4

Input Space Partitioning

In a very fundamental way, all testing is about choosing elements from the input space of the software being tested. The criteria presented previously can be viewed as defining ways to divide the input space according to the test requirements. The assumption is that any collection of values that satisfies the same test requirement will be "just as good." Input space partitioning takes that view in a much more direct way. The input *domain* is defined in terms of the possible values that the input parameters can have. The input parameters can be method parameters and global variables, objects representing current state, or user-level inputs to a program, depending on what kind of software artifact is being analyzed. The input domain is then partitioned into regions that are assumed to contain equally useful values from a testing perspective, and values are selected from each region.

This way of testing has several advantages. It is fairly easy to get started because it can be applied with no automation and very little training. The tester does not need to understand the implementation; everything is based on a description of the inputs. It is also simple to "tune" the technique to get more or fewer tests.

Consider an abstract partition q over some domain D. The partition q defines a set of equivalence classes, which we simply call *blocks*, B_q.[1] The blocks are pairwise disjoint, that is

$$b_i \cap b_j = \emptyset, i \neq j; b_i, b_j \in B_q$$

and together the blocks cover the domain D, that is

$$\bigcup_{b \in B_q} b = D$$

This is illustrated in Figure 4.1. The input domain D is partitioned into three blocks, b_1, b_2, and b_3. The partition defines the values contained in each block and is usually designed from knowledge of what the software is supposed to do.

The idea in partition coverage is that any test in a block is as good as any other for testing. Several partitions are sometimes considered together, which, if not done carefully, leads to a combinatorial explosion of test cases.

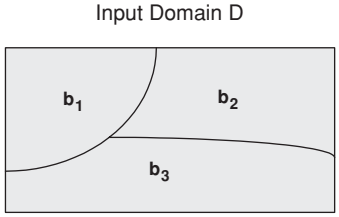

Figure 4.1. Partitioning of input domain *D* into three blocks.

A common way to apply input space partitioning is to start by considering the domain of each parameter separately, partitioning each domain's possible values into blocks, and then combining the variables for each parameter. Sometimes the parameters are considered completely independently, and sometimes they are considered in conjunction, usually by taking the semantics of the program into account. This process is called *input domain modeling* and the next section gives more details.

Each partition is usually based on some *characteristic C* of the program, the program's inputs, or the program's environment. Some possible characteristic examples are:

- Input X is null
- Order of file F (sorted, inverse sorted, arbitrary)
- Min separation distance of two aircraft

Each characteristic C allows the tester to define a partition. Formally, a partition *must* satisfy two properties:

1. The partition must cover the entire domain (completeness)
2. The blocks must not overlap (disjoint)

As an example, consider the characteristic "order of file F" mentioned above. This could be used to create the following (defective) partitioning:

- Order of file F
  - b_1 = Sorted in ascending order
  - b_2 = Sorted in descending order
  - b_3 = Arbitrary order

However, this is **not** a valid partitioning. Specifically, if the file is of length 0 or 1, then the file will belong in all three blocks. That is, the blocks are not disjoint. The easiest strategy to address this problem is to make sure that each characteristic addresses only one property. The problem above is that the notions of being sorted into ascending order and being sorted into descending order are lumped into the same characteristic. Splitting into two characteristics, namely sorted ascending and sorted descending, solves the problem. The result is the following (valid) partitioning of two characteristics.

- File F sorted ascending
  - b_1 = True
  - b_2 = False

- File F sorted descending
  - b_1 = True
  - b_2 = False

With these blocks, files of length 0 or 1 are in the True block for both characteristics.

The completeness and disjointness properties are formalized for pretty pragmatic reasons, and not just to be mathematically fashionable. Partitions that are not complete or disjoint probably reflect a lack of clarity in the rationale for the partition. In particular, if a partition actually encodes two or three rationales, the partition is likely to be quite messy, and it is also likely to violate either the completeness or the disjointness property (or both!). Identifying and correcting completeness or disjointness errors typically results in esthetically more pleasing partitions. Further, formally objectionable "partitions" cause unnecessary problems when generating tests, as discussed below. The rest of this chapter assumes that the partitions are both complete and disjoint.

4.1 INPUT DOMAIN MODELING

The first step in input domain modeling is identification of testable functions. Consider the `TriTyp` program from Chapter 3. `TriTyp` clearly has only one testable function with three parameters. The situation is more complex for Java class APIs. Each public method is typically a testable function that should be tested individually. However, the characteristics are often the same for several methods, so it helps to develop a common set of characteristics for the entire class and then develop specific tests for each method. Finally, large systems are certainly amenable to the input space partition approach, and such systems often supply complex functionality. Tools like UML use cases can be used to identify testable functions. Each use case is associated with a specific intended functionality of the system, so it is very likely that the use case designers have useful characteristics in mind that are relevant to developing test cases. For example, a "withdrawal" use case for an ATM identifies "withdrawing cash" as a testable function. Further, it suggests useful categories such as "Is Card Valid?" and "Relation of Withdrawal Policy to Withdrawal Request."

The second step is to identify all of the parameters that can affect the behavior of a given testable function. This step isn't particularly creative, but it is important to carry it out completely. In the simple case of testing a stateless method, the parameters are simply the formal parameters to the method. If the method has state, which is common in many object-oriented classes, then the state must be included as a parameter. For example, the `insert(Comparable obj)` method for a binary tree class behaves differently depending on whether or not `obj` is already in the tree. Hence, the current state of the tree needs to be explicitly identified as a parameter to the `insert()` method. In a slightly more complex example, a method `find(String str)` that finds the location of `str` in a file depends, obviously, on the particular file being searched. Hence, the test engineer explicitly identifies the file as a parameter to the `find()` method. Together, all of the parameters form the *input domain* of the function under test.

The third step, and the key creative engineering step, is modeling the input domain articulated in the prior step. An *input domain model* (*IDM*) represents the input space of the system under test in an abstract way. A test engineer describes

the structure of the input domain in terms of input *characteristics*. The test engineer creates a *partition* for each characteristic. The partition is a set of *blocks*, each of which contains a set of *values*. From the perspective of that particular characteristic, the values in each block are considered equivalent.

A test input is a tuple of values, one for each parameter. By definition, the test input belongs to exactly one block from each characteristic. Thus, if we have even a modest number of characteristics, the number of possible combinations may be infeasible. In particular, adding another characteristic with n blocks increases the number of combinations by a factor of n. Hence, controlling the total number of combinations is a key feature of any practical approach to input domain testing. In our view, this is the job of the coverage criteria, which we address in Section 4.2.

Different testers will come up with different models, depending on creativity and experience. These differences create a potential for variance in the quality of the resulting tests. The structured method to support input domain modeling presented in this chapter can decrease this variance and increase the overall quality of the IDM.

Once the IDM is built and values are identified, some combinations of the values may be invalid. The IDM must include information to help the tester identify and avoid or remove invalid sub-combinations. The model needs a way to represent these restrictions. Constraints are discussed further in Section 4.3.

The next section provides two different approaches to input domain modeling. The *interface-based* approach develops characteristics directly from input parameters to the program under test. The *functionality-based* approach develops characteristics from a functional or behavioral view of the program under test. The tester must choose which approach to use. Once the IDM is developed, several coverage criteria are available to decide which combinations of values to use to test the software. These are discussed in Section 4.2.

4.1.1 Interface-Based Input Domain Modeling

The interface-based approach considers each particular parameter in isolation. This approach is almost mechanical to follow, but the resulting tests are surprisingly good.

An obvious strength of using the interface-based approach is that it is easy to identify characteristics. The fact that each characteristic limits itself to a single parameter also makes it easy to translate the abstract tests into executable test cases.

A weakness of this approach is that not all the information available to the test engineer will be reflected in the interface domain model. This means that the IDM may be incomplete and hence additional characteristics are needed.

Another weakness is that some parts of the functionality may depend on combinations of specific values of several interface parameters. In the interface-based approach each parameter is analyzed in isolation with the effect that important subcombinations may be missed.

Consider the `TriTyp` program from Chapter 3. It has three integer parameters that represent the lengths of three sides of a triangle. In an interface-based IDM, `Side1` will have a number of characteristics, as will `Side2` and `Side3`. Since the three variables are all of the same type, the interface-based characteristics for each will

likely be identical. For example, since Side1 is an integer, and zero is often a special value for integers, Relation of Side1 to zero is a reasonable interface-based characteristic.

4.1.2 Functionality-Based Input Domain Modeling

The idea of the functionality-based approach is to identify characteristics that correspond to the intended functionality of the system under test rather than using the actual interface. This allows the tester to incorporate some semantics or domain knowledge into the IDM.

Some members of the community believe that a functionality-based approach yields better test cases than the interface-based approach because the input domain models include more semantic information. Transferring more semantic information from the specification to the IDM makes it more likely to generate expected results for the test cases, an important goal.

Another important strength of the functionality-based approach is that the requirements are available before the software is implemented. This means that input domain modeling and test case generation can start early in development.

In the functionality-based approach, identifying characteristics and values may be far from trivial. If the system is large and complex, or the specifications are informal and incomplete, it can be very hard to design reasonable characteristics. The next section gives practical suggestions for designing characteristics.

The functionality-based approach also makes it harder to generate tests. The characteristics of the IDM often do not map to single parameters of the software interface. Translating the values into executable test cases is harder because constraints of a single IDM characteristic may affect multiple parameters in the interface.

Returning to the TriTyp program from Chapter 3, a functionality-based approach will recognize that instead of simply three integers, the input to the method is a triangle. This leads to the characteristic of a triangle, which can be partitioned into different types of triangles (as discussed below).

4.1.3 Identifying Characteristics

Identifying characteristics in an interface-based approach is simple. There is a mechanical translation from the parameters to characteristics. Developing a functionality-based IDM is more challenging.

Preconditions are excellent sources for functionality-based characteristics. They may be explicit or encoded in the software as exceptional behavior. Preconditions explicitly separate defined (or normal) behavior from undefined (or exceptional) behavior. For example, if a method choose() is supposed to select a value, it needs a precondition that a value must be available to select. A characteristic may be whether the value is available or not.

Postconditions are also good sources for characteristics. In the case of TriTyp, the different kinds of triangles are based on the postcondition of the method.

The test engineer should also look for other relationships between variables. These may be explicit or implicit. For example, a curious test engineer given a

method m() with two object parameters x and y might wonder what happens if x and y point to the same object (aliasing), or to logically equal objects.

Another possible idea is to check for missing factors, that is, factors that may impact the execution but do not have an associated IDM parameter.

It is usually better to have many characteristics with few blocks than the reverse. It is also true that characteristics with small numbers of blocks are more likely to satisfy the disjointness and completeness properties.

Generally, it is preferable for the test engineer to use specifications or other documentation instead of program code to develop characteristics. The idea is that the tester should apply input space partitioning by using *domain knowledge* about the problem, not the implementation. However, in practice, the code may be all that is available. Overall, the more semantic information the test engineer can incorporate into characteristics, the better the resulting test set is likely to be.

The two approaches generally result in different IDM characteristics. The following method illustrates this difference:

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//    else returns true if element is in the list, false otherwise
```

If the interface-based approach is used, the IDM will have characteristics for list and characteristics for element. For example, here are two interface-based characteristics for list, including blocks and values, which are discussed in detail in the next section:

- list is null
  - b_1 = True
  - b_2 = False
- list is empty
  - b_1 = True
  - b_2 = False

The functionality-based approach results in more complex IDM characteristics. As mentioned earlier, the functionality-based approach requires more thinking on the part of the test engineer, but can result in better tests. Two possibilities for the example are listed below, again including blocks and values.

- number of occurrences of element in list
  - $b_1 = 0$
  - $b_2 = 1$
  - b_3 = More than 1
- element occurs first in list
  - b_1 = True
  - b_2 = False

4.1.4 Choosing Blocks and Values

After choosing characteristics, the test engineer partitions the domains of the characteristics into sets of values called *blocks*. A key issue in any partition approach is how partitions should be identified and how representative values should be selected from each block. This is another creative design step that allows the tester to tune the test process. More blocks will result in more tests, requiring more resources but possibly finding more faults. Fewer blocks will result in fewer tests, saving resources but possibly reducing test effectiveness. Several general strategies for identifying values are as follows:

- **Valid values**: Include at least one group of valid values.
- **Sub-partition**: A range of valid values can often be partitioned into sub-partitions, such that each sub-partition exercises a somewhat different part of the functionality.
- **Boundaries**: Values at or close to boundaries often cause problems.
- **Normal use**: If the operational profile focuses heavily on "normal use," the failure rate depends on values that are not boundary conditions.
- **Invalid values**: Include at least one group of invalid values.
- **Balance**: From a cost perspective, it may be cheap or even free to add more blocks to characteristics that have fewer blocks. In Section 4.2, we will see that the number of tests sometimes depends on the characteristic with the maximum number of blocks.
- **Missing partitions**: Check that the union of all blocks of a characteristic completely covers the input space of that characteristic.
- **Overlapping partitions**: Check that no value belongs to more than one block.

Special values can often be used. Consider a Java reference variable; null is typically a special case that needs to be treated differently from non null values. If the reference is to a container structure such as a Set or List, then whether the container is empty or not is often a useful characteristic.

Consider the TriTyp program from Chapter 3. It has three integer parameters that represent the lengths of three sides of a triangle. One common partitioning for an integer variable considers the relation of the variable's value to some special value in the testable function's domain, such as zero.

Table 4.1 shows a partitioning for the interface-based IDM for the TriTyp program. It has three characteristics, q_1, q_2, and q_3.

The first row in the table should be read as "Block $q_1.b_1$ is that Side 1 is greater than zero," "Block $q_1.b_2$ is that Side 1 is equal to zero," and "Block $q_1.b_3$ is that Side 1 is less than zero."

Table 4.1. First partitioning of TriTyp's inputs (interface-based)

Partition	b_1	b_2	b_3
q_1 = "Relation of Side 1 to 0"	*greater than 0*	*equal to 0*	*less than 0*
q_2 = "Relation of Side 2 to 0"	*greater than 0*	*equal to 0*	*less than 0*
q_3 = "Relation of Side 3 to 0"	*greater than 0*	*equal to 0*	*less than 0*

Table 4.2. Second partitioning of TriTyp's inputs (interface-based).

Partition	b_1	b_2	b_3	b_4
q_1 = "Length of Side 1"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*
q_2 = "Length of Side 2"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*
q_3 = "Length of Side 3"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*

Consider the partition q_1 for Side 1. If one value is chosen from each block, the result is three tests. For example, we might choose Side 1 to have the value 7 in test 1, 0 in test 2, and −3 in test 3. Of course, we also need values for Side 2 and Side 3 of the triangle to complete the test case values. Notice that some of the blocks represent valid triangles and some represent invalid triangles. For example, no valid triangle can have a side of negative length.

It is easy to refine this categorization to get more fine grained testing if the budget allows. For example, more blocks can be created by separating inputs with value 1. This decision leads to a partitioning with four blocks, as shown in Table 4.2.

Notice that if the value for Side 1 were floating point rather than integer, the second categorization would **not** yield valid partitions. None of the blocks would include values between 0 and 1 (noninclusive), so the blocks would not cover the domain (not be complete). However, the domain D contains integers so the partitions are valid.

While partitioning, it is often useful for the tester to identify candidate values for each block to be used in testing. The reason to identify values now is that choosing specific values can help the test engineer think more concretely about the predicates that describe each block. While these values may not prove sufficient when refining test requirements to test cases, they do form a good starting point. Table 4.3 shows values that can satisfy the second partitioning.

The above partitioning is interface based and only uses syntactic information about the program (it has three integer inputs). A functionality-based approach can use the semantic information of the traditional geometric classification of triangles, as shown in Table 4.4.

Of course, the tester has to know what makes a triangle scalene, equilateral, isosceles, and invalid to choose possible values (this may be simple middle school geometry, but many of us have probably forgotten). An equilateral triangle is one in which all sides are the same length. An isosceles triangle is one in which at least two sides are the same length. A scalene triangle is any other valid triangle. This brings up a subtle problem, Table 4.4 does **not** form a valid partitioning. An equilateral triangle is also isosceles, thus we must first correct the partitions, as shown in Table 4.5.

Table 4.3. Possible values for blocks in the second partitioning in Table 4.2

Param	b_1	b_2	b_3	b_4
Side 1	2	1	0	−1
Side 2	2	1	0	−1
Side 3	2	1	0	−1

Table 4.4. Geometric partitioning of TriTyp's inputs (functionality-based)

Partition	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	*scalene*	*isosceles*	*equilateral*	*invalid*

Now values for Table 4.5 can be chosen as shown in Table 4.6. The triplets represent the three sides of the triangle.

A different approach to the equilateral/isosceles problem above is to break the characteristic Geometric Partitioning into four separate characteristics, namely Scalene, Isosceles, Equilateral, and Valid. The partition for each of these characteristics is boolean, and the fact that choosing Equilateral = true also means choosing Isosceles = true is then simply a constraint. Such an approach is highly recommended, and it invariably satisfies the disjointness and completeness properties.

4.1.5 Using More than One Input Domain Model

For a complex program it might be better to have several small IDMs than one large. This approach allows for a divide-and-conquer strategy when modeling characteristics and blocks. Another advantage with multiple IDMs for the same software is that it allows varying levels of coverage.

For instance, one IDM may contain only valid values and another IDM may contain invalid values to focus on error handling. The valid value IDM may be covered using a higher level of coverage. The invalid value IDM may use a lower level of coverage.

Multiple IDMs may be overlapping as long as the test cases generated make sense. However, overlapping IDMs are likely to have more constraints.

4.1.6 Checking the Input Domain Model

It is important to check the input domain model. In terms of characteristics, the test engineer should ask whether there is any information about how the function behaves that is not incorporated in some characteristics. This is necessarily an informal process.

The tester should also explicitly check each characteristic for the completeness and disjointness properties. The purpose of this check is to make sure that, for each characteristic, not only do the blocks cover the complete input space, but selecting a particular block implies excluding all other blocks in that characteristic.

If multiple IDMs are used, completeness should be relative to the portion of the input domain that is modeled in each IDM. When the tester is satisfied with the

Table 4.5. Correct geometric partitioning of TriTyp's inputs (functionality-based)

Partition	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	*scalene*	*isosceles, not equilateral*	*equilateral*	*invalid*

Table 4.6. Possible values for blocks in geometric partitioning in Table 4.5.

Param	b_1	b_2	b_3	b_4
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

characteristics and their blocks, it is time to choose which combinations of values to test with and identify constraints among the blocks.

EXERCISES

Section 4.1.

1. Answer the following questions for the method search() below:

```
public static int search (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//    else if element is in the list, return an index
//    of element in the list; else return -1
//    for example, search ([3,3,1], 3) = either 0 or 1
//        search ([1,7,5], 2) = -1
```

Base your answer on the following characteristic partitioning:

```
Characteristic: Location of element in list
    Block 1: element is first entry in list
    Block 2: element is last entry in list
    Block 3: element is in some position other than first or last
```

(a) "Location of element in list" fails the disjointness property. Give an example that illustrates this.
(b) "Location of element in list" fails the completeness property. Give an example that illustrates this.
(c) Supply one or more new partitions that capture the intent of "Location of e in list" but do not suffer from completeness or disjointness problems.

2. Derive input space partitioning tests for the **GenericStack** class with the following method signatures:
   - public GenericStack ();
   - public void Push (Object X);
   - public Object Pop ();
   - public boolean IsEmt ();

   Assume the usual semantics for the stack. Try to keep your partitioning simple, choose a small number of partitions and blocks.
   (a) Define characteristics of inputs
   (b) Partition the characteristics into blocks
   (c) Define values for the blocks

4.2 COMBINATION STRATEGIES CRITERIA

The above description ignores an important question: "How should we consider multiple partitions at the same time?" This is the same as asking "What combination of blocks should we choose values from?" For example, we might wish to require a test case that satisfies block 1 from q_2 and block 3 from q_3. The most obvious choice is to choose all combinations. However, just like Combinatorial Coverage from previous chapters, using all combinations will be impractical when more than 2 or 3 partitions are defined.

> CRITERION **4.23 All Combinations Coverage (ACoC):** *All combinations of blocks from all characteristics must be used.*

For example, if we have three partitions with blocks [A, B], [1, 2, 3], and [x, y], then ACoC will need the following twelve tests:

(A, 1, x)	(B, 1, x)
(A, 1, y)	(B, 1, y)
(A, 2, x)	(B, 2, x)
(A, 2, y)	(B, 2, y)
(A, 3, x)	(B, 3, x)
(A, 3, y)	(B, 3, y)

A test suite that satisfies ACoC will have a unique test for each combination of blocks for each partition. The number of tests will be the product of the number of blocks for each partition: $\prod_{i=1}^{Q}(B_i)$.

If we use a four block partition similar to q_2 for each of the three sides of the triangle, ACoC requires $4 * 4 * 4 = 64$ tests.

This is almost certainly more testing than is necessary, and will usually be economically impractical as well. Thus, as with paths and truth tables before, we must use some sort of coverage criterion to choose which combinations of blocks to pick values from.

The first, fundamental assumption is that different choices of values from the same block are equivalent from a testing perspective. That is, we need to take only one value from each block. Several *combination strategies* exist, which result in a collection of useful criteria. These combination strategies are illustrated with the TriTyp example, using the second categorization given in Table 4.2 and the values from Table 4.3.

The first combination strategy criterion is fairly straightforward and simply requires that we try each choice at least once.

> CRITERION **4.24 Each Choice Coverage (ECC):** *One value from each block for each characteristic must be used in at least one test case.*

Given the above example of three partitions with blocks [A, B], [1, 2, 3], and [x, y], ECC can be satisfied in many ways, including the three tests (A, 1, x), (B, 2, y), and (A, 3, x).

Assume the program under test has Q parameters $q_1, q_2, \ldots, q_Q$, and each parameter q_i has B_i blocks. Then a test suite that satisfies ECC will have at least $Max_{i=1}^{Q} B_i$

values. The maximum number of blocks for the partitions for TriTyp is four, thus ECC requires at least four tests.

This criterion can be satisfied by choosing the tests {(2, 2, 2), (1, 1, 1), (0, 0, 0), (−1, −1, −1)} from Table 4.3. It does not take much thought to conclude that these are not very effective tests for this program. ECC leaves a lot of flexibility to the tester in terms of how to combine the test values, so it can be called a relatively "weak" criterion.

The weakness of ECC can be expressed as not requiring values to be combined with other values. A natural next step is to require explicit combinations of values, called *pair-wise*.

> CRITERION **4.25 Pair-Wise Coverage (PWC):** *A value from each block for each characteristic must be combined with a value from every block for each other characteristic.*

Given the above example of three partitions with blocks [A, B], [1, 2, 3], and [x, y], then PWC will need sixteen tests to cover the following combinations:

(A, 1)	(B, 1)	(1, x)
(A, 2)	(B, 2)	(1, y)
(A, 3)	(B, 3)	(2, x)
(A, x)	(B, x)	(2, y)
(A, y)	(B, y)	(3, x)
		(3, y)

PWC allows the same test case to cover more than one unique pair of values. So the above combinations can be combined in several ways, including:

(A, 1, x)	(B, 1, y)
(A, 2, x)	(B, 2, y)
(A, 3, x)	(B, 3, y)
(A, –, y)	(B, –, x)

The tests with '–' mean that any block can be used.

A test suite that satisfies PWC will pair each value with each other value or have at least $(Max_{i=1}^{Q} B_i)^2$ values. Each characteristic in TriTyp (Table 4.3) has four blocks; so at least 16 tests are required.

Several algorithms to satisfy PWC have been published and appropriate references are provided in the bibliography section of the chapter.

A natural extension to PWC is to require t values instead of pairs.

> CRITERION **4.26 T-Wise Coverage (TWC):** *A value from each block for each group of t characteristics must be combined.*

If the value for T is chosen to be the number of partitions, Q, then TWC is equivalent to all combinations. A test suite that satisfies TWC will have at least $(Max_{i=1}^{q} B_i)^t$ values. TWC is expensive in terms of the number of test cases, and experience suggests going beyond pair-wise (that is, $t = 2$) does not help much.

Both PWC and TWC combine values "blindly," without regard for which values are being combined. The next criterion strengthens ECC in a different way by bringing in a small but crucial piece of domain knowledge of the program; asking what is the most "important" block for each partition. This block is called the *base choice*.

> CRITERION **4.27 Base Choice Coverage (BCC):** *A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.*

Given the above example of three partitions with blocks [A, B], [1, 2, 3], and [x, y], suppose base choice blocks are 'A', '1' and 'x'. Then the base choice test is (A, 1, x), and the following additional tests would need to be used:

(B, 1, x)
(A, 2, x)
(A, 3, x)
(A, 1, y)

A test suite that satisfies BCC will have one base test, plus one test for each remaining block for each partition. This is a total of $1 + \sum_{i=1}^{Q}(B_i - 1)$. Each parameter for `TriTyp` has four blocks, thus BCC requires $1 + 3 + 3 + 3$ tests.

The base choice can be the simplest, the smallest, the first in some ordering, or the most likely from an end-user point of view. Combining more than one invalid value is usually not useful because the software often recognizes one value and negative effects of the others are masked. Which blocks are chosen for the base choices becomes a crucial step in test design that can greatly impact the resulting test. It is important that the tester document the strategy that was used so that further testing can reevaluate those decisions.

Following the strategy of choosing the most likely block for `TriTyp`, we chose "greater than 1" from Table 4.2 as the base choice block. Using the values from Table 4.3 gives the base test as (2, 2, 2). The remaining tests are created by varying each one of these in turn: {(2, 2, 1), (2, 2, 0), (2, 2, −1), (2, 1, 2), (2, 0, 2), (2, −1, 2), (1, 2, 2), (0, 2, 2), (−1, 2, 2) }.

Sometimes the tester may have trouble choosing a single base choice and may decide that multiple base choices are needed. This is formulated as follows:

> CRITERION **4.28 Multiple Base Choices (MBCC):** *At least one, and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic.*

Assuming m_i base choices for each characteristic and a total of M base tests, MBCC requires $M + \sum_{i=1}^{Q}(M * (B_i - m_i))$ tests.

For example, we may choose to include two base choices for side 1 in `TriTyp`, "greater than 1" and "equal to 1." This would result in the two base tests (2, 2, 2) and (1, 2, 2). The formula above is thus evaluated with $M = 2$, $m_1 = 2$, and

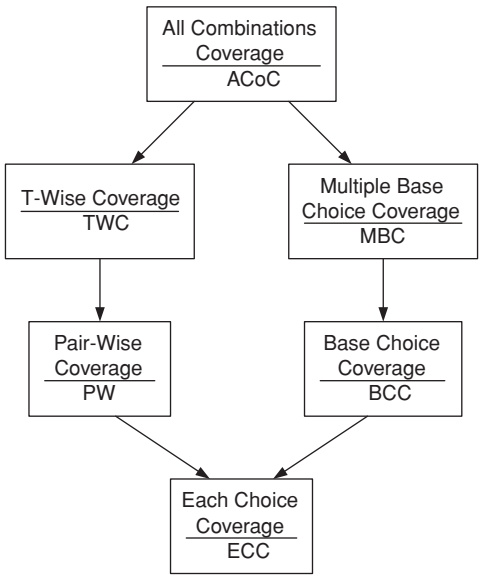

Figure 4.2. Subsumption relations among input space partitioning criteria.

$m_i = 1 \; \forall \; i, 1 < i \leq 3$. That is, 2 + (2*(4 − 2)) + (2*(4 − 1)) + (2*(4 − 1)) = 18. The remaining tests are created by varying each one of these in turn. The MBCC criterion sometimes results in duplicate tests. For example, $(0, 2, 2)$ and $(-1, 2, 2)$ both appear twice for TriTyp. Duplicate test cases should, of course, be eliminated (which also makes the formula for the number of tests an upper bound).

Figure 4.2 shows the subsumption relationships among the input space partitioning combination strategy criteria.

EXERCISES

Section 4.2.

1. Enumerate all 64 tests to satisfy the All Combinations (ACoC) criterion for the second categorization of TriTyp's inputs in Table 4.2. Use the values in Table 4.3.
2. Enumerate all 16 tests to satisfy the pair-wise (PWC) criterion for the second categorization of TriTyp's inputs in Table 4.2. Use the values in Table 4.3.
3. Enumerate all 16 tests to satisfy the multiple base choice (MBCC) criterion for the second categorization of TriTyp's inputs in Table 4.2. Use the values in Table 4.3.
4. Answer the following questions for the method intersection() below:

```
public Set intersection (Set s1, Set s2)
 // Effects:  If s1 or s2 are null throw NullPointerException
 //    else return a (non null) Set equal to the intersection
 //    of Sets s1 and s2
 //    A null argument is treated as an empty set.
```

```
Characteristic: Type of s1
  - s1 = null
  - s1 = {}
  - s1 has at least one element

Characteristic: Relation between s1 and s2
  - s1 and s2 represent the same set
  - s1 is a subset of s2
  - s2 is a subset of s1
  - s1 and s2 do not have any elements in common
```

(a) Does the partition "Type of s1" satisfy the completeness property? If not, give a value for $s1$ that does not fit in any block.
(b) Does the partition "Type of s1" satisfy the disjointness property? If not, give a value for $s1$ that fits in more than one block.
(c) Does the partition "Relation between s1 and s2" satisfy the completeness property? If not, give a pair of values for $s1$ and $s2$ that does not fit in any block.
(d) Does the partition "Relation between s1 and s2" satisfy the disjointness property? If not, give a pair of values for $s1$ and $s2$ that fits in more than one block.
(e) If the "base choice" criterion were applied to the two partitions (exactly as written), how many test requirements would result?

5. Derive input space partitioning tests for the **BoundedQueue** class with the following signature:

   - `public BoundedQueue (int capacity);`
   - `public void Enqueue (Object X);`
   - `public Object Dequeue ();`
   - `public boolean IsEmpty ();`
   - `public boolean IsFull ();`

   Assume the usual semantics for a queue with a fixed, maximal capacity. Try to keep your partitioning simple–choose a small number of partitions and blocks.
   (a) Identify all of the variables. Don't forget the state variables.
   (b) Identify several characteristics that suggest partitions.
   (c) Identify the blocks in the partition for each characteristic. Designate one block in each partition as the "Base" block.
   (d) Define values for the blocks.
   (e) Define a test set that satisfies base choice coverage (BCC).
6. Develop a set of characteristics and accompanying partitions for the pattern checking procedure (the method *pat()* in Figure 2.21 in Chapter 2).
   (a) Develop tests to satisfy the base choice criterion. Your tests should have both inputs and expected outputs.
   (b) Analyze your tests with respect to the data flow test sets developed in Chapter 2. How well does input space partitioning do?

Table 4.7. Examples of invalid block combinations

	Blocks			
Characteristics	**1**	**2**	**3**	**4**
A: length and contents	one element	more than one, unsorted	more than one, sorted	more than one, all identical
B: match	element not found	element found once	element found more than once	–

Invalid combinations: (A1, B3), (A4, B2)

4.3 CONSTRAINTS AMONG PARTITIONS

A subtle point about input space partitioning is that some combinations of blocks are infeasible. This must be documented in the IDM. For example, Table 4.7 shows an example based on the previously described `boolean findElement (list, element)` method. An IDM with two parameters *A*, that has four partitions, and *B*, that has three partitions, has been designed. Two of the partition combinations do not make sense and are thus invalid. In this example, these are represented as a list of invalid pairs of parameter partitions. In the general case other representations can be used, for example, a set of inequalities.

Constraints are relations between blocks from different characteristics. Two kinds of constraints appear. One kind says that a block from one characteristic cannot be combined with a block from another characteristic. The "less than zero" and "scalene" problem is an example of this kind of constraint. The other kind is the inverse; a block from one characteristic **must be** combined with a specific block from another characteristic. Although this sounds simple enough, identifying and satisfying the constraints when choosing values can be difficult.

How constraints are handled when values are selected depends on the coverage criterion chosen, and the decision is usually made when values are chosen. For the ACoC, PWC, and TWC criteria, the only reasonable option is to drop the infeasible pairs from consideration. For example, if PWC requires a particular pair that is not feasible, no amount of tinkering on the test engineer's part can make that requirement feasible. However, the situation is quite different for a criterion such as BCC. If a particular variation (for example, "less than zero" for "Relation of Side 1 to zero") conflicts with the base case (for example, "scalene" for "Geometric Classification"), then the obvious thing to do is change the offending choice for the base case so that the variation *is* feasible. In this case, "Geometric Classification" clearly needs to change to "invalid."

As another example, consider sorting an array. The input to our sort routine will be a variable length array of some arbitrary type. The output will have three parts: (1) a permutation of the input array, sorted in ascending order, (2) the largest value (max), and (3) the smallest value (min). We might consider the following characteristics:

- Length of array
- Type of elements

- Max value
- Min value
- Position of max value
- Position of min value

These characteristics can, in turn, reasonably result in the partitioning summarized as follows:

Length	{0, 1, 2..100, 101..MAXINT}
Type	{int, char, string, other}
Max	{≤ 0, 1, > 1, 'a', 'Z', 'b', ..., 'Y', blank, nonblank}
Min	{···}
Max pos	{1, 2..Length-1, Length}
Min pos	{1, 2..Length-1, Length}

The discerning reader will of course notice that not all combinations are possible. For example, if *Length* = 0, then nothing else matters. Also, some of the *Max* and *Min* values are available only if *Type* = *int*, and others if *Type* = *char*.

4.4 BIBLIOGRAPHIC NOTES

In the research literature, several testing methods have been described that are generally based on the model that the input space of the test object should be divided into subsets, with the assumption that all inputs in the same subset cause similar behavior. These are collectively called *partition testing* and include equivalence partitioning [249], boundary value analysis [249], category partition [283], and domain testing [29]. An extensive survey with examples was published by Grindal et al. [143].

The derivation of partitions and values started with Balcer, Hasling, and Ostrand's category partition method in 1988 [23, 283]. An alternate visualization is that of classification trees introduced by Grochtman, Grimm, and Wegener in 1993 [145, 146]. Classification trees organize the input space partitioning information into a tree structure. The first level nodes are the parameters and environment variables (characteristics); they may be recursively broken into sub-categories. Blocks appear as leaves in the tree and combinations are chosen by selecting among the leaves.

Chen et al. empirically identified common mistakes that testers made during input parameter-modeling [64]. Many of the concepts on input domain modeling in this chapter come from Grindal's PhD work [140, 142, 144]. Both Cohen et al. [84] and Yin et al. [363] suggest functionality oriented approaches to input parameter modeling. Functionality-oriented input parameter modeling was also implicitly used by Grindal et al. [141]. Two other IDM-related methods are Classification Trees [145] and a UML activity diagram based method [65]. Beizer [29], Malaiya [222], and Chen et al. [64] also address the problem of characteristic selection.

Grindal published an analytical/empirical comparison of different constraint-handling mechanisms [144].

Stocks and Carrington [320] provided a formal notion of specification-based testing that encompasses most approaches to input space partition testing. In particular,

they addressed the problem of refining test frames (which we simply and informally call test requirements in this book) to test cases.

The each choice and base choice criteria were introduced by Ammann and Offutt in 1994 [16]. Cohen et al. [84] indicated that valid and invalid parameter values should be treated differently with respect to coverage. *Valid* values lie within the bounds of normal operation of the test object, and *invalid* values lie outside the normal operating range. Invalid values often result in an error message and the execution terminates. To avoid one invalid value masking another, Cohen et al. suggested that only one invalid value should be included in each test case.

Burroughs et al. [58] and Cohen et al. [84, 85, 86] suggested the heuristic pair-wise coverage as part of the Automatic Efficient Test Generator (AETG). AETG also includes a variation on the base choice combination criterion. In AETG's version, called "default testing," the tester varies the values of one characteristic at a time while the other characteristics contain *some* default value. The term "default testing" was also used by Burr and Young [57], who described yet another variation of the base choices. In their version, all characteristics except one contain the default value, and the remaining characteristics contain a maximum or a minimum value. This variant will not necessarily satisfy "each choice coverage."

The Constrained Array Test System (CATS) tool for generating test cases was described by Sherwood [313] to satisfy pair-wise coverage. For programs with two or more characteristics, the in-parameter-order (IPO) combination strategy [205, 206, 322] generates a test suite that satisfies pair-wise coverage for the first two parameters (characteristic in our terminology). The test suite is then extended to satisfy pair-wise coverage for the first three parameters and continues for each additional parameter until all parameters are included.

Williams and Probert invented T-wise coverage [354]. A special case of T-wise coverage called *variable strength* was proposed by Cohen, Gibbons, Mugridge, and Colburn [87]. This strategy requires higher coverage among a subset of characteristics and lower coverage across the others. Assume for example a test problem with four parameters A, B, C, D. Variable strength may require 3-wise coverage for parameters B, C, D and 2-wise coverage for parameter A. Cohen, Gibbons, Mugridge, and Colburn [87] suggested using simulated annealing (SA) to generate test suites for T-wise coverage. Shiba, Tsuchiya, and Kikuno [314] proposed using a genetic algorithm (GA) to satisfy pair-wise coverage. The same paper also suggested using the ant colony algorithm (ACA).

Mandl suggested using orthogonal arrays to generate values for T-wise coverage [224]. This idea was further developed by Williams and Probert [353]. Covering arrays [352] is an extension of orthogonal arrays. A property of orthogonal arrays is that they are *balanced*, which means that each characteristic value occurs the same number of times in the test suite. If only T-wise (for instance pair-wise) coverage is desired, the balance property is unnecessary and will make the algorithm less efficient. In a covering array that satisfies T-wise coverage, each T-tuple occurs at least once but not necessarily the same number of times. Another problem with orthogonal arrays is that for some problem sizes we do not have enough orthogonal arrays to represent the entire problem. This problem is also avoided by using covering arrays.

Several papers have provided experiential and experimental results of using input space partitioning. Heller [156] uses a realistic example to show that testing all

combinations of characteristic values is infeasible in practice. Heller concludes that we need to identify a subset of combinations of manageable size.

Kuhn and Reilly [195] investigated 365 error reports from two large real-life projects and discovered that pair-wise coverage was nearly as effective at finding faults as testing all combinations. More supporting data were given by Kuhn and Wallace [196].

Piwowarski, Ohba, and Caruso [291] describe how to apply code coverage successfully as a stopping criterion during functional testing. The authors formulated functional testing as the problem of selecting test cases from all combinations of values of the input parameters. Burr and Young [57] show that continually monitoring code coverage helps improve the input domain model. Initial experiments showed that ad hoc testing resulted in about 50% decision coverage, but by continually applying code coverage and refining the input domain models, decision coverage was increased to 84%.

Plenty of examples of applying input space partitioning in practice have been published. Dalal, Jain, Karunanithi, Leaton, Lott, Patton, and Horowitz [91, 92] report results from using the AETG tool. It was used to generate test cases for Bellcore's Intelligent Service Control Point, a rule-based system used to assign work requests to technicians, and a GUI window in a large application. Previously, Cohen, Dalal, Kajla, and Patton [85] demonstrated the use of AETG for screen testing, by testing the input fields for consistency and validity across a number of screens.

Burr and Young [57] also used the AETG tool to test a Nortel application that converts email messages from one format to another. Huller [171] used an IPO related algorithm to test ground systems for satellite communications.

Williams and Probert [353] demonstrated how input space partitioning can be used to organize configuration testing. Yilmaz, Cohen, and Porter [362] used covering arrays as a starting point for fault localization in complex configuration spaces.

Huller [171] showed that pair-wise configuration testing can save more than 60% in both cost and time compared to quasi-exhaustive testing. Brownlie, Prowse, and Phadke [50] compared the results of using orthogonal arrays (OA) on one version of a PMX/StarMAIL release with the results from conventional testing on a prior release. The authors estimated that 22% more faults would have been found if OA had been used on the first version.

Several studies have compared the number of tests generated. The number of tests varies when using nondeterministic algorithms. Several papers compared input space partitioning strategies that satisfy 2-wise or 3-wise coverage: IPO and AETG [205], OA and AETG [141], covering arrays (CA) and IPO [352], and AETG, IPO, SA, GA, and ACA [87, 314]. Most of them found very little difference.

Another way to compare algorithms is with respect to the execution time. Lei and Tai [206] showed that the time complexity of IPO is superior to that of AETG. Williams [352] reported that CA outperforms IPO by almost three orders of magnitude for the largest test problems in his study.

Grindal et al. [141] compared algorithms by the number of faults found. They found that BCC performs as well as AETG and OA despite fewer test cases.

Input space partitioning strategies can also be compared based on their code coverage. Cohen et al. [86] found that test suites generated by AETG for 2-wise

coverage reach over 90% block coverage. Burr and Young [57] got similar results for AETG, getting 93% block coverage with 47 test cases, compared with 85% block coverage for a restricted version of BCC using 72 test cases.

NOTES

1 We choose to use blocks for simplicity. These are also sometimes called "partitions" in the literature.