Chapter 7

Functional Testing

7.1 Basic Idea

William Howden developed this method for testing programs while visiting the International Mathematics and Statistics Libraries (IMSL) Inc., Houston in 1977-1978. IMSL Inc. is presently known as *Visual Numerics Inc.* (http://www.vni.com/index.html). The following text is from their website:

```
Fortune 100 companies  have relied  upon IMSL for numerical
analysis applications for over 30 years. The IMSL Numerical
Libraries  are accurate  and  reliable.  IMSL  uses  proven
technology that has been thoroughly tested, well documented,
continuously maintained and used by developers worldwide.

The IMSL  libraries are a comprehensive set of mathematical
and statistical  functions that  programmers can embed into
their software applications. ..... These libraries free you
from  developing  your  own  internal  code  by  providing
pre-written  mathematical  and statistical  algorithms that
you can embed into your C, Java and Fortran applications.''
```

Howden applied the idea of functional testing to programs from Edition 5 of the IMSL package. IMSL is a well documented and well maintaine package. The errors he discovered can be considered to be of some subtlety to have survived to Edition 5 status. The result of his work has been published in *IEEE Transactions on Software Engineering, Vol. SE-6, No. 2, March, 1980, pp. 162-169.*

In mathematics, a *function* is defined to be a set of ordered pairs (X_i, Y_i), where X_i is a vector of input values and Y_i is a vector of output values. In functional testing, a program P is viewed as a *function* that transforms the input vector X_i into an output vector Y_i, such that $Y_i = P(X_i)$. Some instances of P are as follows:

Example 1: Let $Y_1 = \sqrt{(X_1)}$. Here, P–a square-root computing function–computes the square-root Y_1 of a non-negative integer X_1.

Example 2: Let Y_2 = C_compiler(X_2). Here, P–a C compiler– produces object code Y_2 after compiling the C program X_2.

Example 3: Let Y_3 = TelephoneSwitch(X_3). Here, P–a telephone switch–produces a variety of tones and voice signals represented by the vector
Y_2 = `{idle, dial, ring, fast busy, slow busy tone, voice}`
by processing input data represented by the vector
X_3 = `{off hook, on hook, phone number, voice}`.

Example 4: Let Y_4 = sort(X_4). Here, P–an implementation of a sorting algorithm–produces a sorted array Y_4 from the input vector X_4 = `{A, N}`, where `A` is the array to be sorted and `N` is the number of elements in `A`.

The above four examples suggest that sometimes it is easy to view a program as a function in the mathematical sense and sometimes it is more difficult. It is easier when the input values are algorithmically, or mathematically, transformed into output values, such as in **Example 1** and **Example 4** above. In fact, Y_4 is a certain permutation of the input array `A`. It is more difficult when the input values are not directly transformed into the output values. For instance, in **Example 3**, an `off hook` input is not mathematically transformed into a `dial tone` output.

However, in functional testing we are not concerned with the details of how an input vector is transformed into an output vector. Rather, in functional testing

test data are selected on the basis of the important properties of the elements in the domain of a program's input and output variables.

Therefore, the two key concepts in functional testing are as follows:

- Precisely identify the domain of each input and output variable.
- Select values from a data domain having *important* properties.

Precise identification of the domain of an input or output variable is done by analyzing the specification and the design documents. In the following section, we discuss how to select test data from the domain of a variable.

Note: Here we remind the reader of the scope of functional testing. Since we analyze the input and output data domains in functional testing without considering the details of a program, the idea of functional testing can be applied to an entire program as well as to individual components (e.g. function, procedure, or method) of the entire program. Thus, functional testing can be performed both at the *unit* and *system* levels.

7.2 Different types of variables

In this section, we consider numeric variables, arrays, substructures, and strings, and their *important* values.

7.2.1 Numeric variable

The domain of a numeric variable is specified in one of two ways as explained below.

Discrete values: The data domain consists of a set of discrete values. In this case, functional testing requires us to identify such a set.

Contiguous segments of values: The data domain consists of one or more segments of integers or real numbers. In this case, functional testing requires us to identify the minimum (MIN) and maximum (MAX) values of each contiguous segment.

Example: In Figure 7.1, we show the inputs and output variables of the *Frequency Selection Box* (FSB) module of the Bluetooth wireless communication system. Bluetooth technology uses frequency hopping spread spectrum technique for accessing the wireless medium. A piconet *channel* is viewed as a, possibly infinite, sequence of *slots*, where a slot is 625 μs in length. The frequency on which a data packet will be transmitted during a given slot is computed by the FSB shown in Figure 7.1. The FSB module accepts three input variables `MODE`, `CLOCK`, and `ADDRESS`, and generates values of the output variable `INDEX`. All the four variables are numeric variables. The domains of these variables are characterized in the following.

`MODE`: The domain of variable `MODE` is the discrete set `{23, 79}`.

`CLOCK`: The `CLOCK` variable is represented by a 28-bit unsigned number. The smallest increment in `CLOCK` represents the elapse of 312.5 μs. The FSB module uses the *upper* 27 bits of the 28-bit `CLOCK`. Therefore, the range of `CLOCK` is specified as follows.

- MIN = `0x0000000` or `0x0000001`
- MAX = `0xFFFFFFE` or `0xFFFFFFF`

`ADDRESS`: The `CLOCK` variable is represented by a 48-bit unsigned number. The FSB module uses the *lower* 28 bits of the 48-bit `ADDRESS`. Therefore, the range of `ADDRESS` from the viewpoint of the FSB module is specified as follows.

- MIN = `0xyyyyy0000000`, where `yyyyy` is a 20-bit arbitrary value.
- MAX = `0xzzzzzFFFFFFF`, where `zzzzz` is a 20-bit arbitrary value.

`INDEX`: This variable assumes values in a given range as specified in the following.

- MIN = `0`
- MAX = `22` if `MODE = 23`
- MAX = `78` if `MODE = 79`

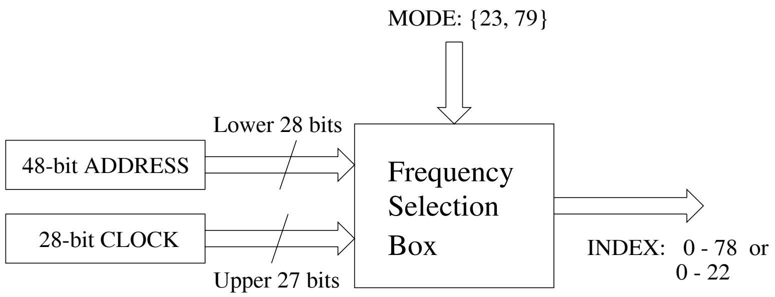


Figure 7.1: Frequency selection box of Bluetooth specification.

Once we characterize the domain of a numeric variable as a discrete set of values or as a set of contiguous segments, test data are selected based on the following *selection criteria*.

Selection criteria for input variables

Discrete domain: If the input domain is a discrete set of variables, then tests are carried out which involve each of the values. Referring to Figure 7.1, the domain of the input variable `MODE` consists of a discrete set of values `{23, 79}`. Therefore, the FSB module must be tested with both the values of `MODE`.

Contiguous domain: if the domain of the variable consists of one or more *intervals* of numbers, then a program is tested as follows.

- Consider the minimum value of the interval.
- Consider the maximum value of the interval.
- Consider a value interior to the interval.
- Consider certain values which have special mathematical properties. These values include
  - 0
  - 1
  - real numbers with *small* absolute values.
- A program must be tested with input values lying outside the domains of the input variables. Here the idea is to observe the behavior of the program in response of *illegal* values.

In case the conceptual minimum value of a variable is $-\infty$ and the conceptual maximum value is $+\infty$, then a very large negative and a very large positive value are chosen as the endpoints of the interval.

Selection criteria for output variables

Discrete domain: If the domain of an output variable consists of a *small* set of discrete values, then the program must be tested with input which result in the generation of each of the output values. For a *large* set of discrete values, one may test the program with input resulting in a number of different output values.

Contiguous domain: If an output variable has a domain which consists of one or more intervals of numbers, then the program needs to be tested with input values which result in the generation of the following output values.

- minimum values of the intervals
- maximum values of the intervals
- an interior point in each interval

Selection criteria for dual-use variables

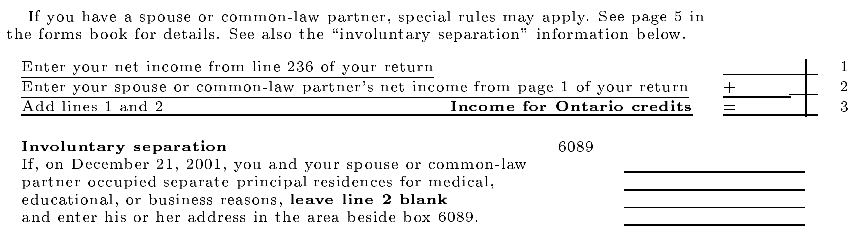
Sometimes a variable serves as an input to a program (or function) and holds the output from the program (or function) at the end of the desired computation. We call such a variable a *dual-use* variable. For such variables, we design additional test cases to meet the following selection criteria.

- Consider a test case such that the program produces a an output value which is *different* from the input value of the same dual-use variable.
- Consider a test case such that the program produces an output value which is *identical* to the input value of the same dual-use variable.

Selection criteria for multiple-type variables

Sometimes an input variable can take on values of *different types.* For axample, a variable may take on values of type *integer* in one program invocation and of type *string* in another invocation. It may be unlikely for a programmer to define a single storage space to hold values of different types. However, such multiple-type variables may arise in real-life systems, and programs must take necessary actions to handle them.

Example: In Figure 7.2, we show a part of the tax forms prepared by Canada Customs and Revenue Agency (CCRA). By analyzing this specification we conclude that a tax payer (user) inputs a *real number* or a *blank* in *Line 2.* The value input in *Line 2* is a real number representing the net income of the spouse or common-law



If you have a spouse or common-law partner, special rules may apply. See page 5 in the forms book for details. See also the "involuntary separation" information below.

Enter your net income from line 236 of your return		1
Enter your spouse or common-law partner's net income from page 1 of your return	+	2
Add lines 1 and 2 **Income for Ontario credits**	=	3

Involuntary separation 6089
If, on December 21, 2001, you and your spouse or common-law partner occupied separate principal residences for medical, educational, or business reasons, **leave line 2 blank** and enter his or her address in the area beside box 6089.

Figure 7.2: A part of Form ON479 of T1 General - 2001 published by Canada Customs and Revenue Agency.

partner of the user if both of them occupied the same residence on December 21, 2001. Otherwise, if they occupied separate principal residences for the specified reasons, then *Line 2* must be left blank and the addredd of the spouse or common-law partner must be provided in box 6089.

Clearly, *Line 2* is an input to a tax filing software system which must be able to handle different *types* of values input in *Line 2*, namely *real* values and *blank*–and a blank is not the same as `0.0`. This is because, if we equate a blank with `0.0` then the software system may not know when and how to interpret the information given in box 6089.

If an input or output variable can take on values of different types, then the following criterion is used.

- **Select test data by individually considering each type of values that the variable can take on.**

Referring to *Line 2* input of Figure 7.2, a tax filing program must be tested as follows.

- First, interpret *Line 2* as a *numeric* variable taking on values from an interval, and apply selection criteria such as selecting the minimum value, the maximum value, and an interior value of the defined interval.
- Second, interpret *Line 2* as a *string* variable taking on values from a discrete set of values, where the discrete set consists of just one member, namely a `blank`.

7.2.2 Arrays

An array has a more complex structure than an individual numeric variable. This is because of three distinct properties of an array, as follows.

Dimensions of the array: Just as we select extremal–both minimum and maximum–values and an intermediate value of a numeric variable, we need to consider arrays of different configurations, such as an array of minimum size, an array of maximum size, an array with a minimum value for the first dimension and a maximum value for the second dimension, and so on.

Values of individual elements of an array: This property is similar to that of a numeric variable. Individual array elements must be treated separately.

Collective interpretation of a substructure of an array: Just as we consider special values of a numeric variable, such as value `0`, `1`, and a small value ϵ, there exist special values of substructures of an array. For example, some well-known substructures of a 2-dimensional array are individual rows and columns, diagonal elements, lower triangular matrix, and upper triangular matrix. These substructures are interpreted as a whole.

Selection criteria for array dimensions

- The first step is to completely specify the dimensions of an array variable.
- The second step is to construct different, special configurations of the array by considering special values of individual array dimensions and their combinations. Assume that an array has k dimensions, and we choose the two endpoints and an interval interior for each dimension. These selections can be combined to form 3^k different sets of dimensions for a k-dimensional array.
- The third step is to apply the selection criteria of Section 7.2.1 to individual elements and substructures of a selected array configuration.

7.3 Combining input values to obtain a test vector

A *test vector*, also called test data, is an instance of the input to a program. Thus, a test vector is a certain configuration of the values of all the input variables. Values of individual input variables chosen in the preceedings sections must be combined to obtain a test vector. If a program has n input variables var_1, var_2, ..., var_n which can take on k_1, k_2, ..., k_n *special* values, respectively, then there are $k_1 \times k_2 \times \ldots \times k_n$ possible combinations of test data.

In Table 7.1, we show the number of special values of different input variables of the FSB module of Figure 7.1. Variable `MODE` takes on values from a discrete set of size 2. Thus, we consider both the values from the discrete set. Variables `CLOCK` and `MODE` take on values from one interval each. Thus, we consider *three* special values for each of them. From Table 7.1, one can generate $2 \times 3 \times 3 = 18$ test vectors.

Variable	Number of special values (k)	Special values
`MODE`	2	`{23, 79}`
`CLOCK`	3	`{0x0000000, 0x000FF00, 0xFFFFFFF}`
`ADDRESS`	3	`{0xFFFFF0000000, 0xFFFFF00FFF00, 0xFFFFFFFFFFFF}`

Table 7.1: Number of special values of inputs to the FBS module of Figure 7.1.

If a program has n input variables, each of which can take on k special values, then there are k^n possible combinations of test vectors. We know that k is a *small* number, but n may be a large number. Even with $k = 2$ and $n = 20$, we will have more than a million test vectors. Thus, there is a need to identify a method to reduce the number of test vectors obtained by considering all possible combinations of sets of special values of variables.

Howden propsed that input variables be put into the same subset if they are *functionally related.* It is difficult to give a formal definition of *functionally related*, but it is easy to identify them. For example,

- variables appearing in the same assignment statement are functionally related, and
- variables appearing in the same branch predicate (the condition part of an `if` statement, for example) are functionally related.

To reduce the total number of input combinations, Howden suggested that we produce all possible combinations of special values of variables falling in the same functionally related subset. In this way, the total number of combinations of special values of the input variables is reduced.

Referring to Figure 7.3(a), let program **P** have *five* input variables, such that `x1` through `x4` take on *three* special values, and `x5` is a Boolean variable. Thus, the total number of combinations of the special values of the four input variables is $3^4 \times 2 = 162$. Let us assume that program **P** has an internal structure as shown in Figure 7.3(b), where variables `x1` and `x2` are functionally related and variables `x3` and `x4` are functionally related. Function $f1$ uses the input variables `x1` and `x2`. Similarly, function $f1$ uses the input variables `x3` and `x4`. Input variable `x5` is used to decide whether the output of $f1$ or the output of $f2$ will be the output of **P**. Thus, we can consider $3^2 = 9$ different combination of `x1` and `x2` as input to $f1$, $3^2 = 9$ different combinations of `x3` and `x4` as input to $f2$, and two different values of `x5` to the decision box d in **P**. Thus, we need 36 ($= (9+9) \times 2$) combinations of the five input variables `x1` through `x5`, which is much smaller than 162.

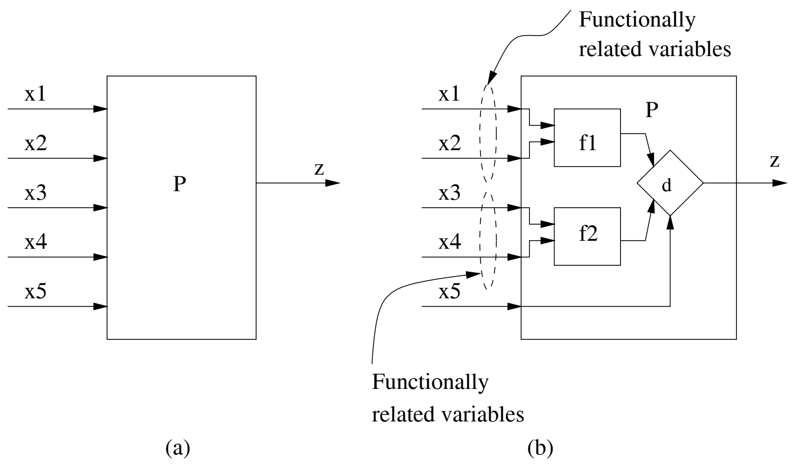

x1
x2
x3
x4
x5
P
z
(a)
Functionally related variables
x1
x2
x3
x4
x5
P
f1
f2
d
z
Functionally related variables
(b)

Figure 7.3: Functionally related variables.

`x` =	+`k`
`x` =	-`k`
`x` =	0

Table 7.2: A set of test data for program **P** in Figure 7.4.

7.4 Testing a function in context

Let us consider a program **P** and a function f in it as shown in Figure **??**. Variable `x` is an input to **P** and input to f as well. Suppose that `x` can take on values in the range $[-\infty, +\infty]$, and that f is called only when the predicate `x >= 20` holds. If we are unaware of the predicate `x >= 20`, then we are likely to select the set of test data shown in Table 7.2 to test **P**. Here, `k` is a number with a large magnitude.

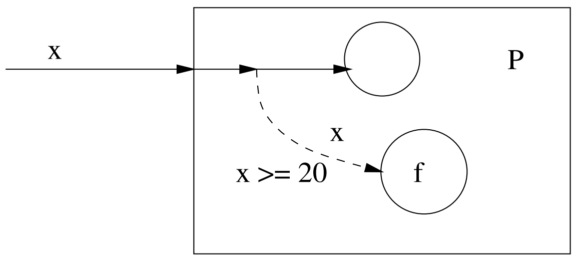


Figure 7.4: A function in context.

The reader may note that the function f will be invoked just once for `x = `+`k`, assuming that `k >= 20`, and it will not be invoked when **P** is run with the other two test data, because of the conditional execution of f. Also, testing the function f in isolation will require us to generate the same test data as above. And, it may be noted that the latter two data points are invalid data, because they fall outside the range of `x` for f in **P**. The valid range of `x` for f is `[20, `$+\infty$`]`, and functional testing *in context* requires us to select the values of `x` shown in Table 7.3. An illegal value of `x` for f is `20 - `ϵ.

7.5 Complexity of applying functional testing

In this chapter, we have explained the basic idea in functional testing. To revisit the topic, in functional testing, we perform the following steps.

- *Identify* the input and output domains of a program.

x =	k, where k >> 20.
x =	y, where 20 < y << k.
x =	20

Table 7.3: A set of data to test function f of program **P** in Figure 7.4 *in context.*

- For selected input values, *compute* the expected outcome as illustrated in Figure 7.5(a).
- For selected output values, *determine* the input values that will cause the program to produce those outputs as illustrated in Figure 7.5(b).

Generating test data by analyzing the *input domains* has the following two characteristics.

- The number of test cases obtained from an analysis of the input domains is likely to be *too many*, because of the need to design test vectors representing different combinations of special values of the input variables.
- Generation of the expected output for a certain test vector is relatively simple. This is because what a test designer have to do is to compute an expected output from an understanding and analysis of the specification of a system.

On the other hand, generating test data by analyzing the *output domains* has the following characteristics.

- The number of test cases obtained from an analysis of the output domains is likely to be *fewer* compared to the same number of input variables, because there is no need to consider different combinations of special values of the output variables.
- Generation of the input vector needed to produce a chosen output value will require one to analyze the specification in the *reverse* direction, as illustrated in Figure 7.5(b), and this will be a more complex task than computing an expected value in the forward direction, as illustrated in Figure 7.5(a).

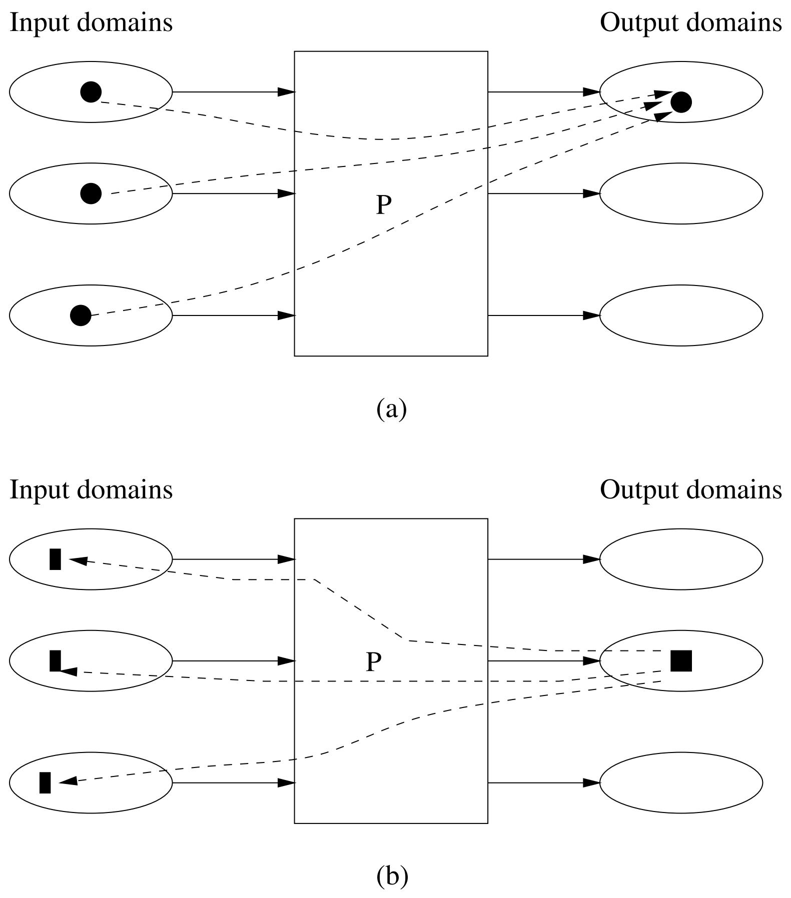


Figure 7.5: Obtaining output values from an input vector (a) and obtaining an input vector from an output value (b) in functional testing.

7.6 Functional testing in general

So far in this chapter we have discussed how to apply the idea of functional testing to a program–the entire program. However, the underlying concept–that is analyzing the input and output domains of a program–can as well be applied to individual modules, functions, or even lines of code of a program. This is because every computing element–irrespective of whether it is an entire program, an individual function of a program, or a line of code in a function–can be described in terms of its input and output domains, and hence the idea of functional testing can be applied to such a computing element.

Referring to Figure 7.6, program **P** is structured into three *modules* **M1**, **M2** and **M3**. Also, **M1** is composed of functions *f1* and *f5*, **M2** is composed of functions *f2* and *f3*, and **M3** is composed of functions *f4* and *f6*. We can apply the idea of functional testing to the entire program **P**, individual modules **M1**, **M2**, and **M3**, and individual functions *f1* through *f6* by considering their respective input and output domains as listed in Table 7.4.

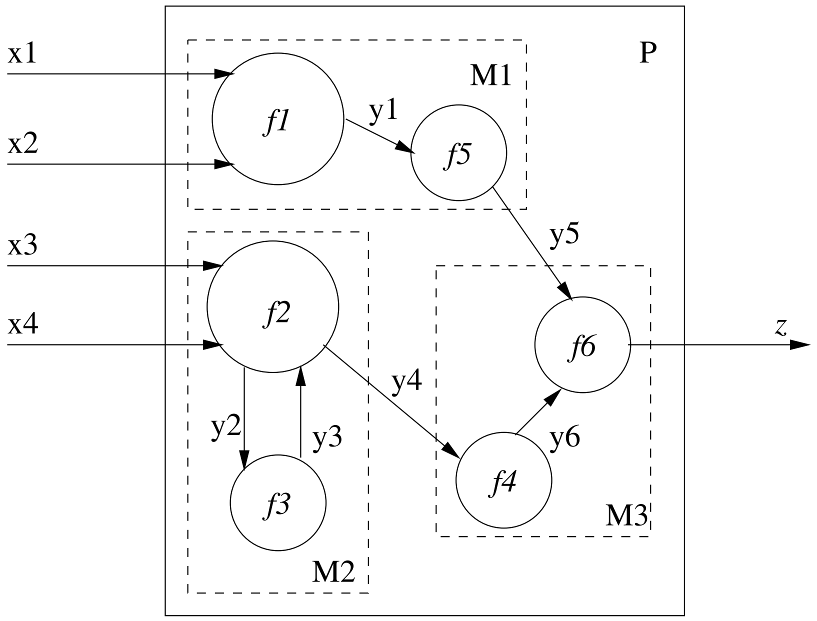


Figure 7.6: Functional testing in general.

Conceptually, one can apply functional testing at any level of abstraction–from a single line of code at the lowest level to the entire program at the highest level. However, as we consider individual modules, functions, and line of code, the task of accurately identifying the input and output domains of the computing element under consideration becomes more and more difficult.

Entity name	**Input variables**	**Output variables**
P	`{x1, x2, x3, x4}`	`{z}`
M1	`{x1, x2}`	`{y5}`
M2	`{x3, x4}`	`{y4}`
M3	`{y4, y5}`	`{z}`
f1	`{x1, x2}`	`{y1}`
f2	`{x3, x4, y3}`	`{y2, y4}`
f3	`{y2}`	`{y3}`
f4	`{y4}`	`{y6}`
f5	`{y1}`	`{y5}`
f6	`{y5}`	`{y6}`

Table 7.4: Input and output domains of functions of **P** in Figure 7.6

7.7 Exercises

`{To be included ...}`