3

September, 2008

testing experience

The Magazine for Professional Testers

Test Techniques in practice -

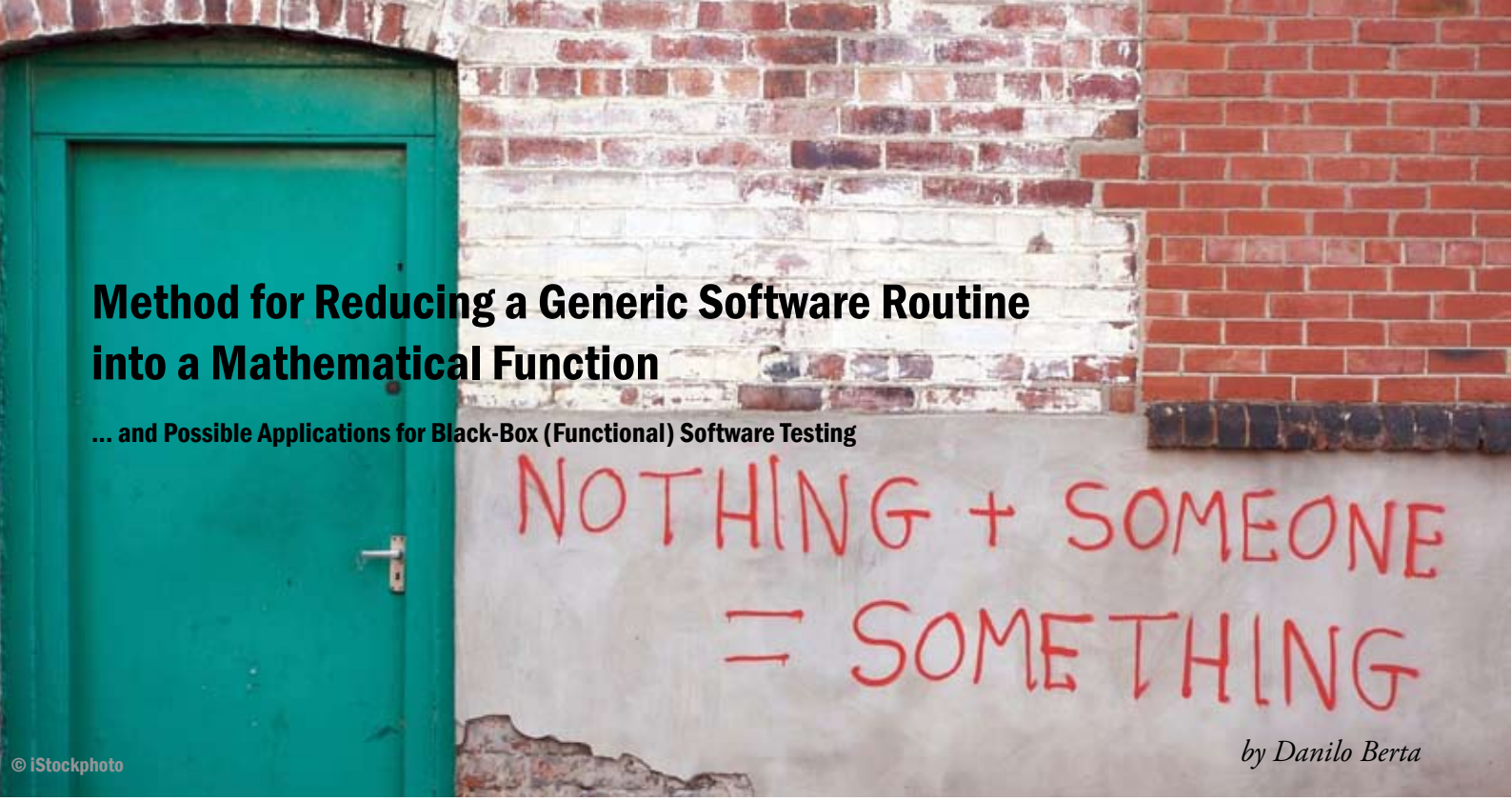
Do they help?
Why do we often test without them?

printed in Germany

free exemplar

www.testingexperience.com

ISSN 1866-5705



Method for Reducing a Generic Software Routine into a Mathematical Function

... and Possible Applications for Black-Box (Functional) Software Testing

by Danilo Berta

1. Abstracts

This article describes a method that can be used to reduce a general software routine, which accepts input of alphanumeric values and returns alphanumeric output values, into a mathematical function. A generic software routine takes and produces alphanumeric strings as inputs/outputs, whereby there is no way to define an order in these strings (e.g. it is not possible to determine between two strings which of them comes before or after the other).

In order to explain the matter in more detail, we will consider a simple software routine which produces as output the reverse string that it has taken as input. This routine can be defined as a mathematical function in this way $f(x_1x_2...x_n)=x_nx_{n-1}...x_1$, where $x_nx_{n-1}...x_1$ are generic alphanumeric characters. If we want to draw a Cartesian graph of the function, we need to know if, for example, the string $x_nx_{n-1}...x_1$ comes before or after the other string $x_nx_1...x_{n-5}$, or any other string which we can use as input of the function.

If we were able to reduce a software routine into a well-defined function, we would also be able to use the well-known mathematical methods to test the same function, sampling some of the input values of the function (in some statistical way) in order to verify whether the outputs of the same function for these input values are as expected. This method, which is strictly speaking a black-box method, can also be applied to any routine with any number of inputs and outputs; it's clear that in these cases we have to deal with multi-variable functions. In this article, we will not deal with statistical sampling methods, but will only describe a method for associating a generic string with a number in an unambiguous way, and will then discuss how to apply this method for the test and the regression test of generic software routines.

2. Introduction with a simple example

Let's start with a very simple example to so we can follow the course of the subsequent discussion with a more technical and mathematical approach.

To keep things simple, we assume that we have a really straightforward simple software routine that produces as output exactly the string it takes as input, whatever the string may be. This function will be:

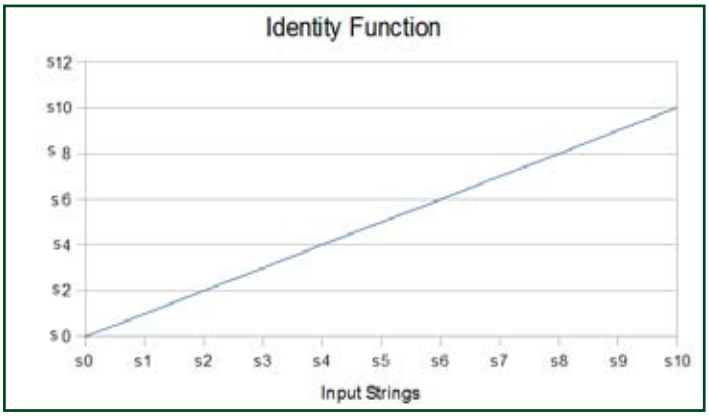


Fig 1: Identity Function Graph Example

(2.1) $f(x_1x_2...x_n)=x_nx_{n-1}...x_1$ where $x_1x_2...x_n$ are generic alphanumeric characters.

If we wanted to draw a Cartesian graph of this function, the first problem we would encounter is how to allocate an order to the input and output string, to be able to define the metrics (the intervals) to use for the X and Y-axes. Intuitively, we can imagine that this function would be the diagonal that splits the Ist quadrant in 2 parts of 45°, as shown in the picture below:

The intervals are labeled as s1, s2,...,s10, meaning that each point represents an input string and s(n)< s(n+1). However, we don't know, for example, whether the input string "ASER" is less than (or greater than) the string "ATER".

We need to define a method to associate a unique number to a string and a method to deduce the string that corresponds to that number.

The first thing that comes to mind is to number the single string elements progressively. If, for example, we suppose that the single elements that compose our input/output strings shall be the first three letters of the alphabet, 'A', 'B' and 'C', we can compile the following table:

Number	String	Number	String	Number	String	Number	String
1	A	13	AAA	25	BBA	37	CCA
2	B	14	AAB	26	BBB	38	CCB
3	C	15	AAC	27	BBC	39	CCC
4	AA	16	ABA	28	BCA	40	AAAA
5	AB	17	ABB	29	BCB	41	AAAB
6	AC	18	ABC	30	BCC	42	AAAC
7	BA	19	ACA	31	CAA	43	AABA
8	BB	20	ACB	32	CAB	44	AABB
9	BC	21	ACC	33	CAC	45	AABC
10	CA	22	BAA	34	CBA	46	AACA
11	CB	23	BAB	35	CBB	47	AACB
12	CC	24	BAC	36	CBC	48	...

Table 1: Association of string with numbers

The table should be read in this way: number 1 is linked to string "A", number 2 is linked to string "B",..., number 46 is linked to string "AACA" and so on.
Clearly, it's possible to go on indefinitely to compile the table above. We need to define a rule to calculate the corresponding number from the string.

The rule is as follows:

Start with the base association of string with number:

Number	String
1	A
2	B
3	C

Table 2: Association of Base String with numbers

As a hypothesis, we will suppose that all strings considered must come from any combination of the base letters {A,B,C} which in the following we will call the **dictionary**. So, string "ACABCCAACC" is valid, while string "ASDFGHAAABBCRR" is not valid, because it contains the elements S,D,F,G,R that are not in the defined dictionary.

The rule is expressed in the following formula, which for now we will describe by example (and we will explain later in detail).

(2.2)AABC = 1*33 + 1*32 + 2*31 + 3*30 = 1*27 + 1*9 + 2*3 +3*1 = 45

This is exactly the number that we find for string "AABC" in Table 1 above. If we examine the method of obtaining a value for string "AABC" it's evident that it's very similar to the way that numbers are usually transformed into base 2, 8 or 16 or any other base into decimal. So, there's nothing new or magic about it. The number "3" we have used in our formula is simply the base of the dictionary, or, in other words, the number of base elements in the dictionary {A,B,C} is exactly equal to three.

This method can be extended for dictionaries with any number of elements. The higher the number of elements in the dictionary, the greater will be the number we deduce for "short strings" (whereby "short strings" means strings with a low number of base elements).

It's a little trickier to find the "opposite rule", i.e. the rule to transform a generic number into a string composed only of elements of some well defined dictionary. For now, we introduce the rule and apply it with an example:

1. Divide the number for the base of the dictionary, considering both the **result** of the division and the **remainder** of the division. For example, for dictionary {A,B,C} the base is 3, as previously stated.
2. If the remainder is zero, subtract 1 from the result and replace the remainder number with the base.
3. If the result is greater than or equal to the base, continue with the division, applying rules (1) and (2) until the result is zero. In our example, this will happen just after we have found a result of the division that is less than 3 and after exactly 4 iterations.
4. Write down the string composed by the remainder of the division, starting from the last, and written sequentially.
5. Replace the number in the string, which was derived as described in point (4), with the letter in the base association string to number of the dictionary (see Table 2).

To explain this in more detail, we will use the number 45 as an example to verify whether the rule is correct. We will expect to find string "AABC", using the dictionary {A,B,C}.

Here is an example of the algorithm's application:

1. 45/3 = 15 remainder 0. Subtract 1 from 15: 15-1 = 14 that is greater than 3, so we have to continue. Replace the remainder 0 with 3.
2. 14/3 = 4 remainder 2. No need to subtract. 4 is greater than 3, continue.
3. 4/3 = 1 remainder 1. The result is less than 3. One division more and then we will stop.
4. 1/3 = 0 remainder 1. Stop here.

So, starting from the last remainder (step #4) and continuing with the other remainders , we have found the numeric string "1123", and using the table in Fig. 2.3 we can now replace the numbers with letters as follows:

$$1 \rightarrow A$$
$$2 \rightarrow B$$
$$3 \rightarrow C$$

In this way, the numeric string "1123" becomes "AABC", which is the string corresponding to the number 45 in the Table 1.

In the following paragraph, mathematical proof of these rules is presented.

3. Mathematical proof of the rules

First of all, we will start with some useful definitions, as used before:

1. A **dictionary** is a collection of **elements** a1,a2,...,an composing the string. It's written as follows: D{a1,a2,...,an}.
2. A **base** of the dictionary D is a number equal to the number of elements of the dictionary.
3. A string is **valid** with regard to dictionary D if it's composed exclusively by elements of D. It's **not valid** if there is at least one element that does not belong to dictionary D.
4. The **length** of a string is a number equal to the number of elements the string is composed of.
5. The **numbers equivalent to the dictionary's elements** (in the following NEDE) are defined by a matrix as follows:

NEDE	Element
$\alpha_1 = 1$	a_1
$\alpha_2 = 2$	a_2
$\alpha_3 = 3$	a_3
...	...
$\alpha_p = p$	a_p

Table 3: Association of Base String with numbers

It is presumed that all α where j=(1,2,...,L) are not zero. This is a fundamental hypothesis for what follows.
Please, note that all α_j for all j=(1,2,...,L) are strictly less than base B of the dictionary.
The first role does not have to be proven, but must be taken as a definition of the algorithm used to associate a number to a string. So, we can state the following:
Definition #1: Given a dictionary D, it's possible to associate an unique number N_0 to any valid string a1a2a3,…aL for this dictionary by applying the following rule:

(3.1) $N_0 =$

$$a_1 B^{L-1} + a_2 B^{L-2} + \ldots + a_L B^0 =$$

$$a_1 B^{L-1} + a_2 B^{L-2} + \ldots + a_L \quad \text{because } B^0 = 1$$

Where B is the dictionary's base and α_j is the numerical equivalent to the generic elements α_j of the dictionary, where j=(1,2,...,L) and L is the length of the string.
Starting from this definition, it's possible to deduce the rule for transforming a generic number into a string composed only by elements of the dictionary.

If we start dividing the number N_0 by base B, we will obtain:

(3.2)

$$\frac{N_0}{B} = a_1 B^{L-2} + a_2 B^{L-3} + ... + a_{L-1} + \frac{a_L}{B}$$

With the position:

(3.3)

$$N_1 = a_1 B^{L-2} + a_2 B^{L-3} + ... + a_{L-1}$$

We have:

(3.4) $$\frac{N_0}{B} = N_1 + \frac{a_L}{B}$$

that is equivalent to $N_0 = N_1 B + a_L$

Continuing in this way, we obtain:

$$N_0 = N_1 B + a_L$$
$$N_1 = N_2 B + a_{L-1}$$
$$N_2 = N_3 B + a_{L-2}$$

(3.5) .....

$$N_{L-2} = N_{L-1} B + a_2$$
$$N_{L-1} = a_1$$
$$N_L = 0$$

From the last equation in (3.5), i.e. $N_{L-1} = \alpha_1$ it should be evident that if we start from equation (3.1), we divide N_0 by $B^{(L-2)}$. For a better understanding, an example is given below where we apply (3.5) for the case where L = 4.

Start with: $N_0 = a_1 B^3 + a_2 B^2 + a_3 B + a_4$

Put: $N_1 = a_1 B^2 + a_2 B + a_3$

we have: $N_0 = N_1 B + a_4$

(3.5 ex)

Put: $N_2 = a_1 B + a_2$

we have: $N_1 = N_2 B + a_3$

Put: $N_3 = a_1$

we have: $N_2 = N_3 B + a_2$

Put: $N_4 = 0$

we have: $N_3 = N_4 B + a_1 \equiv a_1$

In the example L = 4, (L-1) = 3. The next N4 value is exactly equal to 0 (zero), because we divide N_3 (that is less than B) by B, obtaining 0 with remainder $N_3 = \alpha_1$. We have to stop the process at this point.
Remember that we have supposed that all α_j where j=(1,2,...,L) are not zero.

In formula (3.5), the α_j where j=(1,2,...,L) are the remainder of the subsequent division of number N_0 for the base of the dictionary, while the N_j where j=(1,2,...,L) are the results of the division.

From the above discussion we can deduce the following rule to extract from a number N_0 a string of length L composed only of elements from a given dictionary D{a_1,a_2,...,a_n}.

1. Define the dictionary and the number-equivalent elements.
2. Divide number N_0 for the base B of the dictionary, considering both the **result** of the division and the **remainder** of the division, and continue in this way until the result of the division is zero. This happens after L steps.
3. Starting from the last remainder create a string composed by the remainders of the divisions.
4. Consider the remainders of the divisions as the numbers equivalent to the dictionary's elements, and replace these numbers with the equivalent elements of the dictionary, using the "NEDE" table.

Now, all this works fine as long as we find only remainders of the divisions that are non zero. However, what happens if the remainders of the division are zero?

We will use the following simple tip. If the result of the division of X through Y is exactly N, with remainder 0 (zero), it can also be written as follows:

(3.6)

$$\frac{N}{Y} = N \text{ remainder } 0 \rightarrow \frac{X}{Y} = (N-1) \text{ remainder Y}$$

In other words, the result of $\frac{X}{Y}$ should be also be considered as equal to N-1 with remainder Y. For example: 15/5 = 3 remainder 0, but it's also 15/5 = 2 with remainder 5.

So, if we found some α_j equal to zero in the division process in (3.5), we now apply the tip and replace the result with (result -1) and the remainder with the base B of the dictionary.

This results in the following formula:

(3.7)

$N_{L-r} = N_{L-r} B + a_r$ with $a_r = 0$ means: $N_{L-r} = N_{L-r} B$

In other words:

(3.8) $$N_{L-r} = N_{L-r} B = (N_{L-r} - 1)B + B$$

This formula demonstrates the tip explained above.

4. Summary of the rules

In this paragraph, we will summarize the results obtained above and define the rules for transforming a string into a number and, vice versa, a number into a string, with reference to a given dictionary.

Rule #1: Transform a string into a number.

1. Define the dictionary and the elements of the dictionary. The elements in the string must only be composed of any combination of elements of the selected dictionary.
2. Define the NEDE (numerical equivalents to dictionary elements) matrix
3. Apply the following formula:

$$N_0 = a_1 B^{L-1} + a_2 B^{L-2} + ... + a_L$$

Where B is the dictionary's base and α_j is the numerical equivalent for the generic elements of the dictionary, where j=(1,2,...,L) and L is the length of the string.
N0 is the number corresponding to string "$a_1a_2...a_L$", that was searched.

Rule #2: Transform a number into a string.

1. Define the dictionary and the numerical equivalents.
2. Divide number N_0 for the base B of the dictionary, considering both the result of the division and the remainder of the division; in each subsequent division replace the dividend with the result of the previous division, and continue in this way until the result of the division is zero. This happens after L steps, where L is the length of the string.
3. If during execution of the steps described in point 2, you find a remainder equal to zero, replace the result with (result -1) and the remainder 0 with the base B of the dictionary.
4. Starting from the last remainder, create a string composed by the remainders of the divisions.
5. Consider the remainders of the divisions as the numerical equivalents to the dictionary's elements and replace these numbers with the equivalent elements of the dictionary, using the "NEDE" table.


We train in Latin America -
ISTQB Certified Tester Training
contact@bicorp.biz
© iStockphoto

The resulting string is the one that was searched and corresponds to number N_0.

5. Software implementation of the rules

I have prepared two Perl scripts that implement both processes described. Rule #2 described above is suitable to be implemented in a recurring way using Perl (or your favorite programming language), but I prefer to strictly follow the process described to be sure that it will work correctly.
All the scripts can be downloaded from link: http://www.bertadanilo.netsons.org/, or directly from: http://www.bertadanilo.netsons.org/Upload/perl_TestingExperience.zip.

Another Perl script was built with the purpose of generating the mathematical-equivalent function of some software routine. I took as an example the (very simple) routine that produces as output the reverse of the string it has taken as input.

All these scripts were prepared with the sole purpose of providing a "proof of concept". In the following, there is a brief explanation of how they work. All the scripts are released as is, under GPL License.

Script #1. This script, called `StringToNumber.pl` implements rule #1. It takes as input a string and a dictionary in the form of an array of chars separated by spaces, and returns the numerical equivalent to the input string, as per rule #1 explained above. To run the script, from DOS or *IX system, you need to run the following command:

```
perl StringToNumber.pl <input
string> <dictionary>
```

For example:

```
perl StringToNumber.pl ABECDEADD
"A B C D E"
```

which returns the number `637549`

You can avoid having to explicitly call the "`perl`" interpreter before the script name if you give the execution permission (`chmod +x StringToNumber.pl`) on *IX system to the script. In Windows, it depends on how your system is configured. I will not bother you any further with these details. A DOS script called `runStringToNumber.bat` could also be used to run the script (avoiding having to bore yourself with writing the dictionary, which is usually fixed), passing only the string input parameters, as follows:

```
runStringToNumber.bat  ABECDEADD
```

Script #2. This script, called `NumberToString.pl` implements rule #2. It takes as input a number and a dictionary in the form of an array of chars separated by spaces, and returns the numerical equivalent to the input string, as per rule #2 explained above. To run the script, from DOS or *IX system, you need to run the following command:

```
perl NumberToString.pl <input
number> <dictionary>
```

For example:

```
perl NumberToString.pl 637549 "A
B C D E"
```

which returns the string: `ABECDEADD`. The same considerations as for script #1 are valid for this one. A DOS script called `runNumberToString.bat` could also be used to run the script (avoiding having to bore yourself with writing the dictionary, which is usually fixed), passing only the string input parameters, as follows:

```
runNumberToString.bat  637549
```

Script #3. This script, called `EvaluateFunction.pl` implements both of the rules. It takes as input a dictionary in the form of an array of chars separated by spaces, and returns the table of the results of the function that should be used to plot the Cartesian diagram of that same function. The function is written in the code.
In detail, to use the script you will need to:

- Personalize the function to generate the data in order to plot in the main routine of the program. As a default, the function is the "string reverse" that produces as output the reversed input string. For example: ABC becomes CBA.
- Call the program as follows:

```
perl EvaluateFunction.pl <dic-
tionary> <number of point>
```

The output will be a table of data (separated by tabs) with <number of point> points with the x input string and y output string and equivalent numerical value of the software function.

For example, the output of command

```
perl EvaluateFunction.pl "A B C
D E" 10
```

will be a table of 10 values as follows:

X num	Y num	X num	Y num
1	1	A	A
2	2	B	B
3	3	C	C
4	4	D	D
5	5	E	E
6	6	AA	AA
7	11	AB	BA
8	16	AC	CA
9	21	AD	DA
10	26	AE	EA

Table 4: Example

6. The plot of the "string reverse" function

The first column is the X input numerical equivalent of the input strings in the third column, while the second column is the Y output numerical equivalent of the output strings in the fourth column (that is the real output of the reverse string function, or whatever function you decide to call).

These scripts were tested on Perl version v5.10.0 built for MSWin32-x86-multi-thread for the Windows Vista operating system, and Perl version v5.8.8 built for i486-linux-thread-multi on debian version (Kurumin). These are not really intended to control the formal validity of the inputs used (wrong numerical format, wrong dictionary format, and so on); therefore please take care to use the right format, otherwise the result cannot be guaranteed.

Now, at last, we will attempt to plot a Cartesian diagram of a software function that works with a list of symbols (strings). To keep the task simple, we have decided to consider the reverse string function. Using the command

```
perl EvaluateFunction.pl "A B C
D E" 100 > out.txt
```

we have in file `out.txt` a table of 100 points that can be used to plot the "reverse string" function. For this purpose, you may use your favorite spread sheet or any other software you usually use to draw graphs.

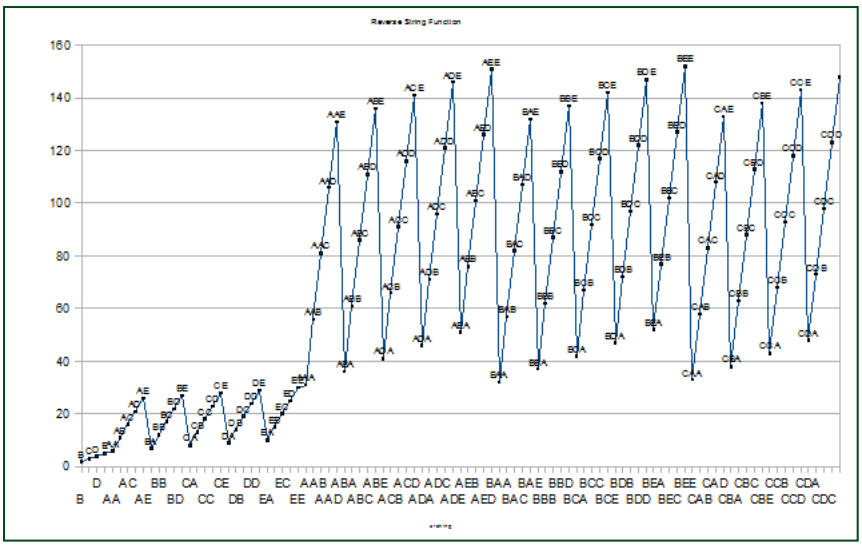
The plot looks like Graph 1.

The Y axis shows the numbers and not the Y-string value, to make the graph easier to manage. Considering that the function is simply a "reverse string" it is easy to imagine the corresponding string value of the Y interval number.
A graph with more points (in this case 500 points), but with fewer details, is shown below.

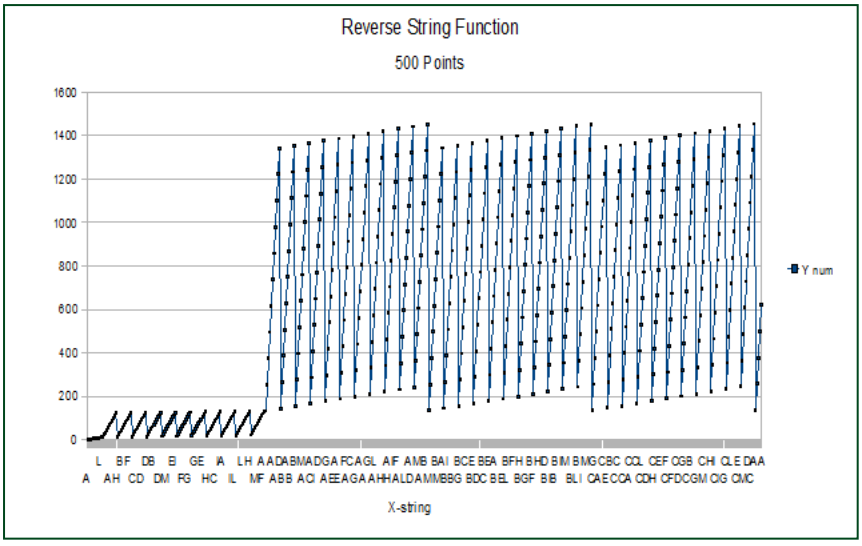
The graph with 10000 points looks like Graph 3.

7. Conclusions: What does all this have to do with software testing?

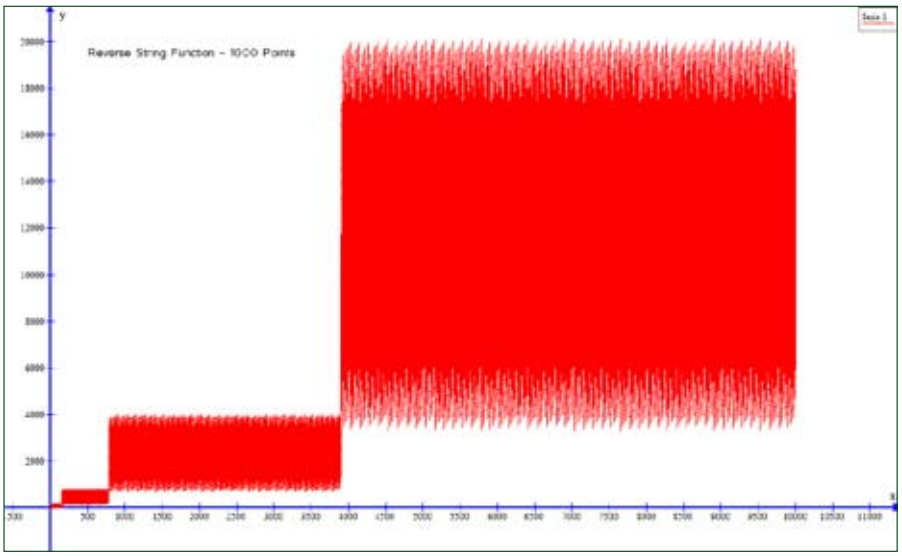
So, we have probably found a method to "reduce" a software routine into a mathematical function. Clearly, what we have described can "easily" be further generalized.
What does all this have to do with testing? At the current moment absolutely nothing; the main idea, as you can guess, is the following: if we know what the mathematical equivalent function behind the software routine under test is, we can easily "test it" under a (small or large) sample of input testing points (the "X" axis).
Now, the problem is that nobody (also, and in particular, no one who wrote the software) knows of the function. I think that nobody knows the profile of the "reverse string" function graphics presented above, or that the "reverse string function" is a well-known and



Graph 1: Shape



Graph 2: Shape



Graph 3: Shape

widely used feature in a lot of software programs.

So, one might ask what this method could be useful for? It could be useful in regression testing, in particular for the testing of single software components and for the test of (probably huge) amounts of database data, especially for systems that cannot be easily accessed, for example, systems who are located in different, non-integrated environments and for which access to the data for the software under test can only be produced only by extraction (e.g. in CSV format). In regression testing you can use the information collected during the "first test run" to clearly define the expected behavior of the function under test. You can therefore transform it into a numeric function (one or more functions) and keep it as reference for the other entire regression test or retest. The advantages are:

a. You collect input and expected output data in numerical format. Knowing the numerical input and output data, you can at any time recalculate the inputs and outputs needed for the software routine (see NumberToString.pl proof of the concept script). Clearly, you must also save the dictionary used in the transformation.

b. After you have run the software on the input data, you can transform the real output data into a table of numbers and draw the "real function". This function must then be compared with the "expected function" (i.e. performing a point-by-point comparison between the two functions). If any differences between the two are found, these can be analyzed. The operation can be easily done using any mathematical software that can draw graphics.

For example, to keep things simple, if you enter users onto a system and would like to verify that all the users' fields are correctly loaded onto the database, you can proceed as follows:

1. Prepare the file with the data to be loaded.
2. Transform all the columns for which you want to verify correct entry into the expected functions.
3. Load the data using the software under test
4. Extract the loaded data from the database and transform it into a function (i.e. a real function, which expresses the real software behavior); then compare the two functions.

This has been done for the first time. For the following test, you can automatize the process. Starting from the input and output numbers (which you saved in the first run), you proceed as follows:

a. Extract the input strings from the input numeric data and from the dictionary.
b. Run the software on the inputs and get

the outputs; transform the outputs into numbers and draw the function(s).

c. Compare the expected output (numbers) with the real output (numbers) and find the differences (if any).

Clearly, there will be no major differences if we compare this method with a more traditional "string comparator". With this method, you will probably not have to develop new "comparator software" to perform a particular comparison, because all you need is software that can draw graphics starting from a table of (x;y) values (or also for more variables).

Acknowledgements

I'm grateful to my friend and colleague Gianmaria Gai for the help he has given me when I wrote this article: he also suggested that I include some references. The problem is that there aren't any. The subject occurred to me one day when I was reading a (very boring) functional analysis for some software. I thought along the lines: "Wouldn't it be nice if we could reduce a big analysis into some mathematical functions! How we can do this? "...
If any reader knows about some other articles and/or books on similar methods, please let me know.

Biography

Berta Danilo graduated in Physics at Turin University (Italy) in 1993 and first started to work as a Laboratory Technician, but soon he switched to the software field as an employee of a consultancy company working for a large number of customers. Throughout his career, he worked in banks as Cobol developer and analyst, for the Italian treasury, the Italian railways (structural and functional software test), as well as for an Italian automotive company, the European Space Agency (creation and test of scientific data files for the SMART mission), telecommunication companies, and the national Italian television company. This involved work on different kinds of automation test projects, software analysis and development projects.
With his passion for software development and analysis – which is not just limited to testing – he has written some articles and a software course for Italian specialist magazines.
His work gives him the possibility to deal mainly with software testing; he holds both the ISEB/ISTQB Foundation Certificate in Software Testing and the ISEB Practitioner Certificate; he is a regular member of the British Computer Society.
Currently, he works as Test Manager for an Italian Editorial Company.