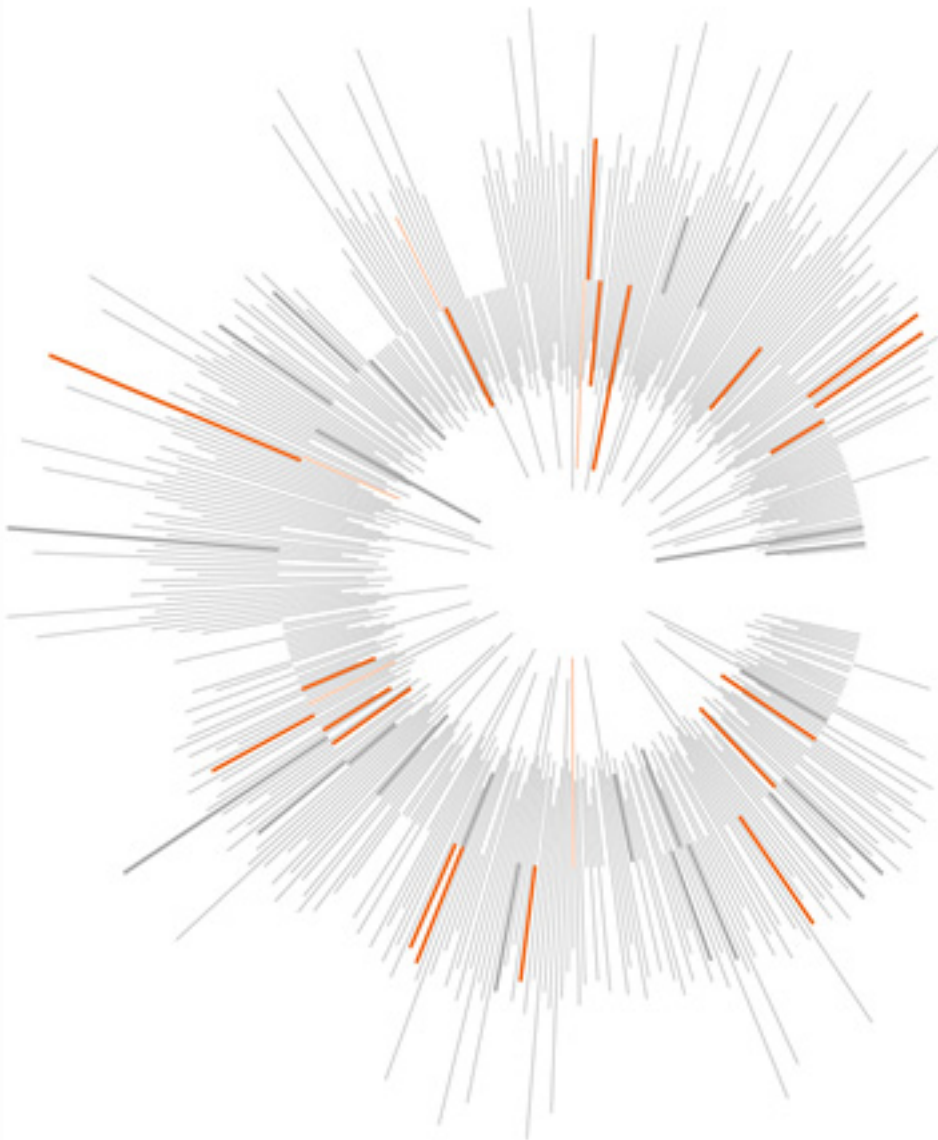


Practical approaches to improving your testing by maximising code coverage in complex database and SOA environments

By Huw Price



Contents

Introduction.....	4
Code or Functional Coverage?.....	5
White or Black Box Testing?.....	5
Trillions of combinations.....	6
Cause and effect.....	6
Requirements Parsing.....	7
Coverage worked example.....	8
The Code to be Tested.....	10
Building Test Input.....	11
Bender RBT.....	11
All Pairs.....	14
Jenny.....	15
All code combinations.....	15
Building the physical data.....	16
Creating the Data Using.....	16
the Application.....	16
Searching the data that matches your criteria.....	17
Hacking existing data.....	17
Creating data.....	18

Expected results.....	22
Code coverage results for overdraft protection.....	22
Summary.....	23
References.....	23
About Grid-Tools.....	23

Introduction

All testers, and some developers, grapple with the problems of testing as much of an application as possible with a minimum amount of effort. This white paper discusses some practical approaches to stressing as much code as possible while keeping the effort level manageable. I would like to thank Richard Bender for his input; inspiring me to try and tie together disparate methodologies into a practical end-to-end approach to increasing code coverage.

Most testers have experience of test automation tools such as QTP, Facilita, etc. and, to a greater or lesser degree, managing the data input and outputs from complex tests. Some testers have exposure to code coverage toolsets while all are familiar with the concepts. Many academics (indeed it seems to be the main research area of a large part of academia), work on code coverage tools and analytic techniques to improve software design (see <http://crest.dcs.kcl.ac.uk/> run by Dr Mark Harman of KCL). However, there are few companies that have an end-to-end integrated approach to maximizing their code coverage during testing.

Before continuing, it is worth noting a few practical issues:

- Most applications are built using disparate modern and legacy technologies for which code coverage tools do not always exist.
- Setting up code coverage can be time consuming and may not be worth the effort.
- Specifications and documentation are not always of a high standard.
- Testing is usually done at the end of a cycle and is not embedded into a development lifecycle.

This paper will cover some of the techniques and tools to build the most accurate input tests and also the practical methods and tools of creating the data to run these test. Simply building an optimized set of inputs is just the start of a solution. Testers need to be able to explode these tests out into real data in complex databases and SOA environments.

Code or Functional Coverage?

Before starting it is worth considering the differences between full code coverage and full functional coverage. There is a lively debate as to whether 100% code coverage is a) attainable and b) necessary. As an ex programmer, I would suggest that 100% code coverage is not really attainable and, in fact, the better the programmer the LESS likely is this to be attained. A simple example of this is an error trap: most code will look up data, for example, a Customer Credit Limit check in an authorization module and if the customer is not found an error will be raised. In a real suite of programs the chances of the customer record being deleted as you move from one program to another are extremely small and would cause much larger issues than the failure of one line of code. Testing this particular error trap, and there could be hundreds in one program, would be, in my opinion, a waste of time. Testing a standard error trap however, would be worthwhile, and forcing a failure of one or two missing data records to test the overall effect of missing data is worth the effort.

So should 100% code coverage be the goal? In my opinion, no, and research by Richard Bender suggests that 90% code coverage would be the maximum that can be expected. A more effective and realistic goal however is a 100% functional coverage, in other words, *are all of the specified requirements satisfied by the tests*. In this article I will refer to code coverage but the techniques are similar for both goals.

White or Black Box Testing?

Code coverage assumes you have access to the code and can see all the paths through the code. Black box testing reverses this, whereby you only have control over the input and the ability to monitor any output. Code coverage tends to be used in component testing, that is, where you have discrete sections of code with defined input and outputs. In a more typical test scenario, larger groups of components need to be tested as one unit. With the advent of complex SOA environments, complete end-to-end tests need to be designed and harder tasks implemented. In addition, testers need to balance the need for complete coverage with only limited time to test. The key challenge is to be more efficient and more effective.

From a practical point of view, having access to complete code gives you very useful information that can be used in designing tests. However, a balanced approach

needs to be used for both black and white box testing. From a tester's point of view, this is all 'just testing' with more or less information.

Trillions of combinations

Hardware engineering tends to have far greater success at eliminating failures than software. The reasons for this would be a digression if described in this article. Having stated this, however, one of the techniques very successfully used is optimizing the test inputs; a simple group of on/off switches that can be set in particular orders quickly grows to trillions of combinations. Hardware engineers have developed techniques to eliminate redundant tests, identify key relationships and make the very large numbers small. The application of some of these techniques to software means the amount of code exercised can be significantly increased whilst keeping the combinations of test inputs to a minimum.

Cause and effect

One very useful methodology, as outlined by Richard Bender, is to identify cause and effect actions. This is where combinations of input cause either explicit actions or intermediate 'nodes' to be set. The identification of these cause and effect graphs are an effective way of optimizing tests inputs.

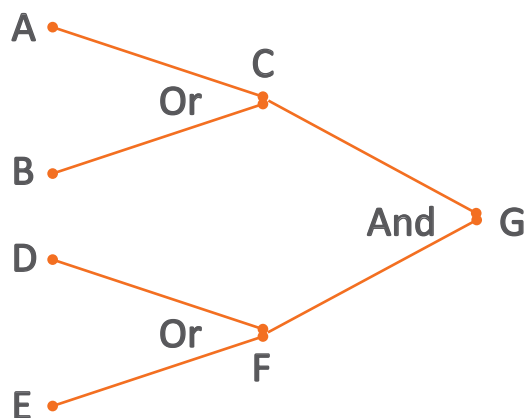


Figure 1- Cause Effect Diagram

In this case, nodes F and C can be turned on by A or B, or D or E respectively so that varying combinations of D and E, and A and B will have little effect on the result G. These nodes can be identified within specific programs or at a higher level within the application. For example, a customer may have a low credit score; setting this as a discrete node as part of the test input rather than having to set up multiple customers, some of whom have a low credit score, will reduce the test inputs for upstream testing.

The identification of what these cause and effect relationships are is based on clear, unambiguous specifications. A more subtle and difficult area for the tester is to identify which of these relationships are important, i.e. which ones to ignore. The tester will also have to decide which of the many data inputs need to be included as 'core' to the testing. Some data elements may not be that important and can be left out.

Requirements Parsing

While access to the code is a useful way of identifying these 'nodes', clear requirements are essential to building test input. Ensuring that specifications are clear and without ambiguity is essential. A simple AND or OR out of place or not clearly defined can cause confusion all the way along the development lifecycle. In the code coverage example below, a comma out of place could create an ambiguous definition.

Code coverage - overdraft protection example

If the customer is a business client or a preferred personal client and they have a checking account, \$100,000 or more in deposits, no overdraft protection and fewer than 5 overdrafts in the last 12 months, set up free overdraft protection. Else, do not give overdraft protection.

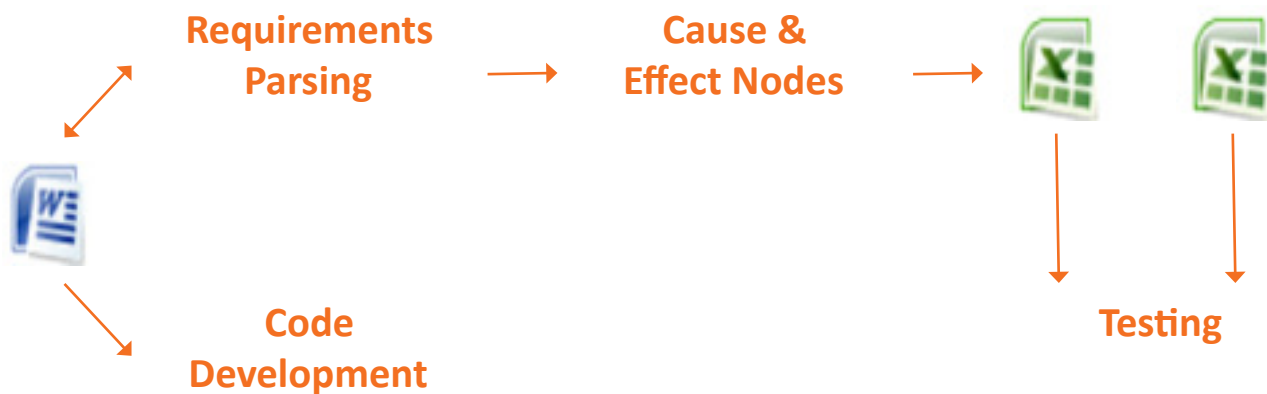
Coverage worked example

If a specification is not clear and is not validated prior to development, the following can happen:

Developer guesses correctly	Tester guesses correctly	Passes testing	Correct code in production
✓	✓	✓	✓
✓	✗	✗	✓
✗	✓	✗	✓
✗	✗	✓	✗

Figure 2- Ambiguous Specifications

A very useful technique to validate specifications is to use test parsing and analysis tools that can quickly identify ambiguous definitions early on in the software development life cycle.



The input from the clear text parsers can be validated and is a valuable resource as input to test case generators such as BenderRBT.

The clear text can be parsed and synthesized. For example, our overdraft example would be parsed to the following:

Automatic Check For Overdraft Protection
FUNCTIONAL VARIATIONS

[Synthesis of NEW tests specified.]

Functional Variations for:

Give_OD:-Checking AND Big_Money AND not current_OD AND Low_ODs AND
[Bus_Client OR Preferred].

[] - Refer to node(s): ****INT-1****

1. If checking and Big_Money and not current_OD and Low_ODS
and [Bus_Client
OR Preferred]
then Give_OD.

2. If not checking
(and Big_Money MASKed and not Current_OD MASKed and
Low_ODs
MASKed and [Bus_Client OR Preferred])
Then not Give_OD.

3. If not Big_Money
(and Checking and not Current_OD and Low_ODs and
[Bus_Client
OR Preferred])
then not Give_OD.

4. If Current_OD
(and Checking and Big_Money and Low_ODs and [Bus Client
OR Preferred])
then not Give_OD.

5. If not Low_ODs
(and Checking and Big_Money and not Current_OD and
[Bus_Client
OR Preferred])
then not Give_OD.

6. If not [Bus_Client
OR Preferred]
(and Checking and Bif_Money and not Current_OD and
Low_ODs)
then not Give_OD.

Functional Variations for:

****INT-1****:-Bus_Client OR Preferred.

7. If Bus_Client
(and not Preferred)
then ****INT-1****.

8. If preferred
(and not Bus_Client)
then not ****INT-1****.

9.If not Bus_Client and not Preferred
then not ****INT-1****.

There were NO Infeasible Variations
There were NO Untestable Variations

Legend:

[] = Relationship represented by ***INT-xx*** node.
(i.e, used for compound Relations statements).
() = Implicit node state(s) in functional variation.
INT-XX nodename = SoftTest-generated non-Observable
INTERmediate node

The Code to be Tested

Once the requirements have been validated, the developers can begin coding or more likely amending an existing system.

```

Begin
  select nvl(count(*),0)
  into wk_ocount
  from bank_account a,
       bank_account_overview v
  where ba_BC_id = wk_customer.bc_id and
        v.bo_ba_id = a.ba_id and
        v.bo_min_bal < 0;
exception
  when no_data_found then
    wk_ocount := 0;
end;

wk_int1 : 'N';
if wk_customer.bc_preffered = 'Y' or wk_business = 'Y'
then
  dbms_output.put_line('line 5');
  wk_int1 := 'Y' and
end if;
wk_int2 := 'N';
if wk_checking = 'Y' and
   wk_customer.ba_total_holdings >= 100000 and
   wk_overdrawn = 'N' then
  wk_ocount < 5 and
  wk_protect = 'N' then
  dbms_output.put_line('line 6');
  wk_int2 := 'Y'
end if;

if wk_int1 = 'Y' and wk_int2 = 'Y' then
  dbms_output.put_line('line 7');
  wk_protect:= 'Y';
else

```

Figure 4- PL/SQL code to implement overdraft protection

Building Test Input

Once the requirements have been clearly defined, the design of the test inputs can begin. There are numerous algorithms, tools and services that can help you with this task. I will use our overdraft example and compare four different methods:

- All code combinations
- All Pairs
- Jenny
- Bender RBT

A strong contender I did not include is the following web service: <http://www.testcover.com>. I would recommend testers to take a look at this site and to sign up for a trial, as they offer a service that is continually being updated with the latest academic research. Other practical techniques include:

- Equivalence Partitioning
- Boundary Value analysis

Bender RBT

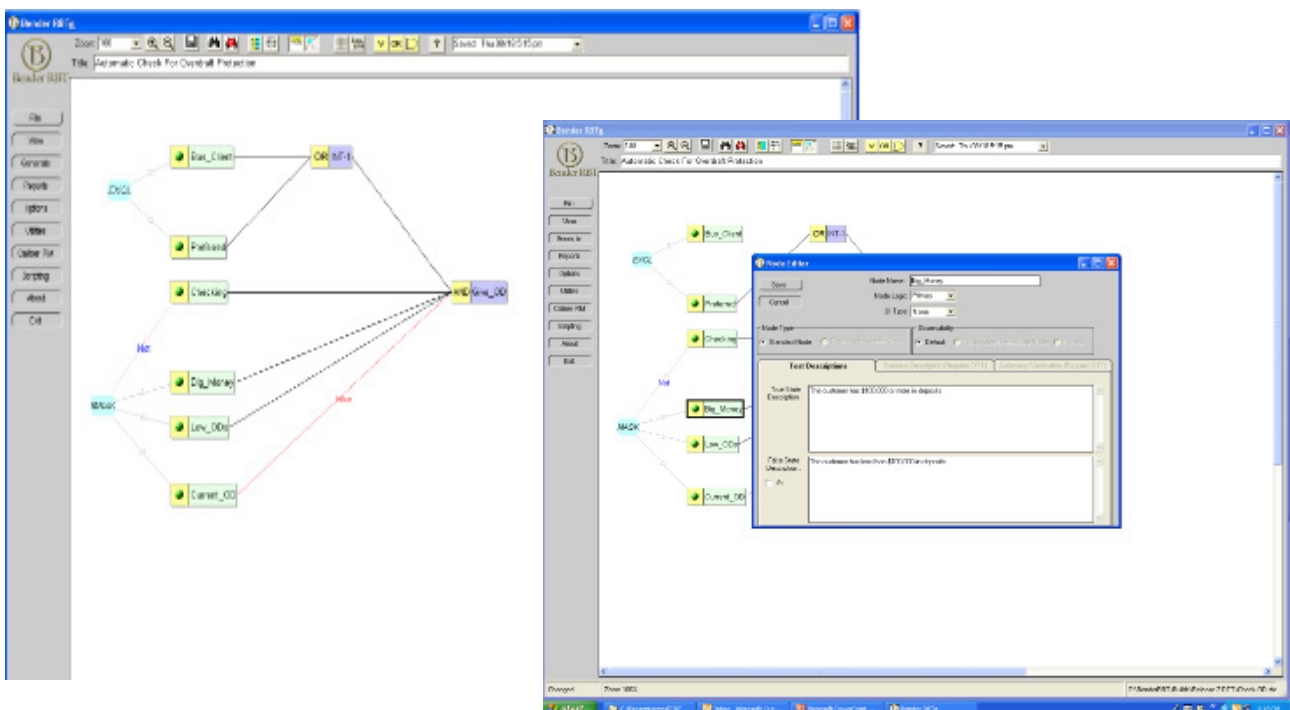


Figure 5- Bender RBT cause effect builder

As you can see in figure 5, the clear specification of AND and OR is easy to identify and verify. The identification of intermediate Nodes is crucial to the creation of smaller sets of test input.

The output of the RBT tool will create a set of tests that can be used as input to test the process.

		T	T	T	T	T	T
		E	E	E	E	E	E
		S	S	S	S	S	S
		T	T	T	T	T	T
		#	#	#	#	#	#
		1	2	3	4	5	6
Causes:							
Bus_Client		T	F	F	T	T	T
Preferred		F	T	F	F	F	F
Checking		T	F	T	T	T	T
Big_Money		T	M	T	F	T	T
Low_ODs		T	M	T	T	F	T
Current_OD		F	M	F	F	F	T
Effects:							
INT-1	<OBS>	T	T	F	T	T	T
Give_OD	{obs}	T	F	F	F	F	F

V A R I A T I O N	T	T	T	T	T	T
	E	E	E	E	E	E
	S	S	S	S	S	S
	T	T	T	T	T	T
	#	#	#	#	#	#
	1	2	3	4	5	6
1	X			X	X	X
2		#				
3			#			
4	#					
5			#			
6		#				
7				#		
8					#	
9						#
Unique Vars	1	2	2	1	1	1
Total Vars	2	2	2	2	2	2

Figure 6- RBT definition and coverage matrix

As you can also see from figure 6, tests take advantage of the INT-1 intermediate node to reduce the set of tests. In addition the production of an expected result will reduce the time required to check any results.

```

Test Statistics
Automatic Check For Overdraft P
Input Graph Filename:  C:\Program Files\Technology Builder
Input Last Modified:   4/27/2001  5:03:50 PM

Design Tests Last Run:  4/27/2001  5:03:50 PM
Caliber-RBT Release:    5.6(153)    / Win32

Run:  Synthesis of New Tests
Number of input statements:  16

Number of Functional Variations:  9
Number of infeasible variations:  0
Number of untestable variations:  0

Number of new test cases defined:  6
Number of tested variations:       9
Number of Feasible Variations:     9
Percentage of functional coverage of feasible variations:
    9/9*100 = 100%

Number of tested variations:       9
Percentage of functional coverage of testable variations:
    9/9*100 = 100%

Number of Primary Causes:  6
The THEORETICAL maximum number of test cases is:
    2^6 = 64

The number of test cases generated by Caliber-RBT is:  6
The test case compression ratio is:
    (2^6)/6 = 11 : 1

Number of Testable Variations:  9
The testable variations to test case compression ratio is:
    9/6 = 2 : 1

```

Figure 7- RBT Summary statistics

Reports of estimated coverage are invaluable in being able to gather code coverage statistics.

Once the test inputs have been defined they can be fed into tools such as Datamaker to generate the physical data; either as input to capture replay tools, directly into databases or as flat files which are fed into applications.

Exc1	Row	Bankbusiness	Bankchecking	Bankoprot	Bankotimes	Banksort	Banktot	Bankpreferred	Bankoverdrawn	Bankexpected
	1	Y	N	2	~BANKSORT~	50000	N	N	Y	
	2	N	N	2	~BANKSORT~	500	Y	N	N	
	3	N	Y	2	~BANKSORT~	500000	N	N	N	
	4	Y	Y	2	~BANKSORT~	500	N	N	N	
	5	Y	Y	6	~BANKSORT~	500000	N	N	N	
	6	Y	Y	2	~BANKSORT~	500000	N	Y	N	

Figure 8 – RBT test cases captured in Datamaker ready to generate the physical data

All Pairs

All Pairs or pairwise testing is a combinatorial testing method that, for each pair of input parameters, tests all possible discrete combinations of those parameters. Tools such as Datamaker will automatically generate these combinations for you.

Exc1	Row	Bankbusiness	Bankchecking	Bankoprot	Bankotimes	Banksort	Banktot	Bankpreferred	Bankoverdrawn	Bankexpected
	1	Y	N	2	~BANKSORT~	50000	N	N	Y	
	2	N	N	2	~BANKSORT~	500	Y	N	N	
	3	N	Y	2	~BANKSORT~	500000	N	N	N	
	4	Y	Y	2	~BANKSORT~	500	N	N	N	
	5	Y	Y	6	~BANKSORT~	500000	N	N	N	
	6	Y	Y	2	~BANKSORT~	500000	N	Y	N	

Figure 9 –All Pairs test cases captured in Datamaker ready to generate the physical data

Jenny

Jenny is a tool for generating regression tests. It will cover most of the interactions with far fewer test cases. It can guarantee pairwise testing of all features that can be used together and it can avoid those feature combinations that cannot.

Data in Bank System V1 Test Bed: Publish Control Variables Container: Jenny								
All 8 rows returned								
Exc1	Row	Bankbusiness	Bankchecking	Bankoprot	Bankotimes	Banktot	Bankpreferred	Bankoverdrawn
<input type="checkbox"/>	1	Y	N	N	2	500000	N	N
<input type="checkbox"/>	2	N	Y	Y	6	500	Y	Y
<input type="checkbox"/>	3	Y	Y	N	6	500	Y	N
<input type="checkbox"/>	4	N	N	Y	6	500000	N	Y

Figure 10 –Jenny cases captured in Datamaker ready to generate the physical data

All code combinations

All combinations of variables are defined and, in this case, this is manageable. If you add a few more variables, however, the number of combinations would grow rapidly.

Data in Bank System V1 Test Bed: Publish Control Variables Container: All Combinations								
95 rows returned. More rows to return!								
Exc1	Row	Bankbusiness	Bankchecking	Bankoprot	Bankotimes	Banktot	Bankpreferred	Bankoverdrawn
<input type="checkbox"/>	52	N	N	N	2	1000	Y	N
<input type="checkbox"/>	53	Y	Y	Y	2	1000	Y	N
<input type="checkbox"/>	54	N	Y	Y	2	1000	Y	N
<input type="checkbox"/>	55	Y	N	Y	2	1000	Y	N
<input type="checkbox"/>	56	N	N	Y	2	1000	Y	N
<input type="checkbox"/>	57	Y	Y	N	6	1000	Y	N
<input type="checkbox"/>	58	N	Y	N	6	1000	Y	N
<input type="checkbox"/>	59	Y	N	N	6	1000	Y	N
<input type="checkbox"/>	60	N	N	N	6	1000	Y	N
<input type="checkbox"/>	61	Y	Y	Y	6	1000	Y	N
<input type="checkbox"/>	62	N	Y	Y	6	1000	Y	N
<input type="checkbox"/>	63	Y	N	Y	6	1000	Y	N
<input type="checkbox"/>	64	N	N	Y	6	1000	Y	N
<input type="checkbox"/>	65	Y	Y	N	2	110000	N	Y
<input type="checkbox"/>	66	N	Y	N	2	110000	N	Y
<input type="checkbox"/>	67	Y	N	N	2	110000	N	Y
<input type="checkbox"/>	68	N	N	N	2	110000	N	Y
<input type="checkbox"/>	69	Y	Y	Y	2	110000	N	Y
<input type="checkbox"/>	70	N	Y	Y	2	110000	N	Y

Figure 11 –All code combinations generated and captured in Datamaker ready to

Building the physical data

Once you have decided on your test cases, you need to prepare the physical data to test your application. The creation of data to satisfy the test usually takes 3 to 5 times more effort than it takes to design the test cases themselves; populating data directly into databases via APIs or directly into disparate applications is hard work and must be factored in. In our case, we use Datamaker to populate the data directly into the database. In our example, the data is held in an Oracle database and is spread across multiple tables.

As the program we are testing is a batch program, the tester needs to build the data to test the code. The data must contain the combinations of attributes we have identified from our test analysis. These vary from 6 test cases for RBT to 128 for all combinations.

As a tester tasked with testing our code, you are immediately confronted with the problem of “how do I create these test cases?” The options you have are:

Creating the Data Using the Application

You could enter the data by hand for each test case, which takes time. Creating an account and entering transactions so that the account has been overdrawn over five times in the last year could be a very complex task. A similar level of complexity could exist for all variables. In addition, the tester would then have to understand how to use the application in areas they are not familiar with. In our example, the credit control testing team would have to run the banking batch transaction system to simulate overdrawn transactions.



Searching the data that matches your criteria

Many sites copy production data into development and testing environments, however, despite the potential data protection issues, this remains common practice.

Once you have existing data you can construct queries to search for the combinations of data that match your test cases. If you are lucky you will find them all very quickly. The problems with this technique are that you are, in effect, re-coding the code that the developer has created to track down the data. For complex queries, this may be difficult and the queries may be slow due to the size of the database. In addition, it is likely you will not find all of your test cases. By its very nature, increasing the code coverage takes you to areas that are very rare in existing data, as the majority of production data is very similar.

Hacking existing data

A very common technique is to find an account holder who is close to our first test case then go in and edit the data to match the first criteria. This process can then be repeated by; either running the batch program each time and checking the result for the account holder, or by tracking down similar account holders and directly editing different account holders one for each of our test cases.



The screenshot shows a database query result in Lucida Console. The query is for the table BANK_ACCOUNT_OVERVIEW (12). The result shows 12 rows of data. The columns are: Bo Id, Bo Date, Bo Ba Id, Bo Max Bal, Bo Min Bal, and Bo Transaction Activity. The data is as follows:

Bo Id	Bo Date	Bo Ba Id	Bo Max Bal	Bo Min Bal	Bo Transaction Activity
2	2008-08-19 00:00:00	1	6359.5	-2543.8	12719
3	2008-07-19 00:00:00	1	14950	5980	29900
5	2008-06-19 00:00:00	1	9930.5	3972.2	19661
8	2008-05-19 00:00:00	1	15066	6026.4	30132
10	2008-04-19 00:00:00	1	14007.5	5603	28015
12	2008-03-19 00:00:00	1	6028.5	2411.4	12057
13	2007-08-19 00:00:00	1	13689	5475.6	27378
14	2008-02-19 00:00:00	1	4530.5	1812.2	9061
16	2008-01-19 00:00:00	1	17208	6883.2	34416
18	2007-12-19 00:00:00	1	2264.5	905.8	4529
20	2007-11-19 00:00:00	1	13627.5	5451	27255
22	2007-10-19 00:00:00	1	4247.5	1699	8495

Figure 12- Customers have multiple accounts of different types with a summary of monthly activity.

- For each of the variables, change our data template (this is the original copied customer which has been turned into a generic data object) to affect the columns in the data that control our desired data effects. For example, if the variable `BANKOVERDRAWN` is set to `Y` place a '-' minus sign in front of the column `BA_CURRENT_BALANCE`.

The screenshot shows the Datamaker interface with a data table and a formula editor. The table has columns: Ba Account Type Bt Id, Ba Bc Id, Ba Setup Date, and Ba Current Balance. The formula editor shows the formula: `@if(~BANKOVERDRAWN~Y, -,)@@randrange(1000,10000)@`. The interface also shows a list of functions, columns, and variables.

Ba Account Type Bt Id	Ba Bc Id	Ba Setup Date	Ba Current Balance
@if(~BANKBUSINESS~Y, 1, 2)@	~PARENT(1)~	@addranddays(~CDATE~, -700, -400)@	@if(~BANKOVERDRAWN~Y, -,)@@randrange(1000,10000)@
@if(~BANKCHECKING~Y, 3, 2)@	~PARENT(1)~	@addranddays(~CDATE~, -700, -400)@	@if(~BANKOVERDRAWN~Y, -,)@@randrange(1000,10000)@

View / Edit data in column - BANK_ACCOUNT.BA_CURRENT_BALANCE

```
@if(~BANKOVERDRAWN~Y, -, )@@randrange(1000,10000)@
```

8794

Functions:

- add(number,number)
- adddays(date,days)
- addluhn(number)
- addmonths(date,months)
- addrand(number,min,max)
- addranddays(date,min,max)
- addseconds(datetime,seconds)

Columns:

- BA_ACCOUNT_NUMBER
- BA_ACCOUNT_TYPE_BT_ID
- BA_BC_ID
- BA_ID
- BA_SETUP_DATE
- BA_SORT_CODE
- Other Tables...

Variables:

- BANKBUSINESS
- BANKCHECKING
- BANKEXPECTED
- BANKOPROT
- BANKOTIMES
- BANKOVERDRAWN
- BANKPREFERRED
- BANKSORT

Figure 14 –In Datamaker the column `BA_CURRENT_BALANCE` will become negative if `BANKOVERDRAWN` is set to `Y`

- Once you are happy with the definitions, create a few test cases by hand and check that the results look correct in the application.

Publish Bank System V1 Test Bed: Customers Container: Customer to Data Target					
Current Date: 2008-09-19 Repeat: 1 times (1 - 100,000)					
Table Name	Seq	Rows in Container	Rows in Data Target	Comments	
✓ BANK_CUSTOMER	2	1	128		
✓ BANK_ACCOUNT	3	2	256		
✓ BANK_ACCOUNT_OVERVIEW	4	24	3072		
Variable	Value	Type	Description		
<input type="checkbox"/> BANKBUSINESS	Y	String	Has a Business Account		
<input type="checkbox"/> BANKCHECKING	Y	String	Has a checking account - Y or N		
<input type="checkbox"/> BANKEXPECTED	?	String	Expected Result		
<input type="checkbox"/> BANKOPROT	N	String	Overdraft Protection Y - N		
<input type="checkbox"/> BANKOTIMES	2	Number	Number of times overdrawn in the last 12 months		
<input type="checkbox"/> BANKOVERDRAWN	N	String	Overdrawn Y or N		
<input type="checkbox"/> BANKPREFERRED	Y	String	Preferred Customer Y or N		
<input type="checkbox"/> BANKSORT	089288	String	Bank Sort Code		
<input type="checkbox"/> BANKTOT	100000	Number	Minimum Total Holdings		

Figure 15 – A single test case data creation Datamaker the column `BA_CURRENT_BALANCE` will become negative if `BANKOVERDRAWN` is set to Y

- Use each test case input to drive each publish. In this case, we have cleared down the data in the schema as part of the publish.

Data in Bank System V1 Test Bed: Publish Control Variables Container: RBT										
All 6 rows returned										
Exc1	Row	Bankbusiness	Bankchecking	Bankoprot	Bankotimes	Banksort	Banktot	Bankpreferred	Bankoverdrawn	Bankexpected
<input type="checkbox"/>	1	Y	N	2	~BANKSORT~	500000	N	N	Y	
<input type="checkbox"/>	2	N	N	2	~BANKSORT~	500	Y	N	N	
<input type="checkbox"/>	3	N	Y	2	~BANKSORT~	500000	N	N	N	
<input type="checkbox"/>	4	Y	Y	2	~BANKSORT~	500	N	N	N	
<input type="checkbox"/>	5	Y	Y	6	~BANKSORT~	500000	N	N	N	
<input type="checkbox"/>	6	Y	Y	2	~BANKSORT~	500000	N	Y	N	

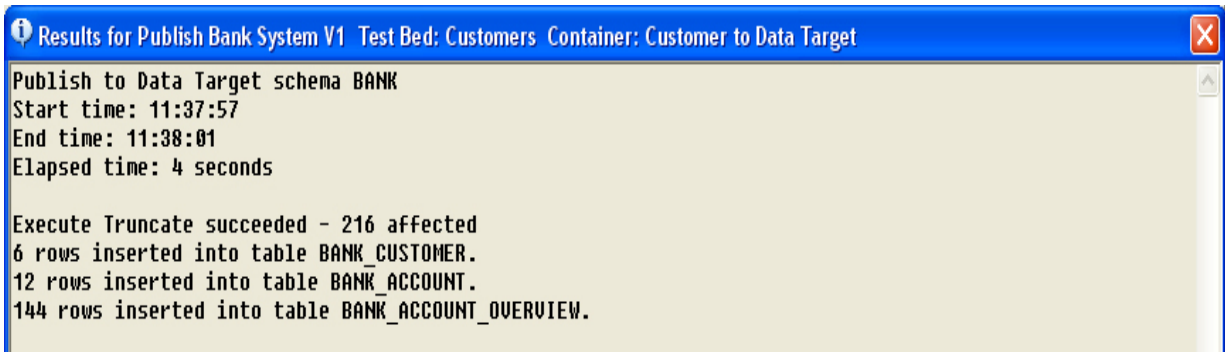


Figure 16 –The RBT test case physical data creation using Datamaker

A similar approach could be used when creating data into XML or flat file.

```

- <BANK_CUSTOMER_row>
  <value_st>Customers</value_st>
  <bc_id>000000000000006</bc_id>
  <bc_sex>MALE</bc_sex>
  <bc_first_name>Jonathan</bc_first_name>
  <bc_last_name>Ramachandran</bc_last_name>
  <bc_location>Herefordshire</bc_location>
  <bc_overdraft_protection>N</bc_overdraft_protection>
  <bc_setup_date>2007/12/26</bc_setup_date>
  <ba_total_holdings>517618</ba_total_holdings>
  <bc_prefered>N</bc_prefered>
  <bc_notes>EXPECTED=N Bus=Y Check=Y Pref=N Protection=N Overdrawn=Y LastYear=2 Total=500000</bc_notes>
</BANK_CUSTOMER_row>
</BANK_CUSTOMER>
- <BANK_ACCOUNT>
- <BANK_ACCOUNT_row>
  <value_st>Customers</value_st>
  <ba_id>000000000000001</ba_id>
  <ba_sort_code>089288</ba_sort_code>
  <ba_account_number>02000400</ba_account_number>
  <ba_account_type_bt_id>1</ba_account_type_bt_id>
  <ba_bc_id>000000000000001</ba_bc_id>
  <ba_setup_date>2007/02/04</ba_setup_date>
  <ba_current_balance>8988</ba_current_balance>
</BANK_ACCOUNT_row>

```

Figure 17 –XML format data created by Datamaker based on the RBT test case design

Expected results

With products such as RBT, it is possible to derive the expected results from the test conditions. The cause and effect analysis will be able to produce an expected result. With Datamaker, we have taken this result and included it in the data publishes. This allows the result to be easily checked against the actual result the program created. We used the column BC_NOTES to include a value EXPECTED=Y or EXPECTED=N. It is then easy to run a query to compare expected against actual.

If a column is not available in any of the tables, you can use other techniques such as updateable views, triggers, etc. to track and check these results.

With most test case design algorithms, the expected results are not generated. This means that the expected results will have to be calculated by hand; a very strong reason to try and reduce the number of test cases.

Code coverage results for overdraft protection

After generating the data directly into the database using Datamaker for each of the test case combinations, the code was run and code coverage statistics gathered.

	Number of test cases	Expected results	100% code coverage
All Pairs	8	x	x
Jenny	8	x	x
All combinations	128	x	✓
RBT	6	✓	✓

Interestingly, the RBT methodology and tool outperformed all other methods. This is not that surprising, as the use of the internal code 'nodes' allows the test cases to be significantly optimized. It would be possible to use triples or quadruple combinations of codes as part of the input rather than the paired used by Jenny and All Pairs, although this would increase the number of combinations.

Based on our practical observations of larger more realistic test scenarios, we would expect All Paired data inputs to result in about 25% to 50% coverage, which is not a bad

start. However, without adding in more sophisticated techniques, such as Equivalence Partitioning and cause and effect mapping, it will be very difficult to attain the goal of 100% coverage.

Summary

- Increasing code coverage is the route to improved testing
- Validate requirements for ambiguity, use text parsing tools to validate clear text
- Identify cause and effect relationships
- Propagate errors to an observable point
- Use advanced tools to generate minimum sets of test input combinations
- Do not use copies of production data in your testing. They are too large and will not contain the specific test cases you need for maximum code coverage
- Modify existing data is error prone, time consuming and is not repeatable
- Model existing data to generate and pivot values to create the perfect data for testing

References

Thanks to Richard Bender of Bender RBT Inc. for his input to this white paper. For more information on Bender RBT see www.BenderRBT.com. See also www.testcover.com, LLC for excellent information on test coverage as well as any article by Hans Schaefer who has written some excellent in-depth articles on test design.

About Grid-Tools

Grid-Tools is a UK company that has become the shining light for testing quality and efficiency in system development programs and other non-production environments. Using a combination of its Datamaker software product and in-house data creation and management expertise, Grid-Tools helps companies plan and execute testing and QA processes and data needs within projects by being adopted as part of the testing strategy and infrastructure. Grid-Tools achieves this through the effective provision of test case analysis and data using a range of algorithms, techniques and know-how

to deliver testing strategies and data that are fit for purpose in terms of timeliness, coverage, volume, consistency and logical coherence. Partners and end-user organizations alike have benefited from working with Grid-Tools by being able to plan testing strategies within system development programs that are effective and flexible and that deliver programs on time, on budget and with fewer defects - all as a result of the improved quality of testing processes, integration and the timely availability of high quality test data. Grid-Tools is also able to work with existing data, using its expertise and data management and creation software tools to provide datasets optimized for coverage, shape, size and regulatory compliance (HIPAA, PCI DSS and GLBA etc) for use in system testing and other non- or pre-production environments.

Grid-Tools is privately funded and is based in the UK with local sales and support staff based in the US and India. The company has over 100 customers spread across the banking, insurance, healthcare and government industries in Europe and the US. Grid-Tools works very closely with a small number of partners to ensure that customers receive the best quality and practices possible in delivering test strategies, data and management to their development and other non-production environments.

Grid Tools

11 Oasis Business Park
Eynsham
Oxfordshire
OX29 4TP

UK: +44 01865 884 600

US: +1 866 563 3120

E: info@grid-tools.com

www.grid-tools.com

Find us on Facebook

Follow us on Twitter

Join the Datamaker circle on LinkedIn

Subscribe to our blog

 **grid-tools**
DATA FIT FOR PURPOSE