Chapter 11

Combinatorial Testing

Requirements specifications typically begin in the form of natural language statements. The flexibility and expressiveness of natural language, which are so important for human communication, represent an obstacle to automatic analysis. Combinatorial approaches to functional testing consist of a manual step of structuring the specification statement into a set of properties or attributes that can be systematically varied, and an automatizable step of producing combinations of choices.

Simple "brute force" synthesis of test cases by test designers squanders the intelligence of expert staff on tasks that can be partly automated. Even the most expert of test designers will perform sub-optimally and unevenly when required to perform the repetitive and tedious aspects of test design, and quality will vary widely and be difficult to monitor and control. Additionally, estimation of the effort and number of test cases required for a given functionality will be subjective.

Combinatorial approaches decompose the "brute force" work of the test designers into steps, to attack the problem incrementally by separating analysis and synthesis activities that can be quantified and monitored, and partially supported by tools. They identify the variability of elements involved in the execution of a given functionality, and select representative combinations of relevant values for test cases. Repetitive activities such as the combination of different values can be easily automated, thus allowing test designers to focus on more creative and difficult activities.

Required Background

- Chapter 10
  Understanding the limits of random testing and the needs of a systematic approach motivates the study of combinatorial as well as model-based testing techniques. The general functional testing process illustrated in Section 10.3 helps position combinatorial techniques within the functional testing process.

11.1 Overview

In this chapter, we introduce three main techniques that are successfully used in industrial environments and represent modern approaches to systematically derive test cases from natural language specifications: The category-partition approach to identifying attributes, relevant values, and possible combinations; combinatorial sampling to test a large number potential interactions of attributes with a relatively small number of inputs; and provision of catalogs to systematize the manual aspects of combinatorial testing.

The category partition approach separates identification of the values that characterize the input space from the combination of different values into complete test cases. It provides a means to estimate the number of test cases early, size a subset of cases that meet cost constraints, and monitor testing progress.

Pairwise and n-way combination testing provide systematic ways to cover interactions among particular attributes of the program input space with a relatively small number of test cases. Like the category partition method, it separates identification of characteristic values from generation of combinations, but it provides greater control over the number of combinations generated.

The manual step of identifying attributes and representative sets of values can be made more systematic using catalogs that aggregate and synthesize the experience of test designers in a particular organization or application domain. Some repetitive steps can be automated, and the catalogs facilitate training for the inherently manual parts.

These techniques address different aspects and problems in designing a suite of test cases from a functional specification, and while one or another may be most suitable for a specification with given characteristics, it is also possible to combine ideas from each.

11.2 Category-Partition Testing

Category-partition testing is a method for generating functional tests from informal specifications. The following steps comprise the core part of the category-partition method:

A. Decompose the specification into independently testable features: Test designers identify features to be tested separately, and identify parameters and any other elements of the execution environment the unit depends on. Environment dependencies are treated identically to explicit parameters. For each parameter and environment element, test designers identify the elementary *parameter characteristics*, which in the category-partition method are usually called *categories.*

Δ parameter characteristic

Δ category

B. Identify Representative Values: Test designers select a set of representative classes of values for each parameter characteristic. Values are selected in isolation, independent of other parameter characteristics. In the category-partition method, classes of values are called *choices,* and this activity is called *partitioning the categories into choices*.

Δ classes of values

Δ choice

C. Generate Test Case Specifications: Test designers impose semantic constraints on values to indicate invalid combinations and restrict valid combinations, e.g., limiting combinations involving exceptional and invalid values.

Categories, choices, and constraints can be provided to a tool to automatically generate a set of test case specifications. Automating trivial and repetitive activities such as these makes better use of human resources and reduces errors due to distraction. Just as important, it is possible to determine the number of test cases that will be generated (by calculation, or by actually generating them) before investing human effort in test execution. If the number of derivable test cases exceeds the budget for test execution and evaluation, test designers can reduce the number of test cases by imposing additional semantic constraints. Controlling the number of test cases *before* test execution begins is preferable to ad hoc approaches in which one may at first create very thorough test suites and then test less and less thoroughly as deadlines approach.

We illustrate the category-partition method using a specification of a feature from the direct sales web site of Chipmunk Computers. Customers are allowed to select and price custom configurations of Chipmunk computers. A *configuration* is a set of selected options for a particular model of computer. Some combinations of model and options are not valid (e.g., digital LCD monitor with analog video card), so configurations are tested for validity before they are priced. The *check configuration* function (Figure 11.1) is given a model number and a set of components, and returns the boolean value **True** if the configuration is valid or **False** otherwise. This function has been selected by the test designers as an independently testable feature.

A. Identify Independently Testable Features and Parameter Characteristics We assume that step *A* starts by selecting the *Check configuration* feature to be tested independently of other features. This entails choosing to separate testing of the configuration check per se from its presentation through a user interface (e.g., a web form), and depends on the architectural design of the software system.

Step *A* requires the test designer to identify the parameter characteristics, i.e., the elementary characteristics of the parameters and environment elements that affect the unit's execution. A single parameter may have multiple elementary characteristics. A quick scan of the functional specification would indicate *model* and *components* as the parameters of *check configuration*. More careful consideration reveals that what is "valid" must be determined by reference to additional information, and in fact the functional specification assumes the existence of a data base of models and components. The data base is an environment element that, although not explicitly mentioned in the functional specification, is required for executing and thus testing the feature, and partly determines its behavior. Note that our goal is not to test a particular configuration of the system with a fixed database, but to test the generic system which may be configured through different database contents.

Having identified *model*, *components*, and *product database* as the parameters and environment elements required to test the *check configuration* functionality, the test designer would next identify the parameter characteristics of each.

Model may be represented as an integer, but we know that it is not to be used arithmetically, but rather serves as a key to the database and other tables. The specification

Check Configuration: Check the validity of a computer configuration. The parameters of *check configuration* are:

Model: A model identifies a specific product and determines a set of constraints on available components. Models are characterized by logical slots for components, which may or may not be implemented by physical slots on a bus. Slots may be required or optional. Required slots must be assigned a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customer's needs.

Example: The required "slots" of the Chipmunk C20 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. (Of these, only the hard disk and memory are implemented using actual hardware slots on a bus.) The optional slots include external storage devices such as a CD/DVD writer.

Set of Components: A set of $\langle slot, component \rangle$ pairs, which must correspond to the required and optional slots associated with the model. A component is a choice that can be varied within a model, and which is not designed to be replaced by the end user. Available components and a default for each slot is determined by the model. The special value "empty" is allowed (and may be the default selection) for optional slots.

In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.

Example: The default configuration of the Chipmunk C20 includes 20 gigabytes of hard disk; 30 and 40 gigabyte disks are also available. (Since the hard disk is a required slot, "empty" is not an allowed choice.) The default operating system is RodentOS 3.2, personal edition, but RodentOS 3.2 mobile server edition may also be selected. The mobile server edition requires at least 30 gigabytes of of hard disk.

Figure 11.1: Functional specification of the feature *check configuration* of the web site of a computer manufacturer.

mentions that a model is characterized by a set of slots for required components and a set of slots for optional components. We may identify *model number*, *number of required slots*, and *number of optional slots* as characteristics of parameter *model*.

Parameter *components* is a collection of $\langle slot, selection \rangle$ pairs. The size of a collection is always an important characteristic, and since components are further categorized as required or optional, the test designer may identify *number of required components with non-empty selection* and *number of optional components with non-empty selection* as characteristics. The matching between the tuple passed to *check configuration* and the one actually required by the selected model is important and may be identified as category *correspondence of selection with model slots*. The actual selections are also significant, but for now the test designer simply identifies *required component selection* and *optional component selection*, postponing selection of relevant values to the next stage in test design.

The environment element *product database* is also a collection, so *number of models in the database* and *number of components in the database* are parameter characteristics. Actual values of database entries are deferred to the next step in test design.

There are no hard-and-fast rules for choosing categories, and it is not a trivial task. Categories reflect the test designer's judgment regarding which classes of values may be treated differently by an implementation, in addition to classes of values that are explicitly identified in the specification. Test designers must also use their experience and knowledge of the application domain and product architecture to look under the surface of the specification and identify hidden characteristics. For example, the specification fragment in Figure 11.1 makes no distinction between configurations of models with several required slots and models with none, but the experienced test designer has seen enough failures on "degenerate" inputs to test empty collections wherever a collection is allowed.

The number of options that can (or must) be configured for a particular model of computer may vary from model to model. However, the category-partition method makes no direct provision for structured data, such as sets of $\langle slot, selection \rangle$ pairs. A typical approach is to "flatten" collections and describe characteristics of the whole collection as parameter characteristics. Typically the size of the collection (the length of a string, for example, or in this case the number of required or optional slots) is one characteristic, and descriptions of possible combinations of elements (occurrence of special characters in a string, for example, or in this case the selection of required and optional components) are separate parameter characteristics.

Suppose the only significant variation among $\langle slot, selection \rangle$ pairs was between pairs that are compatible and pairs that are incompatible. If we treated each $\langle slot, selection \rangle$ pair as a separate characteristic, and assumed n slots, the category-partition method would generate all 2^n combinations of compatible and incompatible slots. Thus we might have a test case in which the first selected option is compatible, the second is compatible, and the third incompatible, and a different test case in which the first is compatible but the second and third are incompatible, and so on, and each of these combinations could be combined in several ways with other parameter characteristics. The number of combinations quickly explodes, and moreover since the number of slots is not actually fixed, we cannot even place an upper bound on the number of combinations that must be considered. We will therefore choose the flattening approach and

Identifying and Bounding Variation

It may seem that drawing a boundary between a fixed program and a variable set of parameters would be the simplest of tasks for the test designer. It is not always so.

Consider a program that produces HTML output. Perhaps the HTML is based on a template, which might be encoded in constants in C or Java code, or might be provided through an external data file, or perhaps both: it could be encoded in a C or source code file that is generated at compile time from a data file. If the HTML template is identified in one case as a parameter to varied in testing, it seems it should be so identified in all three of these variations, or even if the HTML template is embedded directly in print statements of the program, or in an XSLT transformation script.

The underlying principle for identifying parameters to be varied in testing is anticipation of variation in use. Anticipating variation is likewise a key part of architectural and detailed design of software. In a well-designed software system, module boundaries reflect "design secrets," permitting one part of a system to be modified (and retested) with minimum impact on other parts. The most frequent changes are facilitated by making them input or configurable options. The best software designers identify and document not only what is likely to change, but how often and by whom. For example, a configuration or template file that may be modified by a user will be clearly distinguished from one that is considered a fixed part of the system.

Ideally the scope of anticipated change is both clearly documented and consonant with the program design. For example, we expect to see client-customizable aspects of HTML output clearly isolated and documented in a configuration file, not embedded in an XSLT script file and certainly not scattered about in print statements in the code. Thus, the choice to encode something as "data" rather than "program" should at least be a good hint that it may be a parameter for testing, although further consideration of the scope of variation may be necessary. Conversely, defining the parameters for variation in test design can be part of the architectural design process of setting the scope of variation anticipated for a given product or release.

select possible patterns for the collection as a whole.

Should the representative values of the flattened collection of pairs be *one compatible selection, one incompatible selection, all compatible selections, all incompatible selections,* or should we also include *mix of 2 or more compatible and 2 or more incompatible selections*? Certainly the latter is more thorough, but whether there is sufficient value to justify the cost of this thoroughness is a matter of judgment by the test designer.

We have oversimplified by considering only whether a selection is compatible with a slot. It might also happen that the selection does not appear in the database. Moreover, the selection might be incompatible with the model, or with a selected component of another slot, in addition to the possibility that it is incompatible with the slot for which it has been selected. If we treat each such possibility as a separate parameter characteristic, we will generate many combinations, and we will need semantic constraints to rule out combinations like *there are three options, at least two of which are compatible with the model and two of which are not, and none of which appears in the database*. On the other hand, if we simply enumerate the combinations that do make sense and are worth testing, then it becomes more difficult to be sure that no important combinations have been omitted. Like all design decisions, the way in which collections and complex data are broken into parameter characteristics requires judgment based on a combination of analysis and experience.

B. Identify Representative Values This step consists of identifying a list of representative values (more precisely, a list of classes of values) for each of the parameter characteristics identified during step *A*. Representative values should be identified for each category independently, ignoring possible interactions among values for different categories, which are considered in the next step.

Representative values may be identified by manually applying a set of rules known as boundary value testing or erroneous condition testing. The boundary value testing rule suggests selection of extreme values within a class (e.g., maximum and minimum values of the legal range), values outside but as close as possible to the class, and "interior" (non-extreme) values of the class. Values near the boundary of a class are often useful in detecting "off by one" errors in programs. The erroneous condition rule suggests selecting values that are outside the normal domain of the program, since experience suggests that proper handling of error cases is often overlooked.

Table 11.1 summarizes the parameter characteristics and the corresponding value choices identified for feature *check configuration*.[1] For numeric characteristics whose legal values have a lower bound of 1, i.e., *number of models in database* and *number of components in database*, we identify 0, the erroneous value, 1, the boundary value, and *many*, the class of values greater than 1, as the relevant value classes. For numeric characteristics whose lower bound is zero, i.e., *number of required slots for selected model* and *number of optional slots for selected model*, we identify 0 as a boundary value, 1 and *many* as other relevant classes of values. Negative values are impossible here, so we do not add a negative error choice. For numeric characteristics whose legal values have definite lower and upper-bounds, i.e., *number of optional components with*

[1] At this point, readers may ignore the items in square brackets, which indicate constraints identified in step *C* of the category-partition method.

non-empty selection and *number of optional components with non-empty selection*, we identify boundary and (when possible) erroneous conditions corresponding to both lower and upper bounds.

Identifying relevant values is an important but tedious task. Test designers may improve manual selection of relevant values by using the catalog approach described in Section 11.4, which captures the informal approaches used in this section with a systematic application of catalog entries.

C. Generate Test Case Specifications A test case specification for a feature is given as a combination of value classes, one for each identified parameter characteristic. Unfortunately, the simple combination of all possible value classes for each parameter characteristic results in an unmanageable number of test cases (many of which are impossible) even for simple specifications. For example, in the Table 11.1 we find 7 categories with 3 value classes, 2 categories with 6 value classes, and one with four value classes, potentially resulting in $3^7 \times 6^2 \times 4 = 314,928$ test cases, which would be acceptable only if the cost of executing and checking each individual test case were very small. However, not all combinations of value classes correspond to reasonable test case specifications. For example, it is not possible to create a test case from a test case specification requiring a *valid* model (a model appearing in the database) where the database contains zero models.

The category-partition method allows one to omit some combinations by indicating value classes that need not be combined with all other values. The label [*error*] indicates a value class that need be tried only once, in combination with non-error values of other parameters. When [*error*] constraints are considered in the category-partition specification of Table 11.1, the number of combinations to be considered is reduced to $1 \times 3 \times 3 \times 1 \times 1 \times 3 \times 5 \times 5 \times 2 \times 2 + 11 = 2711$. Note that we have treated "component not in database" as an error case, but have treated "incompatible with slot" as a normal case of an invalid configuration; once again, some judgment is required.

Although the reduction from 314,928 to 2,711 is impressive, the number of derived test cases may still exceed the budget for testing such a simple feature. Moreover, some values are not erroneous per se, but may only be useful or even valid in particular combinations. For example, the number of optional components with non-empty selection is relevant to choosing useful test cases only when the number of optional slots is greater than 1. A number of non-empty choices of required component greater than zero does not make sense if the number of required components is zero.

Erroneous combinations of valid values can be ruled out with the *property* and *if-property* constraints. The *property* constraint groups values of a single parameter characteristic to identify subsets of values with common properties. The *property* constraint is indicated with label *property PropertyName*, where *PropertyName* identifies the property for later reference. For example, property *RSNE* (required slots non-empty) in Table 11.1 groups values that correspond to non-empty sets of required slots for the parameter characteristic *Number of Required Slots for Selected Model (#SMRS)*, i.e., values *1* and *many*. Similarly, property *OSNE* (optional slots non-empty) groups non-empty values for the parameter characteristic *Number of Optional Slots for Selected Model (#SMOS)*.

Parameter: Model

Model number
- malformed [error]
- not in database [error]
- valid

Number of required slots for selected model(#SMRS)
- 0 [single]
- 1 [property **RSNE**] [single]
- many [property **RSNE**], [property **RSMANY**]

Number of optional slots for selected model (#SMOS)
- 0 [single]
- 1 [property **OSNE**] [single]
- many [property **OSNE**][property **OSMANY**]

Parameter: Components

Correspondence of selection with model slots
- omitted slots [error]
- extra slots [error]
- mismatched slots [error]
- complete correspondence

Number of required components with non-empty selection
- 0 [if **RSNE**] [error]
- $<$ number of required slots [if **RSNE**] [error]
- $=$ number of required slots [if **RSMANY**]

Number of optional components with non-empty selection
- 0
- $<$ number of optional slots [if **OSNE**]
- $=$ number of optional slots [if **OSMANY**]

Required component selection
- some default [single]
- all valid
- ≥ 1 incompatible with slot
- ≥ 1 incompatible with another selection
- ≥ 1 incompatible with model
- ≥ 1 not in database [error]

Optional component selection
- some default [single]
- all valid
- ≥ 1 incompatible with slot
- ≥ 1 incompatible with another selection
- ≥ 1 incompatible with model
- ≥ 1 not in database [error]

Environment element: Product database

Number of models in database (#DBM)
- 0 [error]
- 1 [single]
- many

Number of components in database (#DBC)
- 0 [error]
- 1 [single]
- many

Table 11.1: Categories and value classes derived with the category-partition method from the specification of Figure 11.1

The *if-property* constraint bounds the choices of values for a parameter characteristic that can be combined with a particular value selected for a different parameter characteristic. The *if-property* constraint is indicated with label *if PropertyName*, where *PropertyName* identifies a property defined with the *property* constraint. For example, the constraint *if RSNE* attached to value *0* of parameter characteristic *Number of required components with non-empty selection* limits the combination of this value with values *1* and *many* of the parameter characteristics *Number of Required Slots for Selected Model (#SMRS)*. In this way, we rule out illegal combinations like *Number of required components with non-empty selection = 0* with *Number of Required Slots for Selected Model (#SMRS) = 0*.

The *property* and *if-property* constraints introduced in Table 11.1 further reduce the number of combinations to be considered to $1 \times 3 \times 1 \times 1 \times (3+2+1) \times 5 \times 5 \times 2 \times 2+11 = 1811$.

The number of combinations can be further reduced by iteratively adding *property* and *if-property* constraints and by introducing the new *single* constraint, which is indicated with label *single* and acts like the *error* constraint, i.e., it limits the number of occurrences of a given value in the selected combinations to *1*.

Test designers can introduce new *property*, *if-property*, and *single* constraints to reduce the total number of combinations when needed to meet budget and schedule limits. Placement of these constraints reflects the test designer's judgment regarding combinations that are least likely to require thorough coverage.

The *single* constraints introduced in Table 11.1 reduces the number of combinations to be considered to $1 \times 1 \times 1 \times 1 \times 1 \times 3 \times 4 \times 4 \times 1 \times 1 + 19 = 67$, which may be a reasonable balance between cost and quality for the considered functionality. The number of combinations can also be reduced by applying the pairwise and n-way combination testing techniques, as explained in the next section.

The set of combinations of value classes for the parameter characteristics can be turned into test case specifications by simply instantiating the identified combinations. Table 11.2 shows an excerpt of test case specifications. The error tag in the last column indicates test case specifications corresponding to the *error* constraint. Corresponding test cases should produce an error indication. A dash indicates no constraints on the choice of values for the parameter or environment element.

Choosing meaningful names for parameter characteristics and value classes allows (semi)automatic generation of test case specifications.

11.3 Pairwise Combination Testing

However one obtains sets of value classes for each parameter characteristic, the next step in producing test case specifications is selecting combinations of classes for testing. A simple approach is to exhaustively enumerate all possible combinations of classes, but the number of possible combinations rapidly explodes.

Some methods, such as the category-partition method described in the previous section, take exhaustive enumeration as a base approach to generating combinations, but allow the test designer to add constraints that limit growth in the number of combinations. This can be a reasonable approach when the constraints on test case generation

Model#	# required slots	# optional slots	# Corr. w/ model slots	# required components	# optional components	Required components selection	Optional components selection	# Models in DB	# Components in DB	Exp result
malformed	many	many	same	EQR	0	all valid	all valid	many	many	Err
Not in DB	many	many	same	EQR	0	all valid	all valid	many	many	Err
valid	0	many	same	-	0	all valid	all valid	many	many	OK
...					...					...
valid	many	many	same	EQR	EQO	in-other	in-mod	many	many	OK
valid	many	many	same	EQR	EQO	in-mod	all valid	many	many	OK
valid	many	many	same	EQR	EQO	in-mod	in-slot	many	many	OK
valid	many	many	same	EQR	EQO	in-mod	in-other	many	many	OK
valid	many	many	same	EQR	EQO	in-mod	in-mod	many	many	OK

Legend

EQR = # req slot
EQO = # opt slot
in-mod $\geq$1 incompat w/ model
in-other $\geq$1 incompat w/ another slot
in-slot $\geq$1 incompat w/ slot

Table 11.2: An excerpt of test case specifications derived from the value classes given in Table 11.1

Display Mode full-graphics text-only limited-bandwidth	**Language** English French Spanish Portuguese	**Fonts** Minimal Standard Document-loaded
Color Monochrome Color-map 16-bit True-color	**Screen size** Hand-held Laptop Full-size	

Table 11.3: Parameters and values controlling Chipmunk web-site display

reflect real constraints in the application domain, and eliminate many redundant combinations (for example, the "error" entries in category-partition testing). It is less satisfactory when, lacking real constraints from the application domain, the test designer is forced to add arbitrary constraints (e.g., "single" entries in the category-partition method) whose sole purpose is to reduce the number of combinations.

Consider the parameters that control the Chipmunk web-site display, shown in Table 11.3. Exhaustive enumeration produces 432 combinations, which is too many if the test results (e.g., judging readability) involve human judgment. While the test designer might hypothesize some constraints, such as observing that monochrome displays are limited mostly to hand-held devices, radical reductions require adding several "single" and "property" constraints without any particular rationale.

Exhaustive enumeration of all n-way combinations of value classes for n parameters, on the one hand, and coverage of individual classes, on the other, are only the extreme ends of a spectrum of strategies for generating combinations of classes. Between them lie strategies that generate all pairs of classes for different parameters, all triples, and so on. When it is reasonable to expect some potential interaction between parameters (so coverage of individual value classes is deemed insufficient), but covering all combinations is impractical, an attractive alternative is to generate k-way combinations for $k < n$, typically pairs or triples.

How much does generating possible pairs of classes save, compared to generating all combinations? We have already observed that the number of all combinations is the product of the number of classes for each parameter, and that this product grows exponentially with the number of parameters. It turns out that the number of combinations needed to cover all possible pairs of values grows only logarithmically with the number of parameters — an enormous saving.

A simple example may suffice to gain some intuition about the efficiency of generating tuples that cover pairs of classes, rather than all combinations. Suppose we have just the three parameters *display mode, screen size,* and *fonts* from Table 11.3. If we consider only the first two, *display mode* and *screen size*, the set of all pairs and the set

Display mode × *Screen size*		*Fonts*
Full-graphics	Hand-held	Minimal
Full-graphics	Laptop	Standard
Full-graphics	Full-size	Document-loaded
Text-only	Hand-held	Standard
Text-only	Laptop	Document-loaded
Text-only	Full-size	Minimal
Limited-bandwidth	Hand-held	Document-loaded
Limited-bandwidth	Laptop	Minimal
Limited-bandwidth	Full-size	Standard

Table 11.4: Covering all pairs of value classes for three parameters by extending the cross-product of two parameters

of all combinations are identical, and contain $3 \times 3 = 9$ pairs of classes. When we add the third parameter, *fonts*, generating all combinations requires combining each value class from *fonts* with every pair of *display mode* × *screen size*, a total of 27 tuples; extending from n to $n+1$ parameters is multiplicative. However, if we are generating pairs of values from *display mode*, *screen size*, and *fonts*, we can add value classes of *fonts* to existing elements of *display mode* × *screen size* in a way that covers all the pairs of *fonts* × *screen size* and all the pairs of *fonts* × *display mode* without increasing the number of combinations at all (see Table 11.4). The key is that each tuple of three elements contains three pairs, and by careful selecting value classes of the tuples we can make each tuple cover up to three different pairs.

Table 11.5 shows 17 tuples that cover all pairwise combinations of value classes of the five parameters. The entries not specified in the table ("–") correspond to open choices. Each of them can be replaced by any legal value for the corresponding parameter. Leaving them open gives more freedom for selecting test cases.

Generating combinations that efficiently cover all pairs of classes (or triples, or ...) is nearly impossible to perform manually for many parameters with many value classes (which is, of course, exactly when one really needs to use the approach). Fortunately, efficient heuristic algorithms exist for this task, and they are simple enough to incorporate in tools.[2]

The tuples in Table 11.5 cover all pairwise combinations of value choices for the five parameters of the example. In many cases not all choices may be allowed. For example, the specification of the Chipmunk web-site display may indicate that monochrome displays are limited to hand-held devices. In this case, the tuples covering the pairs $\langle$*Monochrome*, *Laptop*$\rangle$ and $\langle$*Monochrome*, *Full-size*$\rangle$, i.e., the fifth and ninth tuples of Table 11.5, would not correspond to legal inputs. We can restrict the set of legal combinations of value classes by adding suitable constraints. Constraints can be expressed as tuples with wild-cards that match any possible value class. The patterns describe combinations that should be omitted from the sets of tuples. For example, the constraints

[2]Exercise **??** discusses the problem of computing suitable combinations to cover all pairs.

Language	*Color*	*Display Mode*	*Fonts*	*Screen Size*
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	–	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	–
French	True-color	–	–	Hand-held
Spanish	Monochrome	–	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	–	Hand-held
Portuguese	Monochrome	Text-only	–	–
Portuguese	Color-map	–	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held

Table 11.5: Covering all pairs of value classes for the five parameters

$$OMIT\langle *, *, *, Monochrome, Laptop\rangle$$

$$OMIT\langle *, *, *, Monochrome, Full\text{-}size\rangle$$

indicates that tuples that contain the pair $\langle Monochrome, Hand\text{-}held\rangle$ as values for the fourth and fifth parameter are not allowed in the relation of Table 11.3. Tuples that cover all pairwise combinations of value classes without violating the constraints can be generated by simply removing the illegal tuples and adding legal tuples that cover the removed pairwise combinations. Open choices must be bound consistently in the remaining tuples, e.g., tuple

$$\langle Portuguese, Monochrome, Text\text{-}only, \text{-}, \text{-}\rangle$$

must become

$$\langle Portuguese, Monochrome, Text\text{-}only, \text{-}, Hand\text{-}held\rangle$$

Constraints can also be expressed with sets of tables to indicate only the legal combinations, as illustrated in Table 11.6, where the first table indicates that the value class *Hand-held* for parameter *Screen* can be combined with any value class of parameter *Color*, including *Monochrome*, while the second table indicates that the value classes *Laptop* and *Full-size* for parameter *Screen size* can be combined with all values classes except *Monochrome* for parameter *Color*.

If constraints are expressed as a set of tables that give only legal combinations, tuples can be generated without changing the heuristic. Although the two approaches

express the same constraints, the number of generated tuples can be different, since different tables may indicate overlapping pairs and thus result in a larger set of tuples. Other ways of expressing constraints may be chosen according to the characteristics of the specification and the preferences of the test designer.

So far we have illustrated the combinatorial approach with pairwise coverage. As previously mentioned, the same approach can be applied for triples or larger combinations. Pairwise combinations may be sufficient for some subset of the parameters, but not enough to uncover potential interactions among other parameters. For example, in the Chipmunk display example, the fit of text fields to screen areas depends on the combination of language, fonts, and screen size. Thus, we may prefer exhaustive coverage of combinations of these three parameters, but be satisfied with pairwise coverage of other parameters. In this case, we first generate tuples of classes from the parameters to be most thoroughly covered, and then extend these with the parameters which require less coverage.[3]

11.4 Catalog Based Testing

The test design techniques described above require judgment in deriving value classes. Over time, an organization can build experience in making these judgments well. Gathering this experience in a systematic collection can speed up the process and routinize many decisions, reducing human error and better focusing human effort. Catalogs capture the experience of test designers by listing all cases to be considered for each possible type of variable that represents logical inputs, outputs, and status of the computation. For example, if the computation uses a variable whose value must belong to a range of integer values, a catalog might indicate the following cases, each corresponding to a relevant test case:

1. The element immediately preceding the lower bound of the interval
2. The lower bound of the interval
3. A non-boundary element within the interval
4. The upper bound of the interval
5. The element immediately following the upper bound

The catalog would in this way cover the intuitive cases of erroneous conditions (cases 1 and 5), boundary conditions (cases 2 and 4), and normal conditions (case 3).

The catalog based approach consists in *unfolding* the specification, i.e., decomposing the specification into elementary items, deriving an initial set of test case specifications from pre-conditions, post-conditions, and definitions, and completing the set of test case specifications using a suitable test catalog.

STEP 1: identify elementary items of the specification The initial specification is transformed into a set of elementary items. Elementary items belong to a small set of basic types:

[3] See exercise 11.6 for additional details.

Hand-held devices

Display Mode	**Language**	**Fonts**
full-graphics	English	Minimal
text-only	French	Standard
limited-bandwidth	Spanish	Document-loaded
	Portuguese	
Color		
Color-map	**Screen size**	
16-bit	Hand-held	
True-color		

Laptop and Full-size devices

Display Mode	**Language**	**Fonts**
full-graphics	English	Minimal
text-only	French	Standard
limited-bandwidth	Spanish	Document-loaded
	Portuguese	
Color		
Monochrome	**Screen size**	
Color-map	Laptop	
16-bit	Full size	
True-color		

Table 11.6: Pairs of tables that indicate valid value classes for the Chipmunk web-site display

Preconditions represent the conditions on the inputs that must be satisfied before invocation of the unit under test. Preconditions may be checked either by the unit under test (*validated preconditions*) or by the caller (*assumed preconditions*).

Postconditions describe the result of executing the unit under test.

Variables indicate the values on which the unit under test operates. They can be input, output, or intermediate values.

Operations indicate the main operations performed on input or intermediate variables by the unit under test

Definitions are shorthand used in the specification

As in other approaches that begin with an informal description, it is not possible to give a precise recipe for extracting the significant elements. The result will depend on the capability and experience of the test designer.

Consider the informal specification of a function for converting URL-encoded form data into the original data entered through an html form. An informal specification is given in Figure 11.2.[4]

The informal description of cgi_decode uses the concept of hexadecimal digit, hexadecimal escape sequence, and element of a cgi encoded sequence. This leads to the identification of the following three definitions:

DEF 1 *hexadecimal digits* are: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'a', 'b', 'c', 'd', 'e', 'f'

DEF 2 a *CGI-hexadecimal* is a sequence of three characters: "`%xy`", where `x` and `y` are hexadecimal digits

DEF 3 a *CGI item* is either an alphanumeric character, or character '+', or a CGI-hexadecimal

In general, every concept introduced in the description to define the problem can be represented as a definition.

The description of cgi_decode mentions some elements that are inputs and outputs of the computation. These are identified as the following variables:

VAR 1 *Encoded*: string of ASCII characters

VAR 2 *Decoded*: string of ASCII characters

VAR 3 *return value*: boolean

Note the distinction between a variable and a definition. *Encoded* and *decoded* are actually used or computed, while *hexadecimal digits*, *CGI-hexadecimal*, and *CGI item* are used to describe the elements but are not objects in their own right. Although not

[4]The informal specification is ambiguous and inconsistent, i.e., it is the kind of spec one is most likely to encounter in practice.

cgi_decode: Function cgi_decode translates a cgi-encoded string to a plain ASCII string, reversing the encoding applied by the common gateway interface (CGI) of most web servers.

CGI translates spaces to '+', and translates most other non-alphanumeric characters to hexadecimal escape sequences. `cgi_decode` maps '+' to ' ', "%xy" (where x and y are hexadecimal digits) to to the corresponding ASCII character, and other alphanumeric characters to themselves.

INPUT: encoded A string of characters, representing the input CGI sequence. It can contain:

- alphanumeric characters
- the character '+'
- the substring "%xy", where x and y are hexadecimal digits.

encoded is terminated by a null character.

OUTPUT: decoded A string containing the plain ASCII characters corresponding to the input CGI sequence.

- Alphanumeric characters are copied into the output in the corresponding position
- A blank is substituted for each '+' character in the input.
- A single ASCII character with hexadecimal value xy_{16} is substituted for each substring "%xy" in the input.

OUTPUT: return value `cgi_decode` returns

- 0 for success
- 1 if the input is malformed

Figure 11.2: An informal (and imperfect) specification of function cgi-decode

strictly necessary for the problem specification, explicit identification of definitions can help in deriving a richer set of test cases.

The description of cgi_decode indicates some conditions that must be satisfied upon invocation, represented by the following preconditions:

PRE 1 *(Assumed)* the input string *Encoded* is a null-terminated string of characters.

PRE 2 *(Validated)* the input string *Encoded* is a sequence of CGI items.

In general, preconditions represent all the conditions that should be true for the intended functioning of a module. A condition is labeled as *validated* if it is checked by the module (in which case a violation has a specified effect, e.g., raising an exception or returning an error code). *Assumed* preconditions must be guaranteed by the caller, and the module does not guarantee a particular behavior in case they are violated.

The description of cgi_decode indicates several possible results. These can be represented as a set of postconditions:

POST 1 if the input string *Encoded* contains alphanumeric characters, they are copied to the corresponding position in the output string.

POST 2 if the input string *Encoded* contains '+' characters, they are replaced by ASCII space characters in the corresponding positions in the output string.

POST 3 if the input string *Encoded* contains CGI-hexadecimals, they are replaced by the corresponding ASCII characters.

POST 4 if the input string *Encoded* is a valid sequence, cgi_decode returns 0.

POST 5 if the input string *Encoded* contains a malformed CGI-hexadecimal, i.e., a substring "%xy", where either x or y is absent or are not hexadecimal digits, cgi_decode returns 1

POST 6 if the input string *Encoded* contains any illegal character, cgi_decode returns 1.

The postconditions should, together, capture all the expected outcomes of the module under test. When there are several possible outcomes, it is possible to capture all of them in one complex postcondition or in several simple postconditions; here we have chosen a set of simple contingent postconditions, each of which captures one case. The informal specification does not distinguish among cases of malformed input strings, but the test designer may make further distinctions while refining the specification.

Although the description of cgi_decode does not mention explicitly how the results are obtained, we can easily deduce that it will be necessary to scan the input sequence. This is made explicit in the following operation:

OP 1 Scan the input string *Encoded*.

PRE 1	(*Assumed*) the input string Encoded is a null-terminated string of characters
PRE 2	(*Validated*) the input string Encoded is a sequence of CGI items
POST 1	if the input string Encoded contains alphanumeric characters, they are copied to the output string in the corresponding positions.
POST 2	if the input string Encoded contains '+' characters, they are replaced in the output string by ASCII space characters in the corresponding positions
POST 3	if the input string Encoded contains CGI-hexadecimals, they are replaced by the corresponding ASCII characters.
POST 4	if the input string Encoded is well-formed, cgi-decode returns 0
POST 5	if the input string Encoded contains a malformed CGI hexadecimal, i.e., a substring "%xy", where either x or y are absent or are not hexadecimal digits, cgi_decode returns 1
POST 6	if the input string Encoded contains any illegal character, cgi_decode returns 1
VAR 1	Encoded: a string of ASCII characters
VAR 2	Decoded: a string of ASCII characters
VAR 3	Return value: a boolean
DEF 1	hexadecimal digits are ASCII characters in range ['0' .. '9', 'A' .. 'F', 'a' .. 'f']
DEF 2	CGI-hexadecimals are sequences "%xy", where x and y are hexadecimal digits
DEF 3	A CGI item is an alphanumeric character, or '+', or a CGI-hexadecimal
OP 1	Scan Encoded

Figure 11.3: Elementary items of specification cgi-decode

In general, a description may refer either explicitly or implicitly to elementary operations which help to clearly describe the overall behavior, like definitions help to clearly describe variables. As with variables, they are not strictly necessary for describing the relation between pre- and postconditions, but they serve as additional information for deriving test cases.

The result of step 1 for cgi_decode is summarized in Figure 11.3.

STEP 2 Derive a first set of test case specifications from preconditions, postconditions and definitions The aim of this step is to explicitly describe the partition of the input domain:

Validated Preconditions: A simple precondition, i.e., a precondition that is expressed as a simple boolean expression without *and* or *or*, identifies two classes of input: values that satisfy the precondition and values that do not. We thus derive two test case specifications.

A compound precondition, given as a boolean expression with *and* or *or*, identifies several classes of inputs. Although in general one could derive a different

test case specification for each possible combination of truth values of the elementary conditions, usually we derive only a subset of test case specifications using the modified condition decision coverage (*MC/DC*) approach, which is illustrated in Section 14.3 and in Chapter 12. In short, we derive a set of combinations of elementary conditions such that each elementary condition can be shown to independently affect the outcome of each decision. For each elementary condition *C*, there are two test case specifications in which the truth values of all conditions except *C* are the same, and the compound condition as a whole evaluates to **True** for one of those test cases and **False** for the other.

Assumed Preconditions: We do not derive test case specifications for cases that violate assumed preconditions, since there is no defined behavior and thus no way to judge the success of such a test case. We also do not derive test cases when the whole input domain satisfies the condition, since test cases for these would be redundant. We generate test cases from assumed preconditions only when the MC/DC criterion generates more than one class of valid combinations (i.e., when the condition is a logical disjunction of more elementary conditions).

Postconditions: In all cases in which postconditions are given in a conditional form, the condition is treated like a validated precondition, i.e., we generate a test case specification for cases that satisfy and cases that do not satisfy the condition.

Definition: Definitions that refer to input or output values and are given in conditional form are treated like validated preconditions. We generate a set of test case specification for cases that satisfy and cases that do not satisfy the specification. The test cases are generated for each variable that refers to the definition.

The elementary items of the specification identified in step 1 are scanned sequentially and a set of test cases is derived applying these rules. While scanning the specifications, we generate test case specifications incrementally. When new test case specifications introduce a refinement of an existing case, or vice versa, the more general case becomes redundant and can be eliminated. For example, if an existing test case specification requires a non-empty set, and we have to add two test case specifications that require a size that is a power of two and one which is not, the existing test case specification can be deleted because the new test cases must include a non-empty set.

Scanning the elementary items of the cgi_decode specification given in Figure 11.3, we proceed as follows:

PRE 1: The first precondition is a simple assumed precondition. We do not generate any test case specification. The only condition would be "encoded: a null terminated string of characters," but this matches every test case and thus it does not identify a useful test case specification.

PRE 2: The second precondition is a simple validated precondition. We generate two test case specifications, one that satisfies the condition and one that does not:

TC-PRE2-1 Encoded: a sequence of CGI items

TC-PRE2-2 Encoded: not a sequence of CGI items

All postconditions in the cgi_decode specification are given in a conditional form with a simple condition. Thus, we generate two test case specifications for each of them. The generated test case specifications correspond to a case that satisfies the condition and a case that violates it.

POST 1:

TC-POST1-1 Encoded: contains one or more alphanumeric characters

TC-POST1-2 Encoded: does not contain any alphanumeric characters

POST 2:

TC-POST2-1 Encoded: contains one or more character '+'

TC-POST2-2 Encoded: does not any contain character '+'

POST 3:

TC-POST3-1 Encoded: contains one or more CGI-hexadecimals

TC-POST3-2 Encoded: does not contain any CGI-hexadecimal

POST 4: We do not generate any new useful test case specifications, because the two specifications are already covered by the specifications generated from *POST 2*.

POST 5: We generate only the test case specification that satisfies the condition. The test case specification that violates the specification is redundant with respect to the test case specifications generated from *POST 3*

TC-POST5-1 : Encoded contains one or more malformed CGI-hexadecimals

POST 6: As for *POST 5*, we generate only the test case specification that satisfies the condition. The test case specification that violates the specification is redundant with respect to several of the test case specifications generated so far.

TC-POST6-1 Encoded: contains one or more illegal characters

None of the definitions in the specification of cgi_decode is given in conditional terms, and thus no test case specifications are generated at this step.

The test case specifications generated from postconditions refine test case specification TC-PRE2-1, which can thus be eliminated from the checklist. The result of step 2 for cgi_decode is summarized in Figure 11.4.

STEP 3 Complete the test case specifications using catalogs The aim of this step is to generate additional test case specifications from variables and operations used or defined in the computation. The catalog is scanned sequentially. For each entry of the catalog we examine the elementary components of the specification and add cases to cover all values in the catalog. As when scanning the test case specifications during step 2, redundant test case specifications are eliminated.

PRE 2	*Validated*) the input string Encoded is a sequence of CGI items
[TC-PRE2-2]	Encoded: not a sequence of CGI items
POST 1	if the input string Encoded contains alphanumeric characters, they are copied to the output string in the corresponding positions
[TC-POST1-1]	Encoded: contains alphanumeric characters
[TC-POST1-2]	Encoded: does not contain alphanumeric characters
POST 2	if the input string Encoded contains '+' characters, they are replaced in the output string by ' ' in the corresponding positions
[TC-POST2-1]	Encoded: contains '+'
[TC-POST2-2]	Encoded: does not contain '+'
POST 3	if the input string Encoded contains CGI-hexadecimals, they are replaced by the corresponding ASCII characters.
[TC-POST3-1]	Encoded: contains CGI-hexadecimals
[TC-POST3-2]	Encoded: does not contain a CGI-hexadecimal
POST 4	if the input string Encoded is well-formed, cgi_decode returns 0
POST 5	if the input string Encoded contains a malformed CGI-hexadecimal, i.e., a substring "%xy", where either x or y are absent or non hexadecimal digits, cgi_decode returns 1
[TC-POST5-1]	Encoded: contains malformed CGI-hexadecimals
POST 6	if the input string Encoded contains any illegal character, cgi_decode returns a positive value
[TC-POST6-1]	Encoded: contains illegal characters
VAR 1	Encoded: a string of ASCII characters
VAR 2	Decoded: a string of ASCII characters
VAR 3	Return value: a boolean
DEF 1	hexadecimal digits are in range ['0' .. '9', 'A' .. 'F', 'a' .. 'f']
DEF 2	CGI-hexadecimals are sequences '%xy', where x and y are hexadecimal digits
DEF 3	CGI items are either alphanumeric characters, or '+', or CGI-hexadecimals
OP 1	Scan Encoded

Figure 11.4: Test case specifications for cgi-decode generated after step 2

.

Table 11.7 shows a simple catalog that we will use for the cgi_decoder example. A catalog is structured as a list of kinds of elements that can occur in a specification. Each catalog entry is associated with a list of generic test case specifications appropriate for that kind of element. We scan the specification for elements whose type is compatible with the catalog entry, then generate the test cases defined in the catalog for that entry. For example, the catalog of Table 11.7 contains an entry for boolean variables. When we find a boolean variable in the specification, we instantiate the catalog entry by generating two test case specifications, one that requires a **True** value and one that requires a **False** value.

Each generic test case in the catalog is labeled *in*, *out*, or *in/out*, meaning that a test case specification is appropriate if applied to an input variable, or to an output variable, or in both cases. In general, erroneous values should be used when testing the behavior of the system with respect to input variables, but are usually impossible to produce when testing the behavior of the system with respect to output variables. For example, when the value of an input variable can be chosen from a set of values, it is important to test the behavior of the system for all enumerated values and some values outside the enumerated set, as required by entry *ENUMERATION* of the catalog. However, when the value of an output variable belongs to a finite set of values, we should derive a test case for each possible outcome, but we cannot derive a test case for an impossible outcome, so entry *ENUMERATION* of the catalog specifies that the choice of values outside the enumerated set is limited to input variables. Intermediate variables, if present, are treated like output variables.

Entry *Boolean* of the catalog applies to Return value (VAR 3). The catalog requires a test case that produces the value **True** and one that produces the value **False**. Both cases are already covered by test cases *TC-PRE2-1* and *TC-PRE2-2* generated for precondition *PRE 2*, so no test case specification is actually added.

Entry *Enumeration* of the catalog applies to any variable whose values are chosen from an explicitly enumerated set of values. In the example, the values of CGI item (DEF 3) and of improper CGI hexadecimals in POST 5 are defined by enumeration. Thus, we can derive new test case specifications by applying entry *enumeration* to POST 5 and to any variable that can contain CGI items.

The catalog requires creation of a test case specification for each enumerated value and for some excluded values. For encoded, which should consist of CGI-items as defined in DEF 3, we generate a test case specification where a CGI-item is an alphanumeric character, one where it is the character '+', one where it is a CGI-hexadecimal, and one where it is an illegal value. We can easily ascertain that all the required cases are already covered by test case specifications for *TC-POST1-1*, *TC-POST1-2*, *TC-POST2-1*, *TC-POST2-2*, *TC-POST3-1*, and *TC-POST3-2*, so any additional test case specifications would be redundant.

From the enumeration of malformed CGI-hexadecimals in POST 5, we derive the following test cases: %y, %x, %ky, %xk, %xy (where x and y are hexadecimal digits and k is not). Note that the first two cases, %x (the second hexadecimal digit is missing) and %y (the first hexadecimal digit is missing) are identical, and %x is distinct from %xk only if %x are the last two characters in the string. A test case specification requiring a correct pair of hexadecimal digits (%xy) is a value out of the range of the

Boolean	
[in/out]	True
[in/out]	False
Enumeration	
[in/out]	Each enumerated value
[in]	Some value outside the enumerated set
Range $L \ldots U$	
[in]	$L-1$ (the element immediately preceding the lower bound)
[in/out]	L (the lower bound)
[in/out]	A value between L and U
[in/out]	U (the upper bound)
[in]	$U+1$ (the element immediately following the upper bound)
Numeric Constant C	
[in/out]	C (the constant value)
[in]	$C-1$ (the element immediately preceding the constant value)
[in]	$C+1$ (the element immediately following the constant value)
[in]	Any other constant compatible with C
Non-Numeric Constant C	
[in/out]	C (the constant value)
[in]	Any other constant compatible with C
[in]	Some other compatible value
Sequence	
[in/out]	Empty
[in/out]	A single element
[in/out]	More than one element
[in/out]	Maximum length (if bounded) or very long
[in]	Longer than maximum length (if bounded)
[in]	Incorrectly terminated
Scan with action on elements P	
[in]	P occurs at beginning of sequence
[in]	P occurs in interior of sequence
[in]	P occurs at end of sequence
[in]	PP occurs contiguously
[in]	P does not occur in sequence
[in]	pP where p is a proper prefix of P
[in]	Proper prefix p occurs at end of sequence

Table 11.7: Part of a simple test catalog.

enumerated set, as required by the catalog.

The added test case specifications are:

TC-POST5-2 encoded: terminated with %x, where x is a hexadecimal digit

TC-POST5-3 encoded: contains %ky, where k is not a hexadecimal digit and y is a hexadecimal digit.

TC-POST5-4 encoded: contains %xk, where x is a hexadecimal digit and k is not.

The test case specification corresponding to the correct pair of hexadecimal digits is redundant, having already been covered by TC-POST3-1. The test case TC-POST5-1 can now be eliminated because it is more general than the combination of TC-POST5-2, TC-POST5-3, and TC-POST5-4.

Entry *Range* applies to any variable whose values are chosen from a finite range. In the example, ranges appear three times in the definition of hexadecimal digit. Ranges also appear implicitly in the reference to alphanumeric characters (the alphabetic and numeric ranges from the ASCII character set) in DEF 3. For hexadecimal digits we will try the special values '/' and ':' (the characters that appear before '0' and after '9' in the ASCII encoding), the values '0' and '9' (upper and lower bounds of the first interval), some value between '0' and '9'; similarly '@', 'G', 'A', 'F', and some value between 'A' and 'F' for the second interval; and '`', 'g', 'a', 'f', and some value between 'a' and 'f' for the third interval.

These values will be instantiated for variable encoded, and result in 30 additional test case specifications (5 values for each subrange, giving 15 values for each hexadecimal digit and thus 30 for the two digits of CGI-hexadecimal). The full set of test case specifications is shown in Table 11.8. These test case specifications are more specific than (and therefore replace) test case specifications TC-POST3-1, TC-POST5-3, and TC-POST5-4.

For alphanumeric characters we will similarly derive boundary, interior and excluded values, which result in 15 additional test case specifications, also given in Table 11.8. These test cases are more specific than (and therefore replace) TC-POST1-1, TC-POST1-2, and TC-POST6-1.

Entry *Numeric Constant* does not apply to any element of this specification.

Entry *Non-Numeric Constant* applies to '+' and '%', occurring in DEF 3 and DEF 2 respectively. Six test case specifications result, but all are redundant.

Entry *Sequence* applies to encoded (VAR 1), decoded (VAR 2), and cgi-item (DEF 2). Six test case specifications result for each, of which only five are mutually non-redundant and not already in the list. From VAR 1 (encoded) we generate test case specifications requiring an empty sequence, a sequence containing a single element, and a very long sequence. The catalog entry requiring more than one element generates a redundant test case specification, which is discarded. We cannot produce reasonable test cases for incorrectly terminated strings (the behavior would vary depending on the contents of memory outside the string), so we omit that test case specification.

All test case specifications that would be derived for decoded (VAR 2) would be redundant with respect to test case specifications derived for encoded (VAR 1).

From CGI-hexadecimal (DEF 2) we generate two additional test case specifications for variable encoded: a sequence that terminates with '%' (the only way to produce a one-character subsequence beginning with '%') and a sequence containing '%xyz', where x, y, and z are hexadecimal digits.

Entry *Scan* applies to Scan Encoded (OP 1) and generates 17 test case specifications. Three test case specifications (alphanumeric, '+', and CGI item) are generated for each of the first 5 items of the catalog entry. One test case specification is generated for each of the last two items of the catalog entry when *Scan* is applied to CGI item. The last two items of the catalog entry do not apply to alphanumeric characters and '+', since they have no non-trivial prefixes. Seven of the 17 are redundant. The ten generated test case specifications are summarized in Table 11.8.

Test catalogs, like other check lists used in test and analysis (e.g., inspection check lists), are an organizational asset that can be maintained and enhanced over time. A good test catalog will be written precisely and suitably annotated to resolve ambiguity. Catalogs should also be specialized to an organization and application domain, typically using a process such as defect causal analysis or root cause analysis (Chapters 20 and 18). Entries are added to detect particular classes of faults that have been encountered frequently or have been particularly costly to remedy in previous projects. Refining check lists is a typical activity carried out as part of process improvement. When a test reveals a program fault, it is useful to make a note of which catalog entries the test case originated from, as an aid to measuring the effectiveness of catalog entries. Catalog entries that are not effective should be removed.

Open research issues

In the last decades, structured languages replaced natural language in software specifications, and today unstructured specifications written in natural language are becoming less common. Unstructured natural language specifications are still commonly used in informal development environments that lack expertise and tools, and often do not adopt rigorous development methodologies. Deriving structure from natural language is not a main focus of the research community, which pays more attention to exploiting formal and semi-formal models that may be produced in the course of a project.

Combinatorial methods per se is a niche research area that attracts relatively little attention from the research community. One issue that has received too little attention to date is adapting combinatorial test techniques to cope with constantly changing specifications.

Further Reading

Category partition testing is described by Ostrand and Balcer [OB88]. The combinatorial approach described in this chapter is due to Cohen, Dalal, Fredman, and Patton [CDFP97]; the algorithm described by Cohen et al. is patented by Bellcore. Catalog-based testing of subsystems is described in Marick's *The Craft of Software Testing* [Mar97].

TC-POST2-1	Encoded contains character '+'
TC-POST2-2	Encoded does not contain character '+'
TC-POST3-2	Encoded does not contain a CGI-hexadecimal
TC-POST5-2	Encoded terminates with %x
TC-VAR1-1	Encoded is the empty sequence
TC-VAR1-2	Encoded is a sequence containing a single character
TC-VAR1-3	Encoded is a very long sequence
TC-DEF2-1	Encoded contains '%/y'
TC-DEF2-2	Encoded contains '%0y'
TC-DEF2-3	Encoded contains '%xy', with x in [1..8]
TC-DEF2-4	Encoded contains '%9y'
TC-DEF2-5	Encoded contains '%:y'
TC-DEF2-6	Encoded contains '%@y'
TC-DEF2-7	Encoded contains '%Ay'
TC-DEF2-8	Encoded contains '%xy', with x in [B..E]
TC-DEF2-9	Encoded contains '%Fy'
TC-DEF2-10	Encoded contains '%Gy'
TC-DEF2-11	Encoded contains '%`y'
TC-DEF2-12	Encoded contains '%ay'
TC-DEF2-13	Encoded contains '%xy', with x in [b..e]
TC-DEF2-14	Encoded contains '%fy'
TC-DEF2-15	Encoded contains '%gy'
TC-DEF2-16	Encoded contains '%x/'
TC-DEF2-17	Encoded contains '%x0'
TC-DEF2-18	Encoded contains '%xy', with y in [1..8]
TC-DEF2-19	Encoded contains '%x9'
TC-DEF2-20	Encoded contains '%x:'
TC-DEF2-21	Encoded contains '%x@'
TC-DEF2-22	Encoded contains '%xA'
TC-DEF2-23	Encoded contains '%xy', with y in [B..E]
TC-DEF2-24	Encoded contains '%xF'
TC-DEF2-25	Encoded contains '%xG'
TC-DEF2-26	Encoded contains '%x`'
TC-DEF2-27	Encoded contains '%xa'
TC-DEF2-28	Encoded contains '%xy', with y in [b..e]
TC-DEF2-29	Encoded contains '%xf'
TC-DEF2-30	Encoded contains '%xg'
TC-DEF2-31	Encoded contains '%$'
TC-DEF2-32	Encoded contains '%xyz'
TC-DEF3-1	Encoded contains '/'
TC-DEF3-2	Encoded contains '0'
TC-DEF3-3	Encoded contains c, with c in ['1'..'8']
TC-DEF3-4	Encoded contains '9'
TC-DEF3-5	Encoded contains ':'
TC-DEF3-6	Encoded contains '@'
TC-DEF3-7	Encoded contains 'A'
TC-DEF3-8	Encoded contains c, with c in ['B'..'Y']
TC-DEF3-9	Encoded contains 'Z'
TC-DEF3-10	Encoded contains '['
TC-DEF3-11	Encoded contains '`'
TC-DEF3-12	Encoded contains 'a'
TC-DEF3-13	Encoded contains c, with c in ['b'..'y']
TC-DEF3-14	Encoded contains 'z'
TC-DEF3-15	Encoded contains '{'
TC-OP1-1	Encoded contains '^*a*'
TC-OP1-2	Encoded contains '^+'
TC-OP1-3	Encoded contains ^%xy'
TC-OP1-4	Encoded contains '*a*$'
TC-OP1-5	Encoded contains '+$'
TC-OP1-6	Encoded contains '%xy$'
TC-OP1-7	Encoded contains '*aa*'
TC-OP1-8	Encoded contains '++'
TC-OP1-9	Encoded contains '%xy%zw'
TC-OP1-10	Encoded contains '%x%yz'

where w,x,y,z are hexadecimal digits, a is an alphanumeric character, ^ represents the beginning of the string, and $ represents the end of the string.

Table 11.8: Summary table: Test case specifications for cgi-decode generated with a catalog.

Related topics

Readers interested in learning additional functional testing techniques may continue with the next Chapter that describes model-based testing techniques. Readers interested in the complementarities between functional and structural testing as well as readers interested in testing the decision structures and control and data flow graphs may continue with the following chapters that describe structural and data flow testing. Readers interested in the quality of specifications may proceed to Chapter 18, which describes inspection techniques.

Exercises

11.1. When designing a test suite with the category partition method, sometimes it is useful to determine the number of test case specifications that would be generated from a set of parameter characteristics (categories) and value classes (choices) without actually generating or enumerating them. Describe how to quickly determine the number of test cases in these cases:

(a) Parameter characteristics and value classes are given, but no constraints (*error*, *single*, *property*, or *if-property*) are used.

(b) Only the constraints *error* and *single* are used (without *property* and *if-property*).

When the *property* and *if-property* are also used, they can interact in ways that make a quick closed-form calculation of the number of test cases difficult or impossible.

(c) Sketch an algorithm for counting the number of test cases that would be generated when *if* and *if-property* are used. Your algorithm should be simple, and may not be more efficient than actually generating each test case specification.

11.2. Derive parameter characterists, representative values, and semantic constraints from the following specification of an *Airport connection check* function, suitable for generating a set of test case specifications using the category partition method.

> **Airport connection check:** The airport connection check is part of an (imaginary) travel reservation system. It is intended to check the validity of a single connection between two flights in an itinerary. It is described here at a fairly abstract level, as it might be described in a preliminary design before concrete interfaces have been worked out.
>
> **Specification Signature:** Valid_Connection (Arriving_Flight: flight, Departing_Flight: flight) returns Validity_Code
> Validity_Code 0 (OK) is returned if Arriving_Flight and Departing_Flight

make a valid connection (the arriving airport of the first is the departing airport of the second) and there is sufficient time between arrival and departure according to the information in the airport database described below.
Otherwise, a validity code other than 0 is returned, indicating why the connection is not valid.
Data types
Flight: A ”flight” is a structure consisting of

- A unique identifying flight code, three alphabetic characters followed by up to four digits. (The flight code is not used by the *valid connection* function.)
- The originating airport code (3 characters, alphabetic)
- The scheduled departure time of the flight (in universal time)
- The destination airport code (3 characters, alphabetic)
- The scheduled arrival time at the destination airport.

Validity Code: The validity code is one of a set of integer values with the following interpretations

0: The connection is valid.

10: Invalid airport code (airport code not found in database)

15: Invalid connection, too short: There is insufficient time between arrival of first flight and departure of second flight.

16: Invalid connection, flights do not connect. The destination airport of Arriving_Flight is not the same as the originating airport of Departing_Flight.

20: Another error has been recognized (e.g., the input arguments may be invalid, or an unanticipated error was encountered).

Airport Database
The Valid_Connection function uses an internal, in-memory table of airports which is read from a configuration file at system initialization. Each record in the table contains the following information:

- Three-letter airport code. This is the key of the table and can be used for lookups.
- Airport zone. In most cases the airport zone is a two-letter country code, e.g., ”us” for the United States. However, where passage from one country to another is possible without a passport, the airport zone represents the complete zone in which passport-free travel is allowed. For example, the code ”eu” represents the European countries which are treated as if they were a single country for purposes of travel.
- Domestic connect time. This is an integer representing the minimum number of minutes that must be allowed for a domestic connection at the airport. A connection is ”domestic” if the originating and destination airports of both flights are in the same airport zone.

- International connect time. This is an integer representing the minimum number of minutes that must be allowed for an international connection at the airport. The number -1 indicates that international connections are not permitted at the airport. A connection is ”international” if any of the originating or destination airports are in different zones.

11.3. Consider the value classes obtained by applying the category partition approach to the *Airport Connection Check* example of Exercise 11.2. Eliminate from the test specifications all constraints that do not correspond to infeasible tuples and compute the number of derivable test cases. Apply the combinatorial approach to derive test cases covering all pairwise combinations, and compare the number of derived test cases.

11.4. Derive a set of test cases for the *Airport Connection Check* example of Exercise 11.2 using the catalog based approach.
Extend the catalog of Table 11.7 as needed to deal with specification constructs.

11.5. Given a set of parameter characteristics and value classes, compute a lower bound on the number of tuples required for covering all pairs of values.

11.6. Generate a set of tuples that cover all pairwise combinations of parameters for in Table 11.3.