 \quad f_2(X) >= g(2,c)$$
$$\ldots\ldots\ldots\ldots \quad \ldots\ldots\ldots\ldots$$
$$f_i(X) >= k_i \quad f_i(X) >= g(i,c)$$

where f_i is an arbitrary linear function, X is the input vector, k_i and c are constants, and $g(i,c)$ is a decent function over i and c that yields a constant, such as $k + ic$. The first example is a set of parallel lines, and the second example is a set of systematically (e.g., equally) spaced parallel lines. I would also call a set of lines through a point, such as the spokes of a wheel, if equally spaced in angles, systematic. If the boundaries are systematic and if you have one tied down and generate tests for it, the tests for the rest of the boundaries in that set can be automatically generated.

3.3.5. Orthogonal Boundaries

The U and V boundary sets in Figure 6.3 are **orthogonal;** that is, every inequality in V is perpendicular to every inequality in U. The importance of this property cannot be minimized. If two boundary sets are orthogonal, then they can be tested independently. It's the difference between linear and nonlinear test growth. In Figure 6.3 we have six boundaries in U and four in V. We can confirm the boundary properties in a number of tests proportional to 6 + 4 = 10 (O(n)). If we tilt the boundaries to get Figure

6.5, we must now test the intersections. We've gone from a linear number of cases to a quadratic: from O(n) to O(n^2).*

*The notation O(n) means of the order of. For example, O(n) is a linear growth, O(n^2) is a quadratic growth, and O(2^n) is an exponential growth.

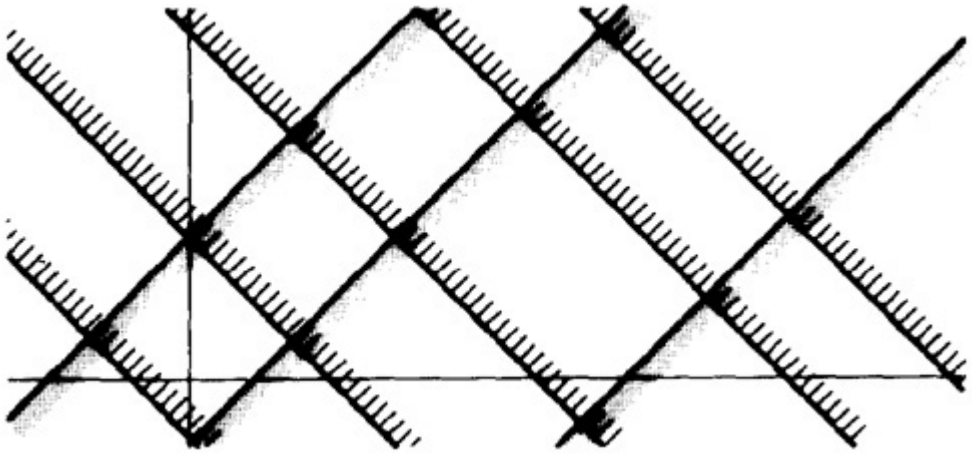


Figure 6.5. Tilted Boundaries.

Actually, there are two different but related orthogonality conditions. Sets of boundaries can be orthogonal to one another but not orthogonal to the coordinate axes (condition 1), or boundaries can be orthogonal to the coordinate axes (condition 2). The first case allows us to simplify intersection testing. The second case means that boundaries are functions of only one variable. Both are desirable properties. Figure 6.6 shows the difference. The boundaries defined by x ($x = A_1, \ldots$) are orthogonal to the x axis and are therefore functions only of x. The other set of boundaries, though systematic, is not orthogonal to either coordinate axis and therefore not to the other set of boundaries. Orthogonality can depend on the coordinate system chosen. For example, concentric circles intersected by equally spaced spokes aren't orthogonal in rectangular coordinates but if you express the variables in polar coordinates, testing is no worse than for Figure 6.3.

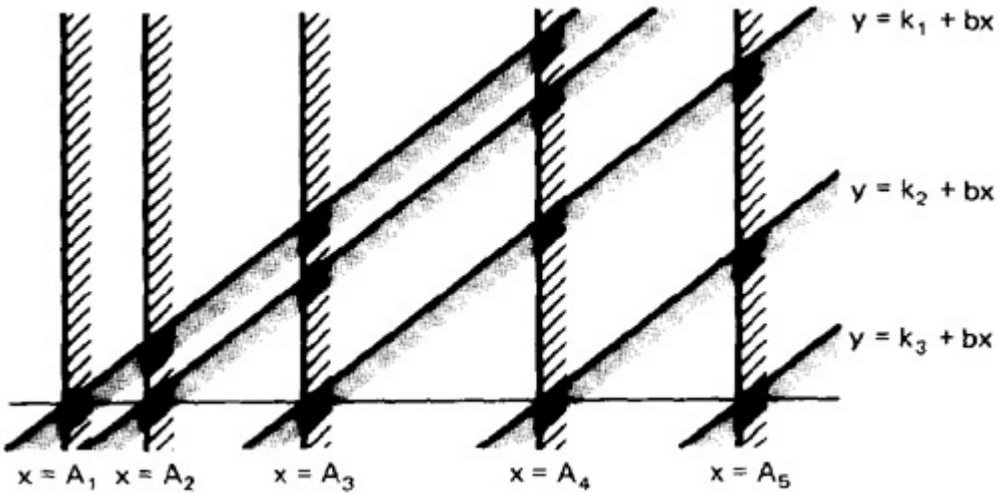


Figure 6.6. Linear, Nonorthogonal Domain Boundaries.

3.3.6. Closure Consistency

Figure 6.6 shows another desirable domain property: boundary closures are consistent and systematic. The shaded areas on the boundary denote that the boundary belongs to the domain in which the shading lies—e.g., the boundary lines belong to the domains on the right. Consistent closure means that there is a simple pattern to the closures—for example, using the same relational operator for all boundaries of a set of parallel boundaries.

Inconsistent closures are often not fundamental; they're often arbitrary and result from hasty solutions rather than analysis. The programmer has to know how to treat the points on every domain boundary. As a programmer you know all about domains and especially adjacent domains, even if you haven't talked about them in these terms before. You spot a boundary ambiguity and you ask "What do I do when . . . *equals* . . .?" If you don't know that there's a pattern to the boundaries, or that boundaries enclose a case in a certain way, then the answer is likely to be arbitrary. Another source of inconsistent closures is a misguided attempt to "simplify" the problem. The salesman doesn't want to burden you with "complexities" such as dealing with >= rather than >. She doesn't realize that programming one relational operator is the same as another so he says, "Use .GT. rather than .GE.", because he thinks that software will be simpler as a result. I've wasted more time than I can reckon trying to program an arbitrary requirement intended to "simplify" my work. Pay special attention to domain boundary closures during requirements reviews. Look for closure patterns and deviations from patterns. Confirm that specified closures are actual rather than arbitrary requirements. You'll simplify not only programming but also testing.

3.3.7. Convex

A geometric figure (in any number of dimensions) is **convex** if you can take two arbitrary points on any two different boundaries, join them by a line and all points on that line lie within the figure. Nice domains are convex; dirty domains aren't. Linear convex domains look like polished, *n*-dimensional diamonds. Linear domains with concavities are like diamonds with chips gouged out of them. The reliability of domain testing breaks down for domains with concavities—specifically, the *n*-on, one-off strategy discussed in Section 4.2 of this chapter may not work. We should start by looking at the specification for signs of concavity. You can smell a suspected concavity when you see phrases such as: ". . . except if . . .," "However . . .," ". . . but not. . . ." In programming, it's often the buts in the specification that kill you.

3.3.8. Simply Connected

Nice domains are **simply connected;** that is, they are in one piece rather than pieces all over the place interspersed with other domains (more "buts," "excepts," etc.). Nice domains are solid rather than laced through with exceptions like termite-infested beams. Simple connectivity is a weaker requirement than convexity; if a domain is convex it is simply connected, but not vice versa.

Consider domain boundaries defined by a compound predicate of the (boolean) form ABC. Say that the input space is divided into two domains, one defined by ABC and, therefore, the other defined by its negation $\overline{A} + \overline{B} + \overline{C}$. The inverse predicate is the union of three predicates, which can define up to three overlapped domains. If the first domain is convex, its complement can't be convex or simply connected—it must have a hollow created by the first domain. If one domain slices out a hunk of number space, the complementary domain must have holes or be in pieces. For example, suppose we define valid numbers as those lying between 10 and 17 inclusive. The invalid numbers are the disconnected domain consisting of numbers less than 10 and greater than 17.

Simple connectivity, especially for default cases, may be impossible. It would be a poor idea to implement default cases as the primary driver of the domain logic if the default domain was concave or

multiply connected. The smart strategy would be to do case logic first and let defaults fall through. Conversely, if the default domain is simply connected, the union of the working cases may not be simply connected but is surely concave, so the smart design (and test) strategy would be to do default case analysis first and to fall through to working cases.

3.4. Ugly Domains and How Programmers and Testers Treat Them

3.4.1. General

Some domains are born ugly and some are uglified by bad specifications. Programmers in search of nice solutions will "simplify" essential complexity out of existence. Testers in search of brilliant insights will be blind to essential complexity and therefore miss important cases. Every simplification of ugly domains by programmers, as discussed below, can be either good or bad. If the ugliness results from bad specifications and the programmer's simplification is harmless, then the programmer has made ugly good. But if the domain's complexity is essential (e.g., the income tax code), such "simplifications" constitute bugs. We'll assume (in this section) that domain complexities are essential and can't be removed—but remember, that's still the goal driving designers and testers.

3.4.2. Nonlinear Boundaries

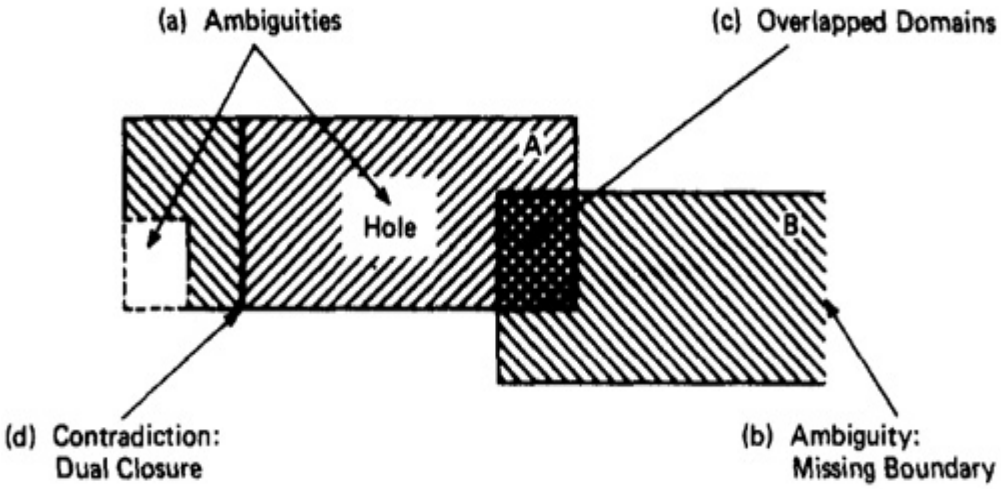
Nonlinear boundaries are so rare in ordinary programming that there's no information on how programmers might "correct" such boundaries if they're essential. Ijust don't see that as a statistically significant source of bugs. If a domain boundary is essentially nonlinear, it's unlikely that programmers will make it linear by accident or by erroneous thinking.

3.4.3. Ambiguities and Contradictions

Figure 6.7 shows several domain ambiguities and contradictions. Remember that although specifications can be ambiguous and/or contradictory, programs can't.

Domain ambiguities are holes in the input space. The holes may lie within domains or in cracks between domains. A hole in a one-variable input space is easy to see. An ambiguity for two variables can be difficult to spot, especially if boundaries are lines going every which way. Ambiguities in three or more variables are almost impossible to spot without formal analysis—which means tools. Because programs can't be ambiguous, programmers must (whether or not they realize it) associate every ambiguity with some domain. Therefore, valid ambiguous cases may be rejected or assigned to the wrong domain and invalid ambiguous cases may be accepted. Because the ambiguity is not consciously realized, we can assume that the result is as likely to be a bug as not. Formal ambiguity detection is provided by specification languages and tools designed for this purpose.

Figure 6.7. Domain Ambiguities and Contradictions.



Two kinds of contradictions are possible: overlapped domain specifications and overlapped closure specifications. Figure 6.7c shows overlapped domains. The programmer's reaction will be to treat the overlap as belonging either to domain A or to domain B. Let's grant that situation a 50-50 bug chance. The tester faced with an unrecognized domain contradiction similarly assigns the overlapped area to one or the other domains. The result is an almost certain argument between designer and tester, when in fact both are wrong, as is the specification.

Figure 6.7d shows a dual closure assignment. This is actually a special kind of overlap. It looks different because we're conscious of the difference between one and two dimensions. The general case, of *n*-dimensional domains means boundaries of $n - 1$ or fewer dimensions—that is, boundary hyperplanes. The two cases (Figures 6.7c and 6.7d) are just overlapped (contradictory) domains with a two-dimensional overlap and a one-dimensional overlap, respectively.

3.4.4. Simplifting the Topology

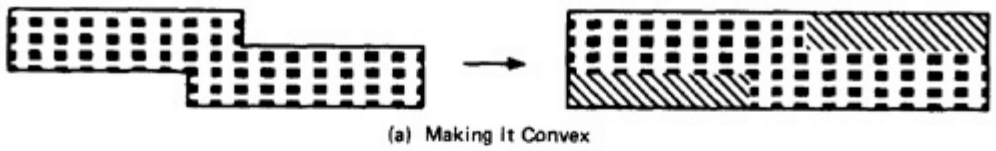
The programmer's and tester's reaction to complex domains is the same—simplify. There are three generic cases: concavities, holes, and disconnected pieces. Programmers introduce bugs and testers misdesign test cases by: smoothing out concavities (Figure 6.8a), filling in holes (Figure 6.8b), and joining disconnected pieces (Figure 6.8c). Connecting disconnected boundary segments and extending boundaries out to infinity are other ways to simplify the domain's topology. Overlooking special cases or merging supposedly equivalent cases (as a result of a "great" but faulty insight) does one or more of these things.

The negation of a compound predicate with ANDs is always a compound predicate with ORs—and these predicates can create complex domain topologies. If you have a compound predicate, there's nothing simple about inverting the decision. If you "simplify" the program by inverting decisions, are you sure that you haven't inadvertently simplified essential domain complexity?

3.4.5. Rectifying Boundary Closures

If domain boundaries are parallel but have closures that go every which way (left, right, left, . . .) the natural reaction is to make closures go the same way (see Figure 6.9). If the main processing concerns one or two domains and the spaces between them are to be rejected, the likely treatment of closures is to make all boundaries point the same way. For example, every bounding hyperplane is forced to belong to the domain or every bounding hyperplane is forced outside of the domain. Again, it's the programmers'

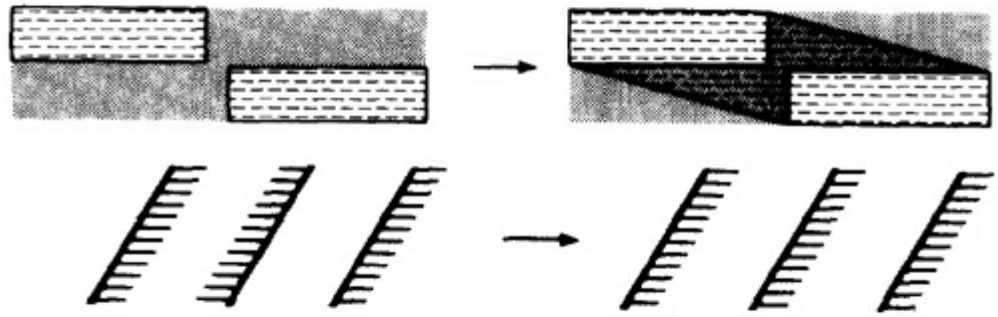
and testers' search for simple rules to cover all cases that results in consistent but incorrect closures.



(a) Making it Convex

(b) Filling in the Holes

Figure 6.8. Simplifying the Topology.



(a) Consistent Direction

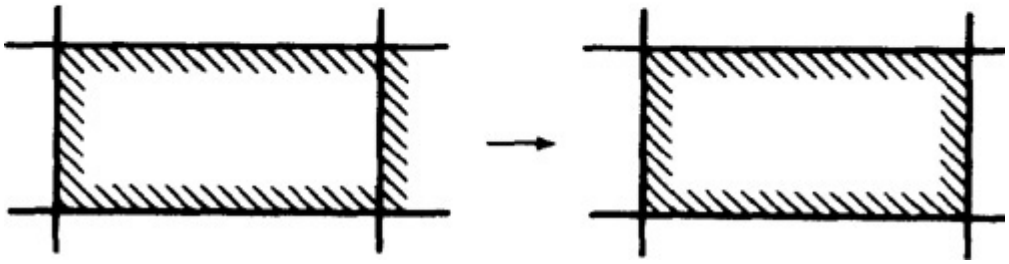


Figure 6.9. Forcing Closure Consistency.

4. DOMAIN TESTING (WHIT78A, WHIT80A, WHIT85B, WHIT87)

4.1. Overview

The domain-testing strategy is simple, albeit possibly tedious.

1. Domains are defined by their boundaries; therefore, domain testing concentrates test points on or near boundaries.
2. Classify what can go wrong with boundaries, then define a test strategy for each case. Pick enough points to test for all recognized kinds of boundary errors.
3. Because every boundary serves at least two different domains, test points used to check one domain can also be used to check adjacent domains. Remove redundant test points.
4. Run the tests and by posttest analysis (the tedious part) determine if any boundaries are faulty and if so, how.
5. Run enough tests to verify every boundary of every domain.

4.2. Domain Bugs and How to Test for Them

4.2.1. General

An **interior point** (Figure 6.10) is a point in the domain such that all points within an arbitrarily small distance (called an **epsilon neighborhood**) are also in the domain. A **boundary point** is one such that within an epsilon neighborhood there are points both in the domain and not in the domain. An **extreme point** is a point that does not lie between any two other arbitrary but distinct points of a (convex) domain.

An **on point** is a point on the boundary. If the domain boundary is closed, an **off point** is a point near the boundary but in the adjacent domain. If the boundary is open, an off point is a point near the boundary but in the domain being tested; see Figure 6.11. You can remember this by the acronym COOOOI: **C**losed **O**ff **O**utside, **O**pen **O**ff **I**nside.

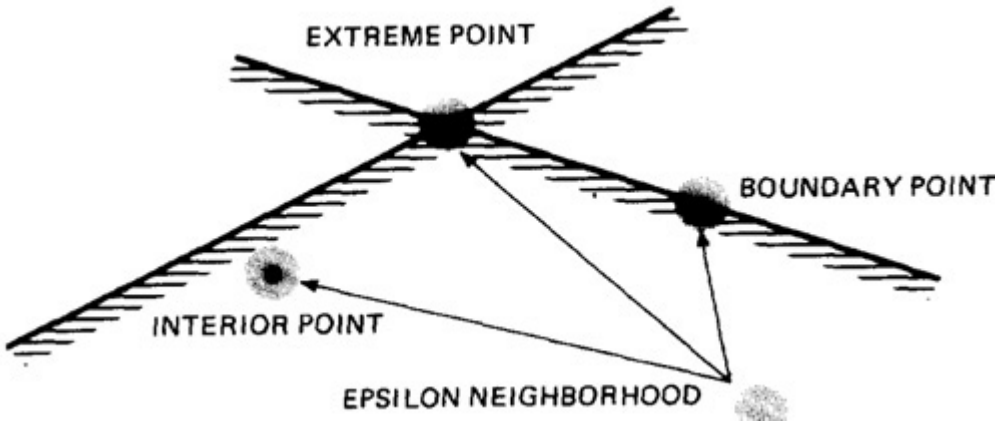


Figure 6.10. Interior, Boundary, and Extreme Points.

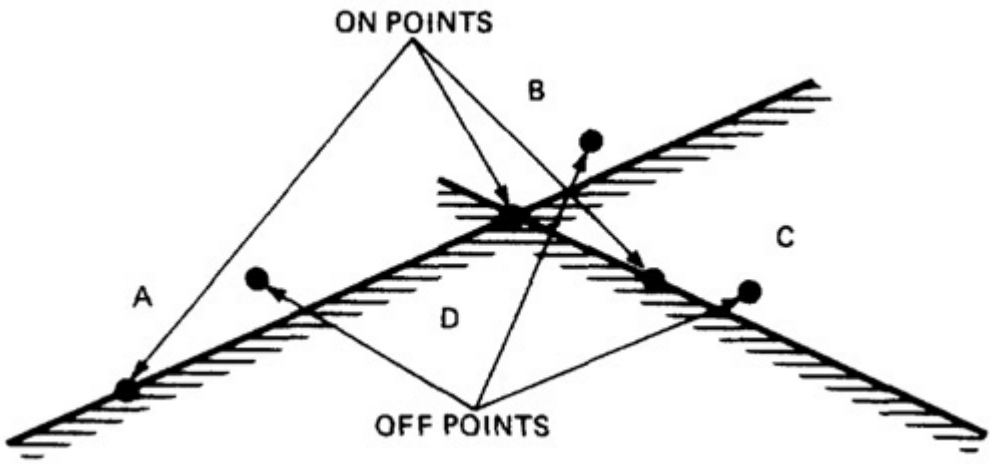


Figure 6.11. On Points and Off Points.

Figure 6.12 shows generic domain bugs: closure bug, shifted boundaries, tilted boundaries, extra boundary, missing boundary.

4.2.2. Testing One-Dimensional Domains

Figure 6.13 shows possible domain bugs for a one-dimensional open domain boundary. The closure can be wrong (i.e., assigned to the wrong domain) or the boundary (a point in this case) can be shifted one way or the other, we can be missing a boundary, or we can have an extra boundary. In Figure 6.13a we assumed that the boundary was to be open for A. The bug we're looking for is a closure error, which converts > to >= or < to <= (Figure 6.13b). One test (marked x) on the boundary point detects this bug

because processing for that point will go to domain A rather than B.

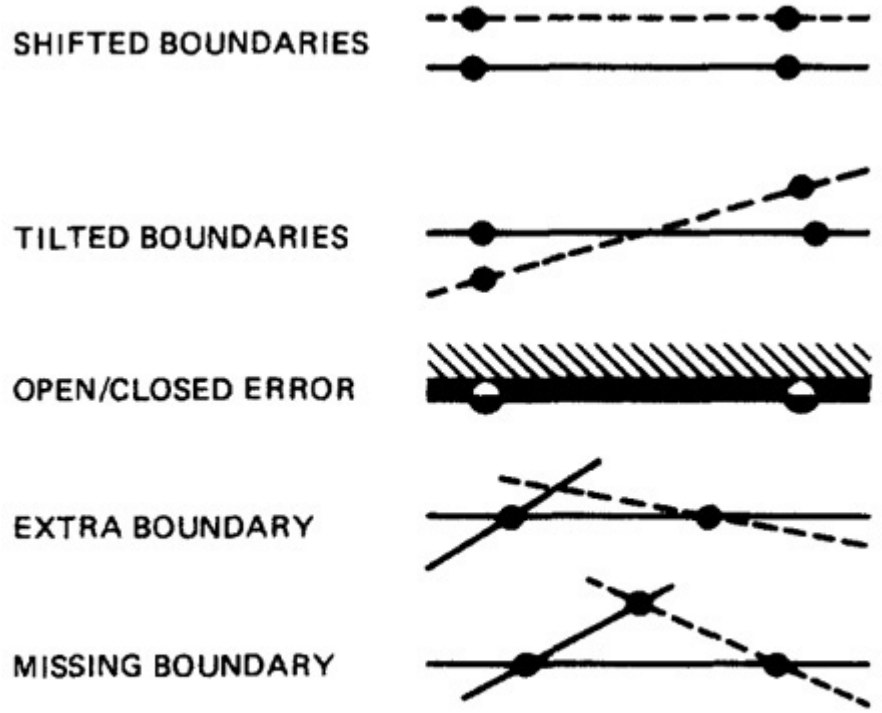


Figure 6.12. Generic Domain Bugs.

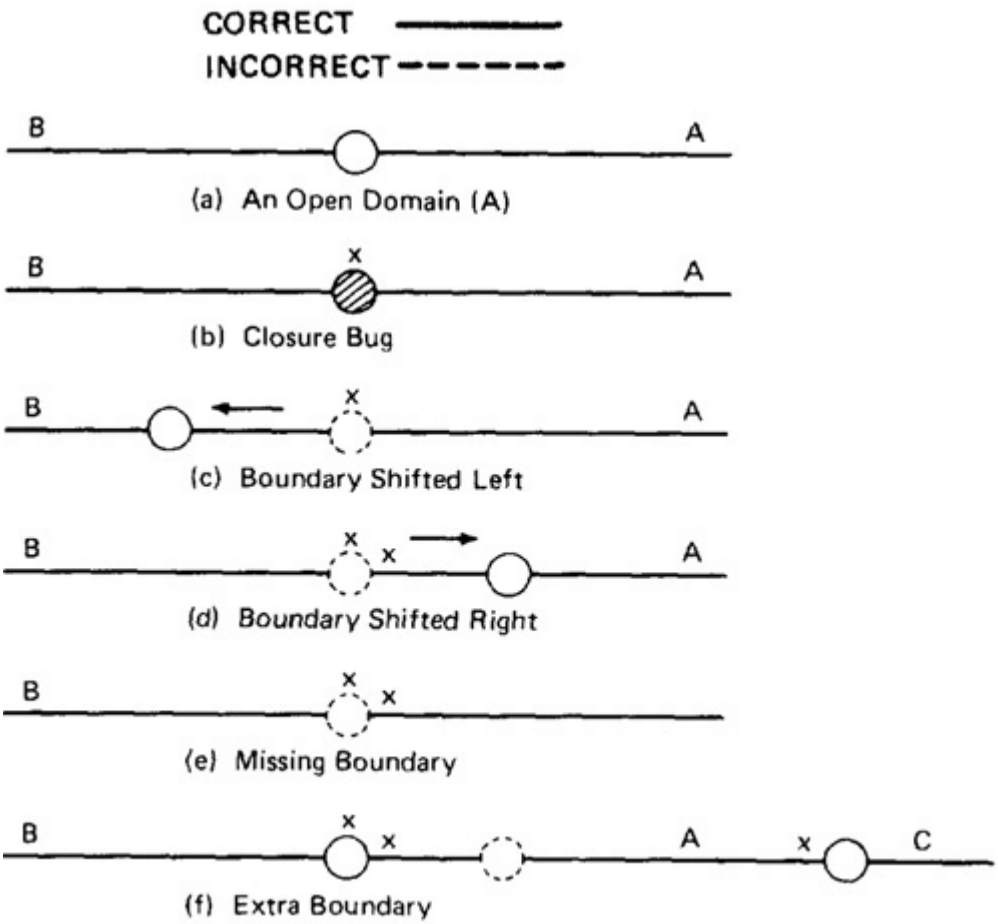


Figure 6.13. One-Dimensional Domain Bugs, Open Boundaries.

In Figure 6.13c we've suffered a boundary shift to the left. The test point we used for closure detects this bug because the bug forces the point from the B domain, where it should be, to A processing. Note that we can't distinguish between a shift and a closure error, but we do know that we have a bug.

Figure 6.13d shows a shift the other way. The on point doesn't tell us anything because the boundary shift doesn't change the fact that the test point will be processed in B. To detect this shift we need a point close to the boundary but within A. The boundary is open, therefore by definition, the off point is in A (**O**pen **O**ff **I**nside). This point also suffices to detect a missing boundary because what should have been processed in A is now processed in B. To detect an extra boundary we have to look at two domain boundaries. In this context an extra boundary means that A has been split in two. The two off points that we selected before (one for each boundary) does the job. If point C had been a closed boundary, the on test point at C would do it.

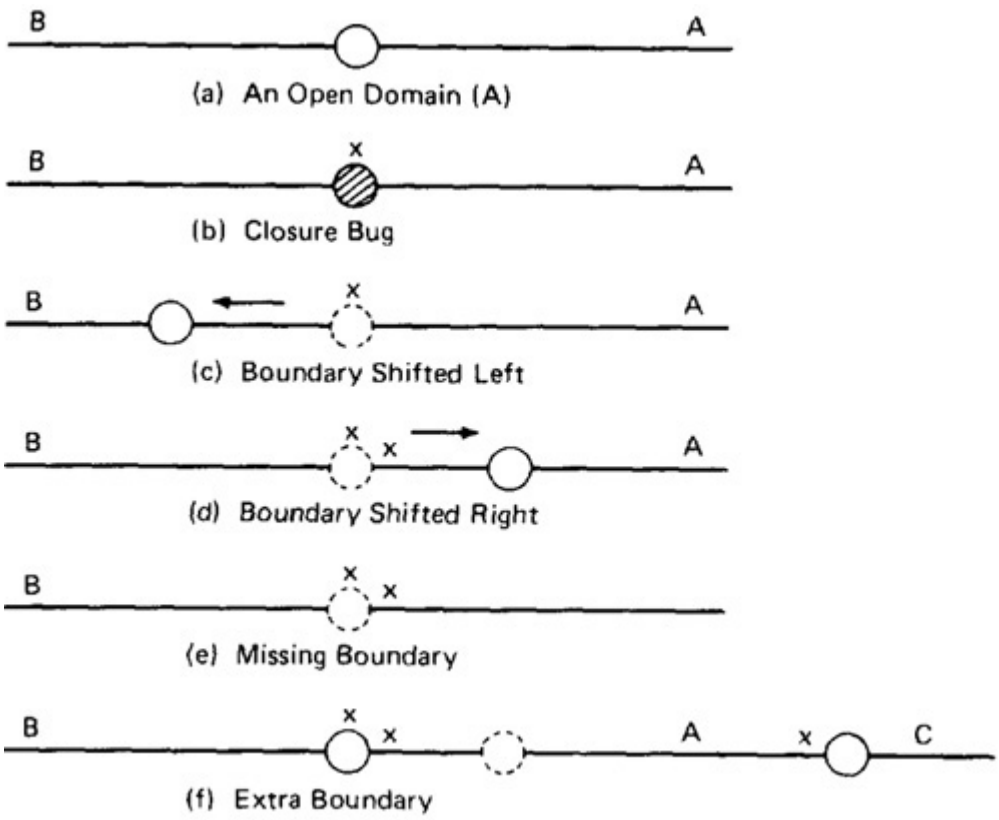


Figure 6.14. One-Dimensional Domain Bugs, Closed Boundaries.

For closed domains look at Figure 6.14. As for the open boundary, a test point on the boundary detects the closure bug. The rest of the cases are similar to the open boundary, except now the strategy requires off points just outside the domain.

4.2.3. Testing Two-Dimensional Domains

Figure 6.15 shows domain boundary bugs for two-dimensional domains. A and B are adjacent domains and the boundary is closed with respect to A, which means that it is open with respect to B. We'll first discuss cases for closed boundaries; turn the figure upside down to see open boundary cases.

1. *Closure Bug*—Figure 6.15a shows a faulty closure, such as might be caused by using a wrong operator (for example, $x >= k$ when $x > k$ was intended, or vice versa). The two on points detect this bug because those values will get B rather than A processing.

2. *Shifted Boundary*—In Figure 6.15b the bug is a shift up, which converts part of domain B into A processing, denoted by A'. This result is caused by an incorrect constant in a predicate, such as $x + y >= 17$ when $x + y >= 7$ was intended. The off point (closed off outside) catches this bug. Figure 6.15c shows a shift down that is caught by the two on points.

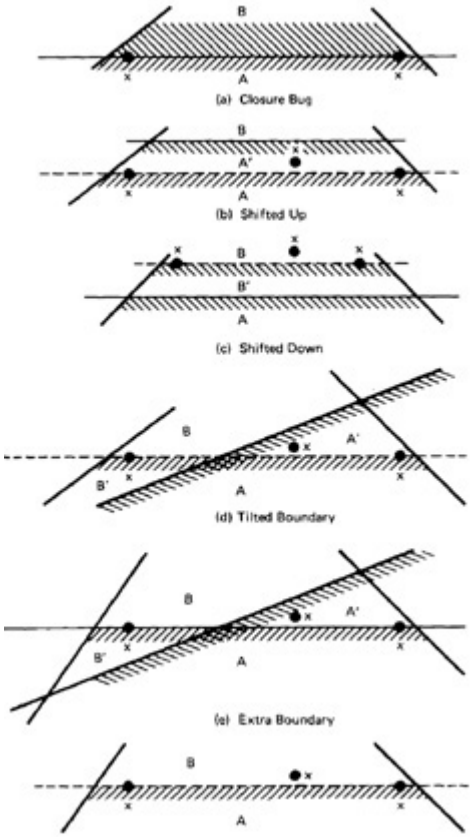


Figure 6.15. Two-Dimensional Domain Bugs.

3. *Tilted Boundary*—A tilted boundary occurs when coefficients in the boundary inequality are wrong. For example, $3x + 7y > 17$ when $7x + 3y > 17$ was intended. Figure 6.15d has a tilted boundary, which creates erroneous domain segments A' and B'. In this example the bug is caught by the left on point. Try tilting your pencil along the boundary in various ways and you'll see that every tilt can be caught by the fact that some point or points go to the wrong domain.

4. *Extra Boundary*—An extra boundary is created by an extra predicate. An extra boundary will slice through many different domains and will therefore cause many test failures for the same bug. The extra boundary in Figure 6.15e is caught by two on points, and depending on which way the extra boundary goes, possibly by the off point also. Note that this extra boundary must result in different functions for A' versus A and B' versus B. If it doesn't—i.e., $f(A) = f(A')$ and/or $f(B) = f(B')$—there is a redundant and useless predicate but no actual bug insofar as functional testing is concerned.

5. *Missing Boundary*—A missing boundary is created by leaving a boundary predicate out. A missing boundary will merge different domains and, as the extra boundary can, will cause many test failures although there is only one bug. A missing boundary, shown in Figure 6.15f, is caught by the two on points because the processing for A and B is the same—either A or B processing. You can't detect a missing boundary if your domain testing is based on the implementation because no structural test technique can detect a missing path. In detecting a missing boundary you are comparing a specified domain, for which you select test points, with the implemented domains.

Turn the figure upside down and consider the B domain, which is open. The off point is correct, by the definition of on and off points, and the discussion is essentially the same as before.

Figure 6.16 summarizes domain testing for two-dimensional domains; it shows a domain all but one of whose boundaries are closed. There are two on points (closed circles) for each segment and one off point (open circles). That's all the testing needed to do domain testing in two dimensions. Note that the selected test points are shared with adjacent domains, so the number of cases needed is reduced. To test a two-dimensional domain whose boundaries may be incomplete, with s boundary segments, we need at least 2*s* test points if we can share them with other domains and at most 3*s* points if we can't.

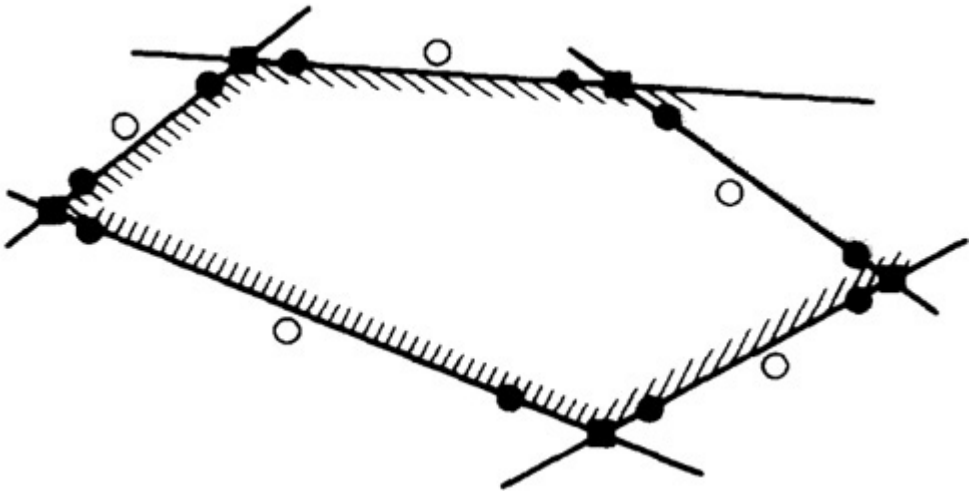


Figure 6.16. Domain-Testing Strategy Summary.

The on points for two adjacent boundary segments can also be shared if both those segments are open or if both are closed (see Figure 6.17). A single extreme point replaces the two on points near the boundary intersection. If two adjacent boundary segments aren't both open or both closed, then two different on points are needed. Try different bug cases to see this.

Although the above domain-testing strategy doesn't require us to test the intersections of the boundaries (the **extreme points**), it's a good idea to do so because bugs tend to congregate in corners. If the basic strategy (for two dimensions) of two on points and one off point per boundary segment shows the boundaries to be correct, what can we expect to learn from intersection testing? Look for a blowup, crash, or endless loop. Testing extreme points doesn't require much extra work because you had to find them in order to know where to put the on points.

Domain testing is blind to small domain errors. Whatever off points we pick, they are within an epsilon neighborhood of the boundary. A shift of less than this value or a small enough tilt will not be caught. Other refinements of domain testing, especially for small errors and discrete spaces, are discussed in WHIT78A: read that paper if you're building a domain-testing tool.

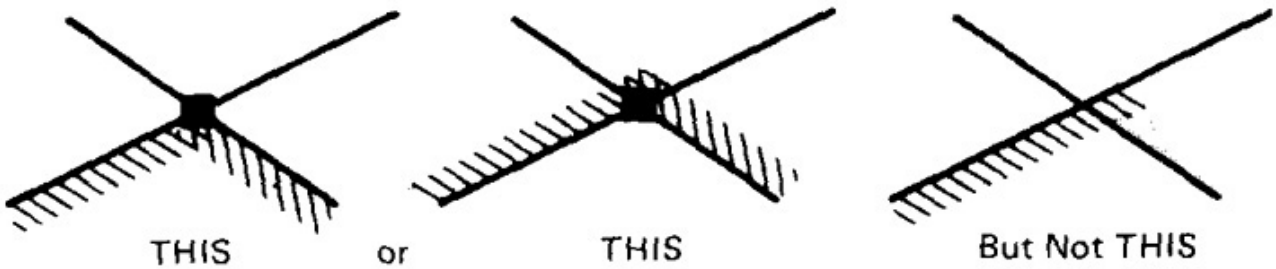


Figure 6.17. Shared on Points.

4.2.4. Equality and Inequality Predicates

Equality predicates such as $x + y = 17$ define lower-dimensional domains. For example, if there are two input variables, a two-dimensional space, an equality predicate defines a line—a one-dimensional domain. Similarly, an equality predicate in three dimensions defines a planar domain. We get test points for equality predicates by considering adjacent domains (Figure 6.18). There are three domains. A and B are planar while C, defined by the equality boundary predicate between A and B, is a line. Applying the two-on, one-off rule to domains A and B and remembering **O**pen **O**ff **I**nside, we need test point *b* for B and point *a* for A and two other points on C (*c* and *c*') for the on points. This is equivalent to testing C with two on and two off points. There is a pathological situation in which the bug causes the boundary to go through the two selected off points: it can be detected by another on point at the intersection between the correct domain line and the two off points (marked *d*). Because such bugs require careful design and because our failure to catch them depends on an unfortunate choice of the two off points, they can be ignored.

4.2.5. Random Testing?

Look at Figure 6.16. Let's add one typical test case someplace in the middle of the domain to verify the computation (not really needed). Domain testing, especially when it incorporates extreme points, has two virtues: it verifies domain boundaries efficiently, and many selected test cases (on points and extreme points) correspond to cases where experience shows programmers have trouble. Consider Figure 6.16. What is the probability that a set of randomly chosen test points will meet the above criteria? End of argument against random testing.*

*Not really—random test points have value when used to determine statistically valid notions of confidence: e.g., what is the probability that this routine will fail in use? See DURA84 and HAML88. Random test data though, as a means for breaking software or catching bugs, is not effective, and even its proponents suggest augmenting it with domain boundary and extreme points: but see JENG89 for conditions under which random testing is either better or worse.

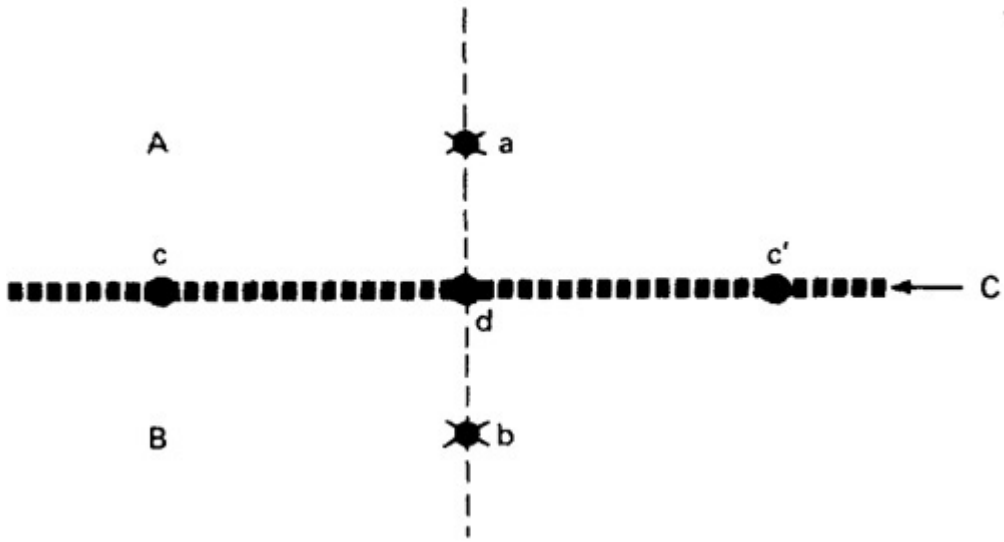


Figure 6.18. Equality Predicates.

4.2.6. Testing n-Dimensional Domains

For domains defined over an *n*-dimensional input space with *p* boundary segments, the domain testing strategy generalizes to require at most $(n + 1)p$ test points per domain, consisting of *n* on points and one

off point (see WHIT78A). When extreme point sharing is possible, the number of points required per domain can be as little as $2p$ per domain. Domain boundaries which are orthogonal to the coordinate axes, have consistent closures, and extend over the entire number space can be tested independently of other boundaries. Equality predicates defined over m dimensions ($m < n$) create a subspace of $n - m$ dimensions, which is then treated the same way as general n-dimensional domains.

4.3. Procedure

The procedure is conceptually straightforward. It can be done by hand for two dimensions and a few domains. Without tools the strategy is practically impossible for more than two variables.

1. Identify input variables.
2. Identify variables which appear in domain-defining predicates, such as control-flow predicates.
3. Interpret all domain predicates in terms of input variables. We'll assume that the interpreted predicates are linear or can be easily transformed to linear (see Section 6.2 below). Note that interpretation need not be based on actual paths. For example, if all variable assignments that lead to an uninterpreted predicates are specified, then interpretation can be based on a "path" through this set of specifications. This path may or may not correspond to implemented paths. In other words, the paths we want are not paths in the control flowgraph but paths in the specified data flowgraph.
4. For p binary predicates, there are at most 2^p combinations of TRUE-FALSE values and therefore, at most 2^p domains. Find the set of all non-null domains (see Chapter 10). The result is a boolean expression in the predicates consisting of a set of AND terms joined by ORs—for example, ABC + DEF + GH + IJKL, where the capital letters denote predicates. Each **product term** (that is, term consisting of predicates joined by ANDs) is a set of linear inequalities that defines a domain or a part of a multiply connected domain.
5. Solve these inequalities (see a book on linear algebra or linear programming) to find all the extreme points of each domain. White (WHIT86) discusses a method called "the scattering method" based on linear programming that yields a usable set of test points.
6. Use the extreme points to solve for nearby on points and to locate midspan off points for every domain.

More details on how to (build tools that) determine test points can be found in PERE85 and WHIT86. If it seems to you that you are rapidly becoming involved in linear algebra, simultaneous-equation solving, linear programming, and a lot of supporting software, you're right. I said at the outset that this was a tool-intensive strategy, but don't feel cheated. What you should ask yourself is: "Did the person who specified these domains, their closures, their extreme points, and all the rest—did that person go through the above analysis in order to verify the validity of the specified domains? Did she have the foggiest notion of the complexity she specified?"

4.4. Variations, Tools, Effectiveness

Domain testing, although it's been around for over a decade, has not progressed as far toward day-to-day application as other strategies. Variations have been explored that vary the number of on and off points and/or the extreme points. The basic domain testing strategy discussed here is called the $N \times 1$ strategy because it uses N on points and one off point. Clarke et al. (CLAR82, RICH85) discuss strategies that use N on and N off points per boundary segment ($N \times N$ strategy) and strategies based on extreme points and off points near extreme points ($V \times V$ strategy, V for "vertex"). As with all testing techniques, many variations are possible if you consider the way to pick on points, off points, and extreme points. Also, as we'll see in Section 6, it's possible to weaken the basic strategy and to consider boundary hyperplanes as

the basic thing to test rather than domain boundary segments caused by intersections of those hyperplanes. As with all test techniques, more research will lead to sharper strategies that find more bugs with fewer tests and a better understanding of how the variations relate to one another.

We don't have many statistics on the cost-effectiveness of domain testing. Richardson and Clarke (RICH85) discuss a generalization that they call "partition analysis," which includes domain testing, computation verification, and both structural and functional information. Their methodology applied to the Kernighan and Plauger and other test sets shows promise in that almost all bugs were caught.

The present tool situation is par for the course. Some specification-based tools such as T (PROG88), use heuristic domain-testing principles based on the observation that bugs are likelier at the extreme points of domains. Most researchers in domain testing have built experimental tools to assist them, as with other test techniques, but to date there are no commercial tools that incorporate strict domain testing.

The fact that domain testing is tool-intensive should not be a barrier to its effective exploitation. An appropriate perspective can be gained by looking at the related field of hardware logic testing, which is also tool-intensive. Logic testers are ahead of software testers because they've been at it longer and because their problem is simpler. The operative mode of hardware testing is extremely tool-intensive. Why should we expect, why should we restrict, software testing methods to what can be done by hand? After all, what *are* computers for?

5. DOMAINS AND INTERFACE TESTING

5.1. General

Don't be disappointed by domain testing because it's difficult to apply to two dimensions and humanly impossible for more than two. Don't reject it because it seems that all a person can do is handle one dimension at a time and you can't (yet) buy tools to handle more than one dimension. The conceptual vocabulary of domain testing, even one variable at a time, has a lot to offer us in integration testing. Recall that we defined integration testing as testing the correctness of the interface between two otherwise correct components. Components A and B have been demonstrated to satisfy their component tests, and as part of the act of integrating them we want to investigate possible inconsistencies across their interface. Although the interface between two components can be a subroutine call, shared data objects, values left in registers or global variables, or some combination thereof, it's convenient to talk about the interface as if it is a subroutine call. We're looking for bugs in that "call" when we do interface testing. Another way to put this is to say that we're looking for incompatible notions of what constitutes "good" values in the call. Let's assume that the call sequence is correct and that there are no type incompatibilities. What's left is a possible disagreement about domains—the variables' domain as seen by the caller and the called routine. For a single variable, the **domain span** is the set of numbers between (and including) the smallest value and the largest value. For every input variable we want (at least): compatible domain spans and compatible closures. Note that I said "compatible" and not "equal." As we'll see, domains need not be equal to be compatible.

5.2. Domains and Range

The set of output values produced by a function is called the **range** of the function, in contrast with the **domain**, which is the set of input values over which the function is defined. In most of testing we are only minimally concerned with output values. We have to examine them to confirm that the correct function has been implemented, but other than that, we place no restrictions on output values. Thus, for most testing, our aim has been to specify input values and to predict and/or confirm output values that

result from those inputs. Interface testing requires that we select the output values of the calling routine. A more precise statement of the kind of compatibility needed between caller and called is to say that the caller's range must be compatible with the called routine's domain. An interface test consists of exploring the correctness of the following mappings:

caller domain → caller range (caller unit test)
caller range → called domain (integration test)
called domain → called range (called unit test)

Another way to put this is to say that the caller's range is the caller's notion of the called routine's domain. We'll usually be looking at things from the point of view of the called routine and therefore be talking in terms of domains.

5.3. Closure Compatibility

Assume that the caller's range and the called domain spans the same numbers—say, 0 to 17. Figure 6.19 shows the four ways in which the caller's range closure and the called's domain closure can agree. I've turned the number line on its side. The thick line means closed and the thin line means open. Figure 6.19 shows the four cases consisting of domains that are closed both on top (17) and bottom (0), open top and closed bottom, closed top and open bottom, and open top and bottom. Figure 6.20 shows the twelve different ways the caller and the called can disagree about closure. Not all of them are necessarily bugs. The four cases in which a caller boundary is open and the called is closed (marked with a "?") are probably not buggy. It means that the caller will not supply such values but the called can accept them. Note that inconsistencies even if harmless in one domain may be erroneous in adjacent domains. If there are only two domains (valid, not valid) and we're looking at the valid domain, then the inconsistency is harmless; otherwise it's a bug.

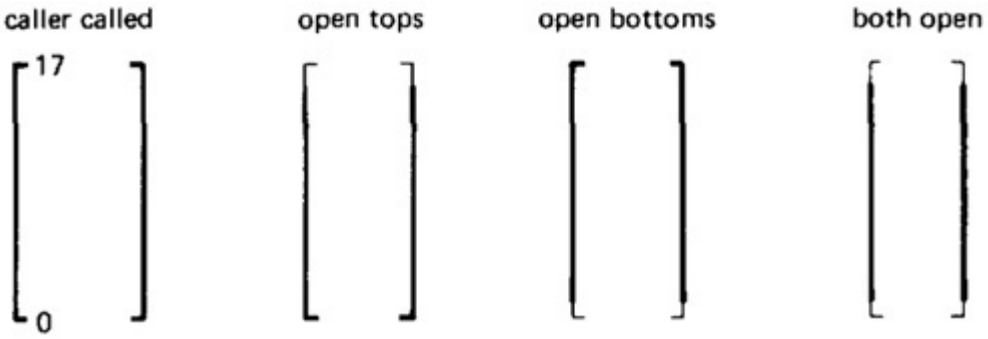


Figure 6.19. Range/Domain Closure Compatibility.

5.4. Span Compatibility

Figure 6.21 shows three possibly harmless span incompatibilities. I've eliminated closure incompatibilities to simplify things. In all cases, the caller's range is a subset of the called's domain. That's not necessarily a bug. Consider, for example, a square-root routine that accepts negative numbers and provides an imaginary square root for them. The routine is used by many callers; some require complex number answers and some don't. This kind of span incompatibility is a bug only if the caller expects the called routine to validate the called number for the caller. If that's so, there are values that can be included in the call, which from the caller's point of view are invalid, but for which the called routine will not provide an error response.

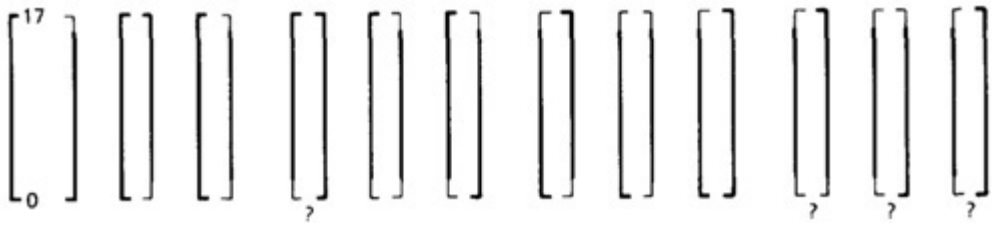


Figure 6.20. Equal-Span Range/Domain Compatibility Bugs.

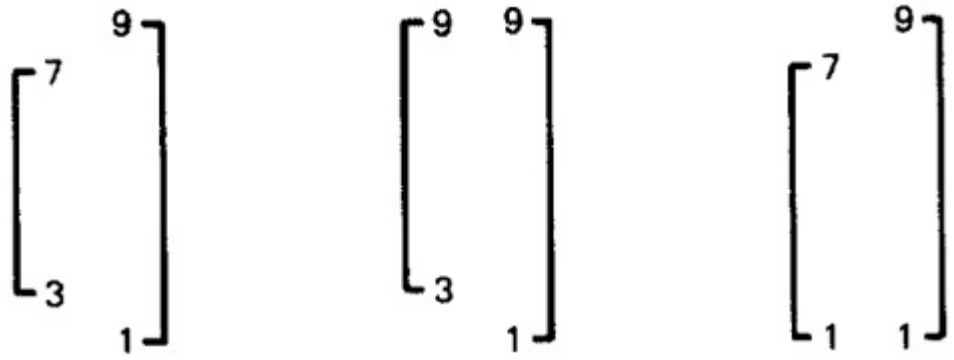


Figure 6.21. Harmless Range/Domain Span Incompatibility Bug. (Caller Span Is Smaller Than Called.)

Figure 6.22a shows the opposite situation, in which the called routine's domain has a smaller span than the caller expects. All of these examples are buggy. In Figure 6.22b the ranges and domains don't line up; hence good values are rejected, bad values are accepted, and if the called routine isn't robust enough, we have crashes. Figure 6.22c combines these notions to show various ways we can have holes in the domain: these are all probably buggy.

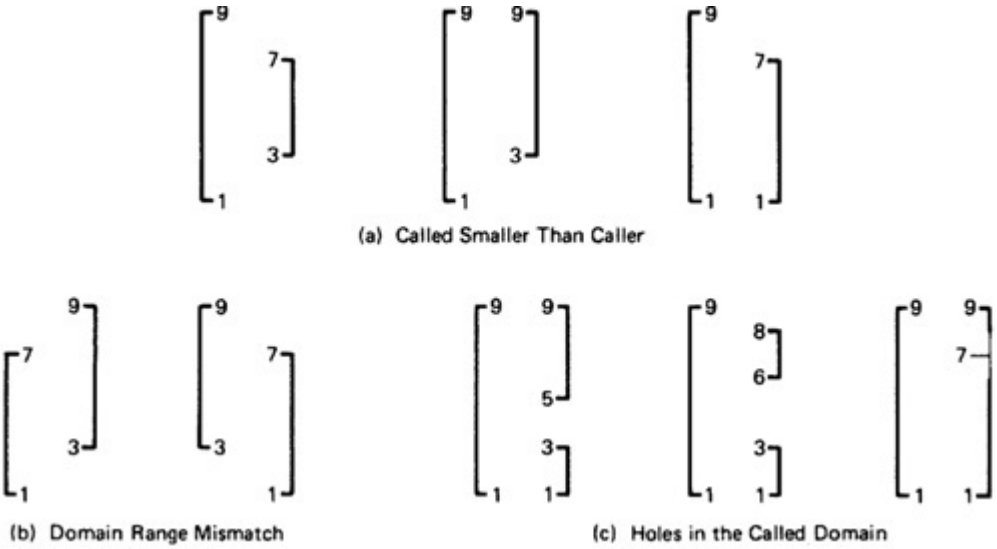


Figure 6.22. Buggy Range/Domain Mismatches.

5.5. Interface Range/Domain Compatibility Testing

I tried to generalize caller/called range/domain incompatibilities by first considering combinations of closure and span errors for one variable. That led to a vast combination of cases that were no more revealing than those shown above. I curbed my ambition and tried to picture all possible range/domain disagreements for two variables simultaneously. The result would have been called (had I persisted) *Beizer's Picture Book of Two-Dimensional Range/Domain Incompatibilities*. It was a fruitless and pointless exercise because for interface testing, bugs are more likely to concern single variables rather than peculiar combinations of two or more variables. Without automation, it's impractical to even

consider double-error cases, never mind higher dimensions. That's why we test one variable at a time.

Test every input variable independently of other input variables to confirm compatibility of the caller's range and the called routine's domain span and closure of every domain defined for that variable. For subroutines that classify inputs as "valid/invalid" the called routine's domain span and closure (for valid cases) is usually broader than the caller's span and closure. The reason is that common subroutines serve more than one mistress and must accept a span of values equal to the union of all caller's ranges for that variable. Therefore, the burden of providing valid inputs is always on the caller, not the called. There are two boundaries to test and it's a one-dimensional domain; therefore, it requires one on and one off point per boundary or a total of two on points and two off points for the domain—pick the off points appropriate to the closure (COOOOI).

If domains divide processing into several cases, you must use normal domain-testing ideas and look for exact range/domain boundary matches (span and closure). You should be able to lump all valid cases into a contiguous "superdomain" so that this set can be tested for compatibility with the called routine's valid set. Then it might be possible that mere compatibility, rather than an exact match, will suffice for the top and bottom domains. The caller range's lower bound must be compatible only with the called domain's lower bound—similarly for the caller's and called's range/domain upper bounds.

5.6. Finding the Values

Start with the called routine's domains and generate test points in accordance to the domain-testing strategy used for that routine in component testing. A good component test should have included all the interesting domain-testing cases, and as a consequence there's no need to repeat the work. Those test cases are the values for which you must find the input values of the caller. If the caller's domains are linear, then this is another exercise in solving inequalities. Some things to consider:

1. The solution may not be unique. That doesn't matter because you need *any* caller input that will produce the required output.
2. There may be no solution for the specific points you need. In that case you'll probably have to make do with the maximum and minimum values that can be produced as outputs by the caller. Note that if the caller's domains are ugly (nonlinear, incomplete boundaries, unsystematic, not orthogonal, and so on), the problem can be very difficult. For nice domains, you do it one variable at a time and that's no big deal.
3. In general, you must find and evaluate *an* inverse function for the caller. Note "*an* inverse" rather than "*the* inverse."
4. Unless you're a mathematical whiz you won't be able to do this without tools for more than one variable at a time. We have very little data on the frequency of interface range/domain incompatibility bugs for single variables on no data on the frequencies of such bugs for several variables simultaneously. Consequently, there's no point in heavy investment in complicated multivariable tests whose effectiveness is unknown.

6. DOMAINS AND TESTABILITY

6.1. General

The best way to do domain testing is to avoid it by making things so simple that it isn't needed. The conceptual vocabulary of domain testing is a rich source of insights into design. We know what makes domain testing trivial: orthogonal domain boundaries, consistent closure, independent boundaries, linear boundaries, and the other characteristics discussed in Section 3.3. We know what makes domain testing

difficult. What can we, as designers, do? Most of what we can do consists of applying algebra to the problem.

6.2. Linearizing Transformations

In the unlikely event that we're faced with essential nonlinearity we can often convert nonlinear boundaries to equivalent linear boundaries. This is done by applying **linearizing transformations**. The methods are centuries old. Here is a sample.

1. Polynomials—A boundary is specified by a polynomial or multinomial in several variables. For a polynomial, each term (for example, x, x^2, x^3, . . .) can be replaced by a new variable: $y_1 = x$, $y_2 = x^2$, $y_3 = x^3$, ... For multinomials you add more new variables for terms such as xy, x^2y, xy^2, ... You're trading the nonlinearity of the polynomial for more dimensions. The advantage is that you transform the problem from one we can't solve to one for which there are many available methods (e.g., linear programming for finding the set of on and off points.)

2. *Logarithmic Transforms*—Products such as xyz can be linearized by substituting $u = \log(x)$, $v = \log(y)$, $w = \log(z)$. The original predicate ($xyz > 17$, say) now becomes $u + v + w > 2.83$.

3. *More General Transforms*—Other linearizable forms include $x/(ax + b)$ and ax^b. You can also linearize (approximately) by using the Taylor series expansion of nonlinear functions, which yields an infinite polynomial. Lop off the low-order terms to get an approximate polynomial and then linearize as for polynomials. There's a rich literature on even better methods for finding approximating polynomials for functions.

6.3. Coordinate Transformations

Nice boundaries come in parallel sets. Parallel boundary sets are sets of linearly related boundaries: that is, they differ only by the value of a constant. Finding such sets is straightforward. Its an $O(n^2)$ procedure for n boundary equations. Pick a variable, say x. It has a coefficient a_i in inequality i. Divide each inequality by it's x coefficient so that the coefficients of the transformed set of inequalities is unity for variable x. If two inequalities are parallel, then all the coefficients will be the same and they will differ only by a constant. A systematic comparison will find the parallel sets, if any. There are more efficient algorithms, and any serious tool builder would do well to research the issue. We now have parallel sets of inequalities. For each such set, pick one representative, any one, and put the rest aside for now; but keep track of how many different boundaries each represents. We now have a bunch of linear inequalities that are not parallel to one another.

The next objective is to extract, from the set of nonparallel inequalities determined above, a subset that can, by suitable coordinate transformations, be converted into a set of orthogonal boundary inequalities. If you'll recall, the reason for doing this is to obtain a new set of variables in terms of which the inequalities can be tested one at a time, independently of the other inequalities. Generally, we have more inequalities (n, say) than we have variables (m, say) so we'll not be able to pack them all into an m-dimensional space such that they are all independent. But we can pick m of them. I don't claim the following to be the best possible way—only a reasonable heuristic. Convert the inequalities to equalities—we're interested in the boundaries only, and they're specified by equalities. Start with the equation that represents the most boundaries, then pick the equation that represents the next most boundaries, etc. Continue adding equations until they're all used up or until you have m of them. After you select the equations, you apply a procedure called Gram-Schmidt orthogonalization (see KOLM88), which transforms your original set of variables X into a new, orthogonal set of variables U. For these new variables the inequalities are of the form $u_i >= k_i$, which you'll agree is easy to test. One test per

hyperplane confirms the on point, another test confirms the inequality's direction. If the hyperplane represents several parallel hyperplanes, then you'll need two tests each to confirm them. There's no need to test intersections with the other nonparallel hyperplanes if funny things aren't happening to the boundary segments—that is, if the inequality's direction applies to the entire boundary hyperplane and the hyperplane extends to plus and minus infinity with the same closure all the way. But your choice of the *m* hyperplanes was arbitrary. Furthermore, parallel hyperplanes had been eliminated, so that any *m* could have been transformed to an orthogonal coordinate set. Therefore, as my mathematician friend Tom Seidman pointed out, you didn't have to test the boundary segments in the first place and you didn't have to do the transformation. In other words, given a (nice) *k*-dimensional hyperplane, its correctness can be confirmed by $k + 1$ tests, no matter how many domains it splits the world into. We need another test for the off point, for a total of $k + 2$ tests per hyperplane.

The $N \times 1$ domain-testing strategy is a worst-case strategy that allows arbitrary things to be done to the closure of each and every domain boundary segment. If we postulate reasonable bugs (as contrasted to carefully designed bugs) testing requirements are considerably reduced. As with all strategies, reducing the strength of the strategy by leaving out test cases increases your vulnerability to certain kinds of bugs. If those bugs cannot be reasonably expected, then nothing of value has been sacrificed.

6.4. A Canonical Program Form

Go back to Figure 6.1 and consider it now as a design objective. I'll make a few changes to it, though, starting with a variable transformation that linearizes and orthogonalizes as many inequalities as possible in order to simplify testing. The routine's structure then looks something like this:

1. Input the data.
2. Apply linearizing transforms to as many predicates as possible.
3. Transform to an orthogonal coordinate system.
4. For each set of parallel hyperplanes in the orthogonal space, determine the case by a table lookup or by an efficient search procedure (e.g., binary halving) to put the value of that variable into the right bin.
5. Test the remaining inequalities (those that couldn't be brought into the orthogonal space) to determine the required subcase.
6. You can now direct the program to the correct case processing routine by a table lookup or by a tree of control-flow predicates based on the case numbers for each dimension.

Testing is clearly divided into the following: testing the predicate and coordinate transformations; testing the individual case selections (independently of one another); testing the control flow (now over very simple predicates—or table-driven); and then testing the case processing. The resulting routine is probably very big, but also fast. You can compact it by merging various steps, alternating case determination with processing, and so on, but you're starting from a base of a canonical form that, though big, is clear. The bugs don't go away, however, because now you have added the possibility of an erroneously programmed predicate or coordinate transformation. If that's more complicated than using the original variables in control-flow predicates scattered throughout, you might have made things worse. I'm not suggesting that the above canonical form is *the* way programs should be written but that it is a starting point for a rational examination of more design options than you're probably seeing now—to convert a purely intuitive procedure into a more mechanical one. Think about it.

6.5. Great Insights?

Sometimes programmers have great insights into programming problems that result in much simpler programs than one might have expected. My favorite example is the calendar routine. It's easier to start

the year on March 1 rather than January 1 because leap year corrections are then applied to the "last" day of the year. Isn't that just a coordinate change? In retrospect, when we look at some of our productive insights, insights that made a tough programming problem easy, many of them can be explained in terms of a judicious change to a new coordinate system in which the problem becomes easy. There's ample precedence for such things tin other engineering disciplines—a good coordinate systems can break the back of many tough problems. Designers! Instead of waiting for inspiration to grant you the favor of a great insight, go out and look for them. Make the domains explicit. Separate domain definition from processing. Look for transformations to new, orthogonal, coordinate sets that are therefore easy to test. Examine your transformed variables and see whether you can find a clean functionally meaningful interpretation for them—you usually can. Testers! Look for ways to organize tests based on specifications into a minimal set of domain-verifying tests.

7. SUMMARY

1. Programs can be viewed as doing two different things: (a) classifying input vectors into domains, and (b) doing the processing appropriate to the domain. Domain testing focuses on the classification aspect and explores domain correctness.
2. Domains are specified by the intersections of inequalities obtained by interpreting predicates in terms of input variables. If domain testing is based on structure, the interpretation is specific to the control-flow path through the set of predicates that define the domain. If domain testing is based on specifications, the interpretation is specific to the path through a specification data flowgraph.
3. Every domain boundary has a closure that specifies whether boundary points are or are not in the domain. Closure verification is a big part of domain testing.
4. Almost all domain boundaries found in practice are based on linear inequalities. Those that aren't can often be converted to linear inequalities by a suitable linearization transformation.
5. Nice domains have the following properties: linear boundaries, boundaries that extend from plus to minus infinity in all variables, have systematic inequality sets, form orthogonal sets, have consistent closures, are convex, and create domains that are all in one piece. Nice domains are easy to test because the boundaries can be tested one at a time, independently of the other boundaries. If domains aren't nice, examine the specifications to see whether they can be changed to make the boundaries nice; often what's difficult about a boundary or domain is arbitrary rather than based on real requirements.
6. As designers, guard against incorrect simplifications and transformations that make essentially ugly domains nice. As testers, look for such transformations.
7. The general domain strategy for arbitrary convex, simply connected, linear domains is based on testing at most $(n + 1)p$ test points per domain, where n is the dimension of the interpreted input space and p is the number of boundaries in the domain. Of these, n points are on points and one is an off point. Remember the definition of off point—COOOOI.
8. Real domains, especially if they have nice boundaries, can be tested in far less than $(n + 1)p$ points: as little as $O(n)$.
9. Domain testing is easy for one dimension, difficult for two, and tool-intensive for more than two. Beg, borrow, or build the tools before you attempt to apply domain testing to the general situation. Finding the test points is a linear programming problem for the general case and trivial for the nicest domains.
10. Domain testing is only one of many related partition testing methods which includes more points, such as n-on + n-off, or extreme points and off-extreme points. Extreme points are good because bugs tend to congregate there.
11. The domain-testing outlook is a productive tactic for integration interface testing. Test range/domain compatibility between caller and called routines and all other forms of intercomponent communications.
12. Instead of waiting for the programming muse to strike you with a brilliant intuition that will

simplify your implementation, use the canonical model as a mental tool that you can exploit to systematically look for brilliant insights. As a tester, use the canonical program model as a framework around which to organize your test plan.

<ch5 toc ch7>