



BenderRBT V8.0

Requirements-Based Testing

Bender RBT Inc.'s BenderRBT is a requirements-based, functional test case design system that drives clarification of application requirements and designs the minimum number of test cases for maximum functional coverage. By thoroughly evaluating application requirements for errors and logical inconsistencies, BenderRBT enables project teams to refine and validate the requirements earlier in the development cycle. The earlier in the cycle requirement errors are found and corrected, the less costly and time-consuming they are to fix. BenderRBT uses the requirements as a basis to design the minimum number of test cases needed for full functional coverage. BenderRBT then allows project teams to review both the requirements and the test cases in a variety of formats, including a logic diagram and structured English functional specification, to ensure that the requirements are correct, complete, fully understood and testable.

Most testing activities, and the tools that support them, can be divided into the following seven activities:

- ◆ **Define Test Completion Criteria**
- ◆ **Design Test Cases**
- ◆ **Build Test Cases – Including scripting and data provisioning**
- ◆ **Execute Tests**
- ◆ **Verify Test Results**
- ◆ **Verify Test Coverage**
- ◆ **Manage the Test Library**

BenderRBT addresses defining the test completion criteria, designing functional tests to meet the necessary criteria, verifying the test coverage, and assists in verifying test results and in maintaining the test library.

Test Activity	BenderRBT	Other Tools
Define Test Completion Criteria	BENDER 	
Design Test Cases	BENDER 	
Build Tests		Playback Tool / Data Base Utilities
Execute Tests		Playback Tool
Verify Test Results		Playback Tool / Data Base Utilities
Verify Test Coverage	BENDER 	
Manage Test Library		Test Manager

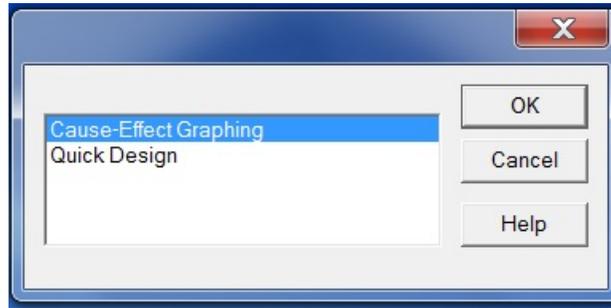
Feature/Benefit Table

Highly optimized algorithms	<ul style="list-style-type: none"> ◆ Minimizes the number of test cases needed to achieve maximum functional coverage ◆ Minimizes the time required to build, manage and maintain tests ◆ Enables 80-90% code coverage ◆ Identifies nodes where observability issues may mask errors
Automated test case generation	<ul style="list-style-type: none"> ◆ Ensures a consistently high level of coverage, independent of tester skill levels and experience ◆ Details the test cases in your choice of format ◆ Lists causes and associated effects for each test to allow easier test script development, review by stakeholders, approval and implementation
Quantitative test completion criteria	<ul style="list-style-type: none"> ◆ Allows management to track status of testing within and across projects in a consistent manner
Natural language test cases	<ul style="list-style-type: none"> ◆ Test cases are designed in a natural language, such as English, for easy review ◆ Test cases can be used by users to validate completeness and correctness of requirements

Target platform independent	<ul style="list-style-type: none"> ◆ Allows tests to be designed for any type of application running on any target platform in any language ◆ Ensures reusability and portability of test scripts ◆ Protects investment in team member skills
Coverage analyzer	<ul style="list-style-type: none"> ◆ Optimizes test planning ◆ Tracks test status throughout the test process
Flawed logic detection	<ul style="list-style-type: none"> ◆ Flags logical inconsistencies in requirements for faster correction
Capture/playback integration	<ul style="list-style-type: none"> ◆ Accelerates test script implementation in major capture/playback tools
Coverage matrix	<ul style="list-style-type: none"> ◆ Shows which functional variations are covered by each test case
Definition matrix	<ul style="list-style-type: none"> ◆ Summarizes input and output conditions for each test case for at-a-glance review
Functional specification generation	<ul style="list-style-type: none"> ◆ Provides an "as-built" specification for the application under test ◆ Ensures that the specification and test cases are in sync
Logic diagram	<ul style="list-style-type: none"> ◆ Enables project team to view the relationships between nodes graphically for better understanding
Integration with requirements management	<ul style="list-style-type: none"> ◆ Allows traceability between requirements and test cases ◆ Generates a Functional Specification from the Cause-Effect Graph model which can be exported as a rich text file (RTF) and brought into an RM tool or into a word processor
Integration with playback tools and test library managers	<ul style="list-style-type: none"> ◆ RBT supports TestIF – the OMG’s Test Integration Facility ◆ There direct exports to Quality Center and TOSCA ◆ The Test Definition Matrix and Functional Coverage Matrix can be exported as comma delimited files and brought into Excel ◆ All of the text based reports can be exported as RTF files and brought into a word processor.
Synergy with code coverage monitors	<ul style="list-style-type: none"> ◆ BenderRBT also has strong synergy with code coverage monitors. These tools keep track of which statements and branches have been executed by the tests. While important to measuring the thoroughness of the tests, monitors are underutilized in the industry. This is because they reveal that most test libraries rarely cover more the 40% of the code. Using BenderRBT the code coverage of tests designed prior to coding typical approaches 90%. Only a slight effort is then required to complete the code coverage.

Choice of Two Test Design Methods

BenderRBT comes with two distinct test case design engines. When you invoke RBT directly you will be given a choice of which you would like to use.



RBT Test Design Engine Options

Cause-Effect Graphing (C-E Graphing) takes you to the Graphing based test engine. Quick Design (QD) takes you to the Pairs-Wise based test engines. This includes Orthogonal Pairs and Optimized Pairs. C-E Graphing is intended for business critical, mission critical, and/or safety critical functions. It ensures that you not only got the right answer, but that you got the right answer for the right reason. It addresses the fact that multiple defects can sometimes cancel each other out. C-E Graphing ensures that defects are propagated to an observable point where testers can see the problem. QD is aimed at testing user interfaces (e.g., web pages, screens in client server applications). It is also applicable in designing configuration tests and quick shake-downs of even critical functions. Both C-E Graphing and QD address reducing the nearly infinite number of potential tests down to small, highly optimize test libraries. They both have full constraint rules support (One and Only One, Exclusive, Inclusive, Requires, and Masks) to ensure that the tests created are physically possible while still supporting full negative testing.

BenderRBT Cause-Effect Graph Based Test Design Engine

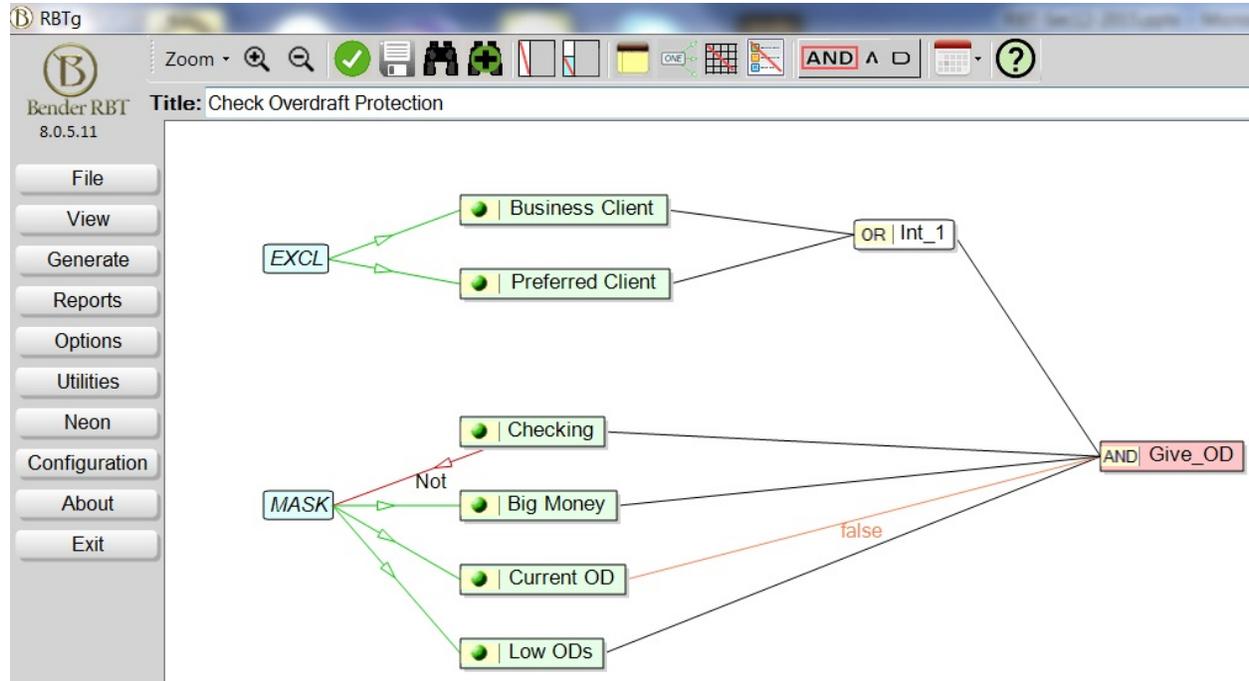
Better Requirements

Developing high-quality applications begins with the requirements. Requirements must be deterministic and unambiguous in order to ensure that the application is developed and tested accurately. RBT assists project teams in analyzing and reviewing the application requirements to eliminate logical inconsistencies and errors. Using cause-effect graphing, an innovative approach which graphically displays relationships and constraints between application nodes (inputs and outputs), the project team can analyze every aspect of the functional requirements in RBT. RBT then evaluates the recorded information to identify precedence problems in relations and logical errors. RBT provides detailed analysis information in a variety of easy-to-read formats. Analysts and project stakeholders collaboratively can review the natural language test cases generated by RBT, enabling them to identify and correct any requirement errors earlier in the development cycle.

Cause-Effect Graphing

A proven technique for effective requirements validation and test case design, cause-effect graphing is the process of transforming specifications into a graphic representation. This graphic representation depicts the functional relationships and conditions present in the requirements, illustrating how each input relates to every other input, as well as every output. Constraints and

observability of nodes also are established during this process, allowing the project team to identify potential problem areas. In developing the cause-effect graph, the test team evaluates the requirements for completeness, consistency, sufficient level of detail and lack of ambiguity, often finding defects that otherwise would not be found until integration testing.



BenderRBT's Graphic Front-End

The graphic front end to RBT allows project teams to quickly create cause-effect graphs, complete with node relationships, constraints, and attributes. When a node is created, users are prompted to enter the required attributes, reducing the risk of incompletely defined nodes. When the cause-effect graph is completed, RBT then designs the test cases based on the requirements depicted in the graph. RBT also uses the cause-effect graph to further evaluate the requirements for logical consistency. The project team can use the test cases generated by RBT to review requirements with stakeholders, or they can use the structured English requirements document automatically generated by RBT. The more readable the requirements are, the more likely the project team is to develop the right application.

TEST#1 -- Check Overdraft Protection

Cause states:

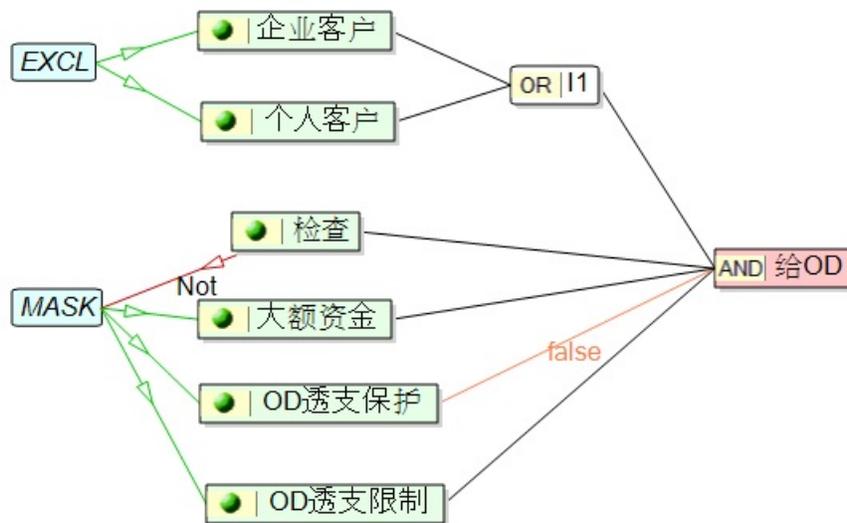
- The customer is a Personal Preferred Client
- The customer has a checking account
- The customer has \$100,000 or more in their checking account
- The customer does not have overdraft protection on the checking account
- The customer has had less than six overdrafts in the last 12 months

Effect states:

- Give the customer free overdraft protection

BenderRBT's Script Test Definitions Report details every step of the test cases designed, including the input conditions and the expected results (or effects) of each step.
Localization Support

All of the user entered information – Graph Title, Notes, Node Names, Node Descriptions – can be entered in any language. RBT will then generate the all of its output using this information. Here is the above graph built using Chinese:



Here is an example of a test generated:

TEST#1 -- 检查透支保护

Cause states:

- 客户不是企业账户
- 客户是个人账户
- 客户有一个支票账户
- 客户的支票账户超过10W美元以上
- 客户的支票账户没有透支保护
- 客户在过去的12个月里透支不足5次

Effect states:

- 给客户免费透支保护

Minimum Tests

In many testing environments, tests are developed using “gut feel” or combinatorics-based methods. Gut feel testing relies on individual testers to develop the tests to be used, while combinatorics-based testing uses all possible combinations of the inputs. While these test development methods are widely used, they do not ensure full functional coverage, let alone guarantee the minimum number of required tests. BenderRBT uses a mathematically rigorous algorithm to determine the minimum number of test cases required for full functional test coverage.

For instance, in an application with 37 inputs, an exhaustive combinatorics-based approach will result in over 130 billion possible test cases. A gut feel testing approach might reduce this number to 50 or 100 tests, but there is no way to know whether they are the right tests for the application. Because the skill level and experience of the individual testers may vary, there is no way to guarantee a high level of functional coverage. In this example, RBT reduces the possible number of test cases to only 22 in a one second. Since these tests are based on the actual, documented requirements, the test team will be testing 100% of the application’s functionality. This minimum set of tests cases also significantly decreases the amount of time required to design and build tests, reducing the overall testing effort.

In every comparison study our clients have done over the years, RBT has reduced the number of necessary tests by a minimum 4X for equivalent coverage. For groups just using “gut feel” testing it has been closer to a 10X reduction.

Maximum Coverage

Using a gut feel test design approach, the test team can not be sure that their tests cover 100% of the application’s functionality. In fact, studies have shown that in gut feel testing environments, the tests only cover an average of 30-40% of the application’s functionality. RBT’s proven automated test case design approach ensures that the functional test coverage will achieve 100%,

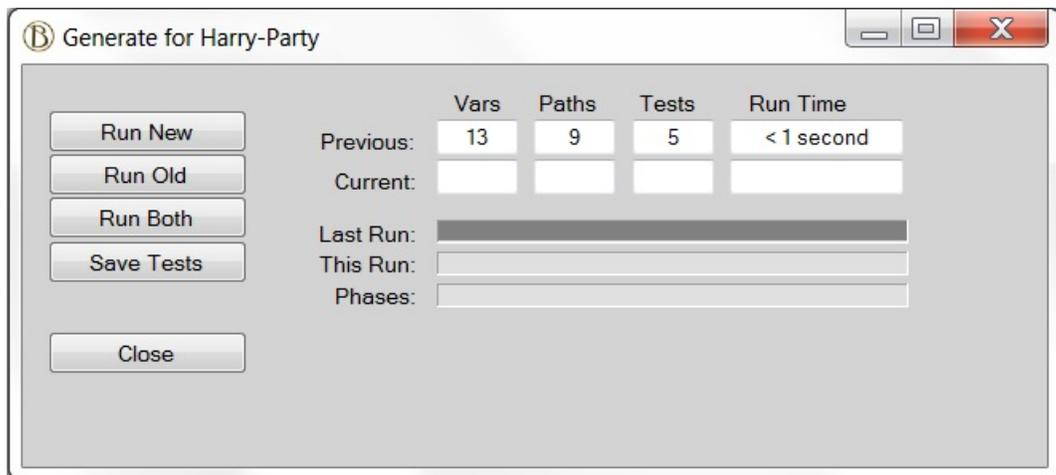
with the minimum number of tests. RBT carefully evaluates all of the cause and effect information it is given to reduce the possible number of test cases to a minimum set that is functionally complete. RBT also cross-references the functions with the test cases. When evaluated with the status of executed tests, this information allows the project team to calculate the percentage of functionality running correctly. Management then can make an informed decision about whether the application is ready for production.

Protecting Your Investment In Test Cases

The Cause-Effect Graphing process is an iterative one. You generally graph, review the results, and tune the graph until you are sure the requirements are solid and that the graph reflects those requirements. You then implement the test cases. When you commit to building the executable tests you want to ensure that RBT knows that this set of tests is the one you are implementing. This will allow you to protect your investment in these tests.

If RBT is aware of existing tests, it can evaluate those tests as the requirements and graph change. How much coverage do the old tests give you? What new tests will you need? What modifications have to be made to the old tests? RBT can answer those questions for you.

Therefore, RBT gives you a number of options in generating test cases.



Test Generation Options

The **Run New** option will design a new set of tests based on the graph you have just entered.

The **Run Old** option will evaluate the coverage of a set of existing tests against the current version of the graph.

The **Run Both** option will evaluate the coverage of a set of existing tests and then supplement these tests to complete the coverage of the graph.

Note: This feature can be used to factor in test cases that were not designed by RBT. There is a dialog for allowing the user to tell RBT about existing test cases, regardless of their source.

Matrix Views

When planning the testing phase, it is important to understand the functional coverage of each test case, as well as the state of each node in each test case. RBT provides two matrix views that show this information in detail. The Coverage Matrix shows which functional variations are covered by each test. It also illustrates that every test exercises at least one functional variation not covered by any other test. Using this matrix, the test team can be sure that they are testing 100% of the application’s functionality. RBT’s Definition Matrix summarizes the input and output conditions included in each of the test cases generated by RBT. Both of these matrixes may be exported to Excel for further annotation by the tester.

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

BenderRBT’s Functional Coverage Matrix identifies which functional variations are in which test cases. An “X” means that the variation is in two or more tests. A “#” means the variation is only in one test.

V A R I A T I O N	T	T	T	T	T	T	T
	E	E	E	E	E	E	E
	S	S	S	S	S	S	S
	T	T	T	T	T	T	T
	#	#	#	#	#	#	#
	1	2	3	4	5	6	7
	New/old						
	1		#				
2	X		X	X	X	X	
3	Infeasible						
4						#	
5	X	X	X	X	X		
6						#	
7				#			
8	X	X	X			X	
9					#		
10	#						
11		X		X		X	
12			#				

Coverage Analysis

Weak Coverage:
11 / 19 * 100 = 57% Clear All

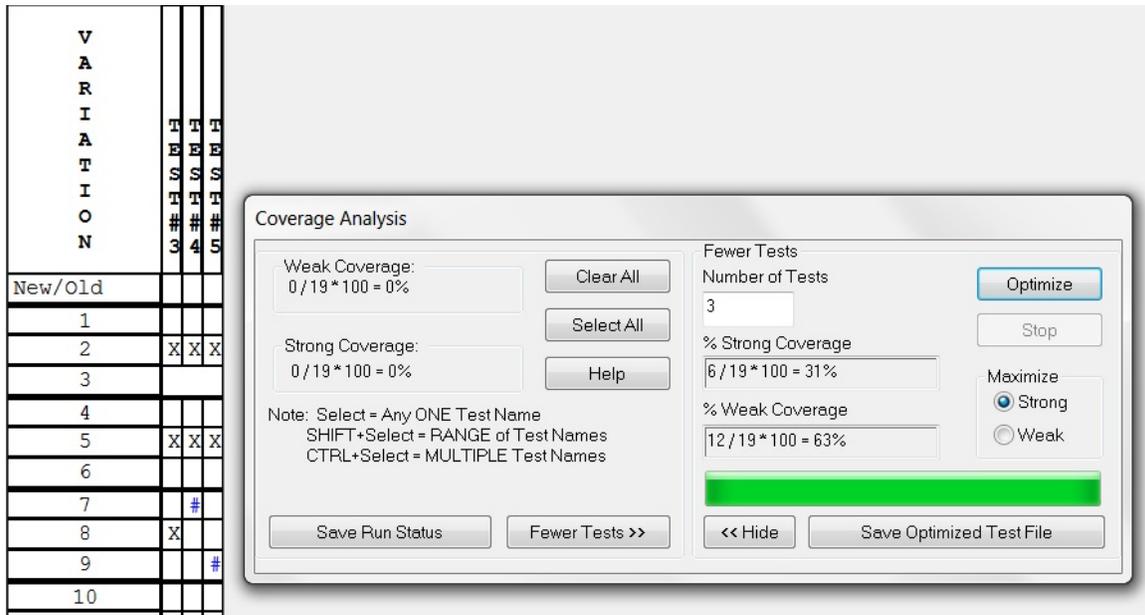
Strong Coverage:
2 / 19 * 100 = 10% Select All

Help

Note: Select = Any ONE Test Name
SHIFT+Select = RANGE of Test Names
CTRL+Select = MULTIPLE Test Names

Save Run Status Fewer Tests >>

BenderRBT's Coverage Analysis Matrix allows the project team to quantifiably determine the status of testing. When one or more test cases are selected, the Coverage Analysis function calculates the selected test cases' percentage of weak and strong functional coverage.



Fewer Tests Dialog

This feature allows you to enter in a number less than or equal to the number of total tests and have RBT determine which is the optimal subset of tests – i.e. which tests would give you the greatest possible coverage.

	U I T Y P e	T # 1	T # 2	T # 3	T # 4	T # 5	T # 6	T # 7
New/Old								
Causes:								
Bus-Client		F	T	F	t	t	t	t
Preferred-Client		T	F	F	f	f	f	f
Checking		T	T	T	F	T	T	T
Big-Money		T	T	M	F	T	T	T
OD-Protection		F	F	F	M	F	T	F
Few-ODs		T	T	M	T	T	T	F
Effects:								
I1		T	T	F	T	T	T	T
Give-OD {obs}		T	T	F	F	F	F	F

BenderRBT’s Definition Matrix uses a table format to display the state of each node in each test case, allowing at-a-glance understanding of each test case.

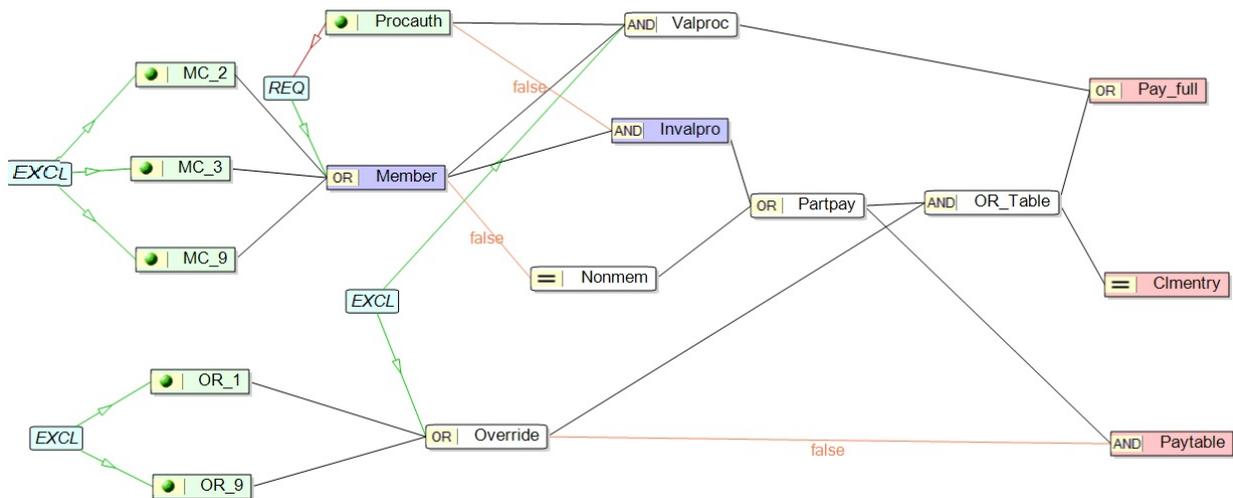
Strong Support For Agile

Agile projects are highly iterative within and across releases. Common problems on agile projects are that tests are often a sprint behind and specifications are never fully documented. In addition to the ability to protect the investment in tests implemented from prior versions of the graphs, RBT can generate a Functional Specification from the models.

This user story from a dental insurance application:

“Determine the amount to be paid for each dental insurance claim.”

Resulted in this Cause-Effect Graph:



Which in turn generated tests such as:

TEST 1

<p>Cause States:</p> <ul style="list-style-type: none"> The Dentist is a Member Dentist The procedure was not preauthorized An override code was entered <p>Effect States:</p> <ul style="list-style-type: none"> Override the partial payment Make an entry on the paid claims report

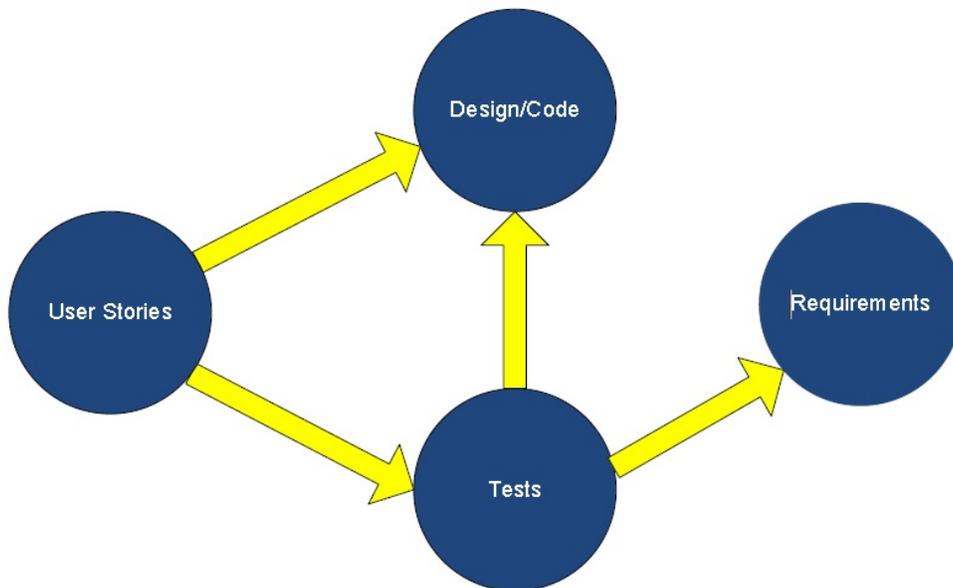
TEST 2

<p>Cause States:</p> <ul style="list-style-type: none"> The Dentist is a Member Dentist The procedure was preauthorized <p>Effect States:</p> <ul style="list-style-type: none"> Pay the full amount of the claim Do not make an entry on the paid claims report
--

And generated this Functional Specification:

1. IF The member is a full member
OR The member is an associate member
OR The member is a temporary member
THEN This is a member dentist
ELSE This is a non-member dentist.
2. IF This is a member dentist
AND The procedure was preauthorized
THEN This is a valid procedure for the member dentist.
3. IF This is a member dentist
AND The procedure was not preauthorized
THEN This is not a valid procedure for the member dentist.
4. IF [This is a non-member dentist]
OR This is not a valid procedure for the member dentist
THEN This is a potential partial payment situation.
6. IF This is a potential partial payment situation
AND The override code was accepted
THEN Override the partial payment.
7. IF This is a valid procedure for the member dentist
OR Override the partial payment
THEN Pay the full amount of the claim.
8. IF This is a potential partial payment situation
AND No override code was entered
THEN Make a partial payment of the claim based on the system table.
9. IF Override the partial payment
THEN Make an entry on the paid claims report
ELSE Do not make an entry on the paid claims report.

This ensures that the code, the tests, and the specifications are all provably in sync at the time of the release.



Quick Design – Pair-Wise Based Test Design Engines

Quick Design has multiple test case design engines, all based on Pair-Wise testing. One is used for Orthogonal Pairs – create a balanced set of tests with pairs in equal numbers of tests to the extent possible. This is used for designing tests for configuration testing and for creating seed tests for performance testing. The two other engines are for Optimized Pairs testing – cover the set of pairs with the minimal number of tests.

Quick Design allows you to design tests in just minutes. You just identify each test input Variable. For each Variable you define the States you want to test.

Add Variable

Variable Name: Customer

True Description: The customer is a

False Description: /b

It is OK for all states to be false

Defining a Variable in Quick Design

QDesign - [QD1.bqd*]

File Edit View Inputs Run Reports Options Utilities Window Help

Field Name	Value
Variable Name	Customer
True Description	The customer is a
False Description	/b
All false is OK	0

Add State

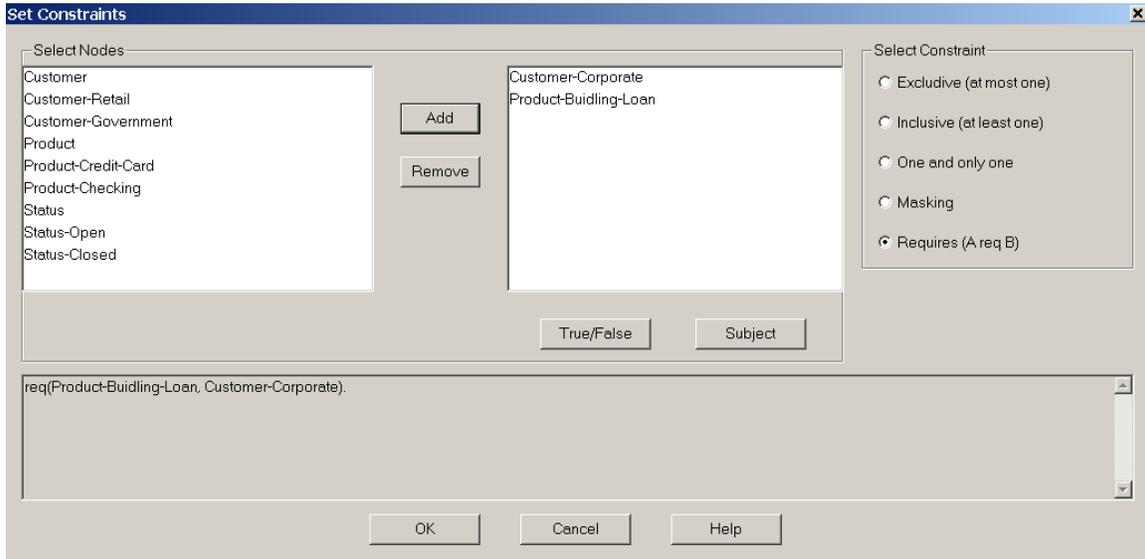
Corporate

Description: Corporate customer

Defining a State in Quick Design

QD concatenates the Variable description with the State description in the generated test scripts. This saves typing and ensures consistent wording of test scripts. In the above example the final description would read “The customer is a Corporate customer”.

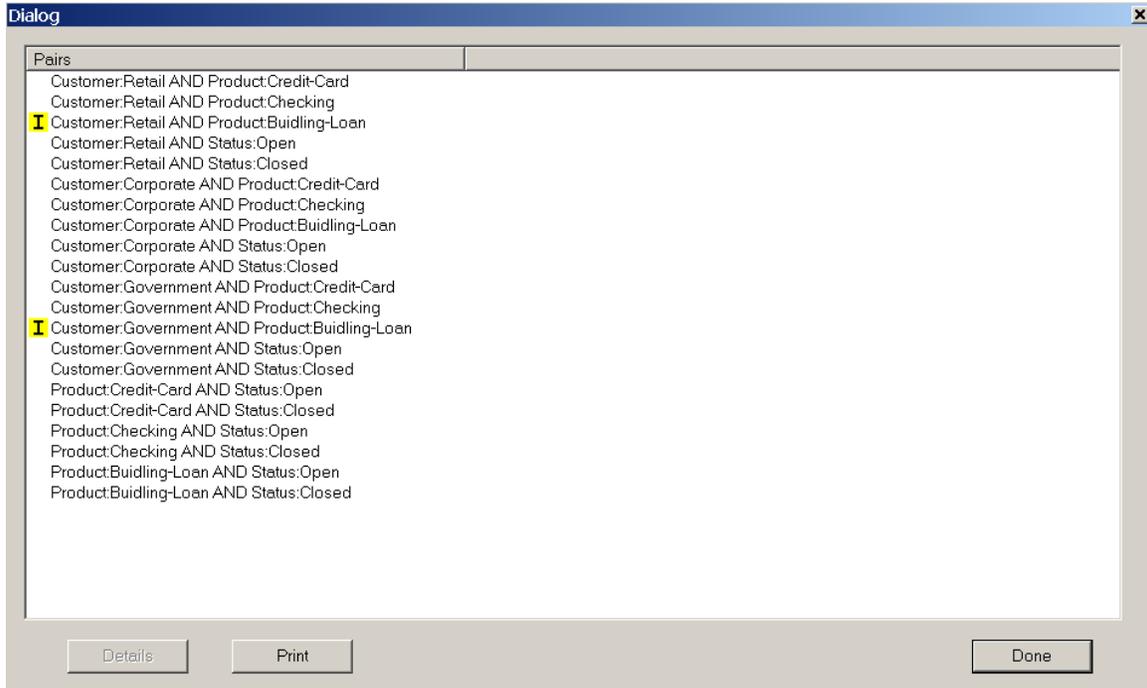
If needed, you then apply constraints across the Variables/States which identify combinations of data which are physically impossible at this point in the system. However, you still want to do full negative testing.



Defining a Constraint in Quick Design

In this example the constraint rule is that only corporate customers may have building loans. Other functions prior to this one would have rejected any attempt by retail customers or government customers from getting this type of loan. The production data base would not contain any building loans for any customer other than corporate customers. Therefore, we do not want to generate any tests at this point contrary to this rule. Note, however, that in testing the predecessor functions you should have tried creating a building loan for the other customer types. The test result should have been that the loan application was rejected.

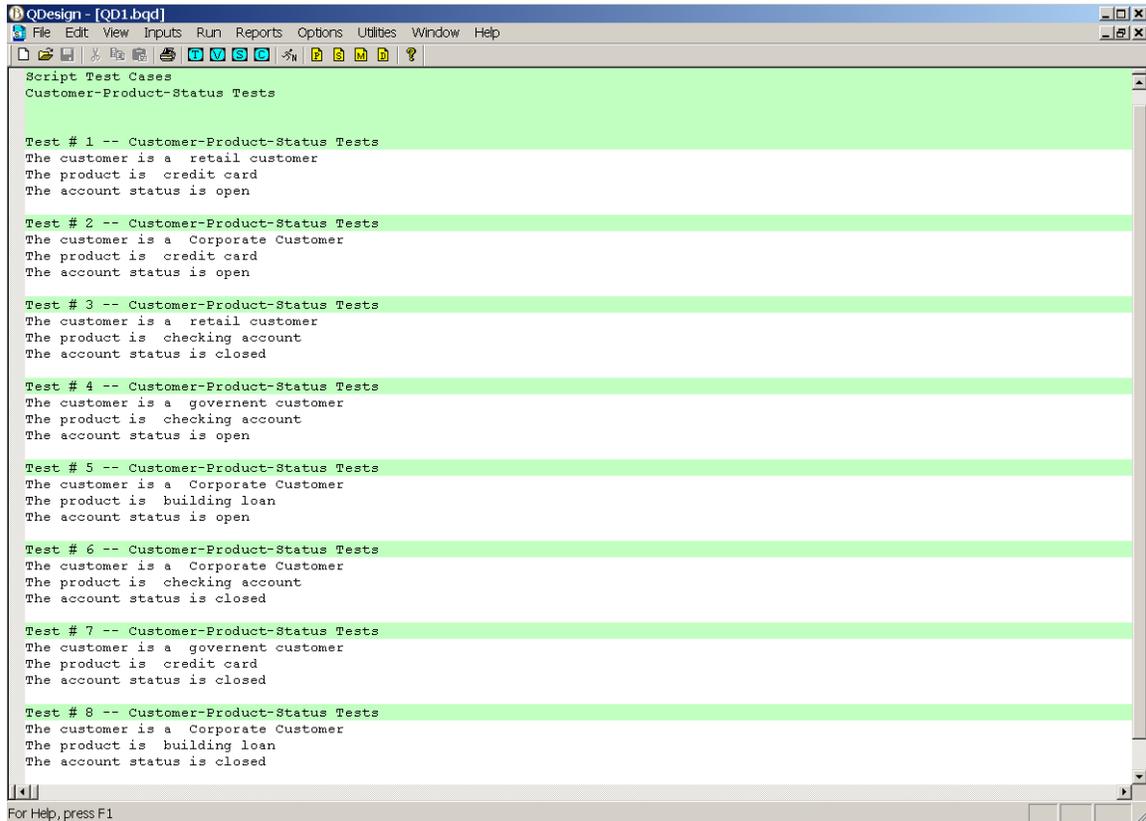
Quick Design then generates all possible pairs across the Variables/States. This is documented in the Pairs Report.



Quick Design Pairs Report

Note that two of the pairs have a yellow "I" next to them. These are the infeasible pairs – i.e. they violated the constraint we set up.

Quick Design then merges the pairs into tests, again ensuring that no constraints are violated. You have two choices in generating tests: Orthogonal Pairs or Optimized Pairs. In Orthogonal Pairs testing each pair occurs the same number of times across the set of test cases. In Optimized Pairs each pair is in at least one test. The goal is to do this in the fewest number of tests possible. We generally recommend orthogonal pairs for configuration testing and optimized pairs for function testing.



Quick Design Test Scripts Report

As in the Cause-Effect Graphing component, you have the options of creating new tests, evaluating old tests, supplementing old tests as needed, and revising descriptions. You can also define pre-existing tests not created by Quick Design.

As in Cause-Effect Graphing, you get the coverage report.

QDesign - [QD1.bqd]							
File Edit View Inputs Run Reports Options Utilities Window Help							
P a i r	T e s t # 1	T e s t # 2	T e s t # 3	T e s t # 4	T e s t # 5	T e s t # 6	T e s t # 7 8
Customer:Retail AND Product:Credit-Card	#						
Customer:Retail AND Product:Checking		#					
Customer:Retail AND Product:Building-Loan	Infeasible						
Customer:Retail AND Status:Open	#						
Customer:Retail AND Status:Closed		#					
Customer:Corporate AND Product:Credit-Card	#						
Customer:Corporate AND Product:Checking					#		
Customer:Corporate AND Product:Building-Loan				X			X
Customer:Corporate AND Status:Open	X		X				
Customer:Corporate AND Status:Closed					X		X
Customer:Government AND Product:Credit-Card						#	
Customer:Government AND Product:Checking			#				
Customer:Government AND Product:Building-Loan	Infeasible						
Customer:Government AND Status:Open			#				
Customer:Government AND Status:Closed							#
Product:Credit-Card AND Status:Open	X	X					
Product:Credit-Card AND Status:Closed							#
Product:Checking AND Status:Open			#				
Product:Checking AND Status:Closed		X			X		
Product:Building-Loan AND Status:Open				#			
Product:Building-Loan AND Status:Closed							#
Unique Pairs	2	1	2	3	1	1	3
Total Pairs	3	3	3	3	3	3	3

Quick Design Pair Coverage Report Optimized Tests

Quick Design also has a utility to calculate coverage based on which tests passed. You can also define subsets of the set of tests with maximum coverage.

You also get the Test Definition Matrix.

	Test # 1	Test # 2	Test # 3	Test # 4	Test # 5	Test # 6	Test # 7	Test # 8
Customer:Retail	T	F	T	F	F	F	F	F
Customer:Corporate	F	T	F	F	T	T	F	T
Customer:Government	F	F	F	T	F	F	T	F
Product:Credit-Card	T	T	F	F	F	F	T	F
Product:Checking	F	F	T	T	F	T	F	F
Product:Building-Loan	F	F	F	F	T	F	F	T
Status:Open	T	T	F	T	T	F	F	F
Status:Closed	F	F	T	F	F	T	T	T

Quick Design Test Definition Matrix

Minimum System Requirements

- ◆ Windows XP, Vista, Win7, Win 8, Win10
- ◆ 128 Mb RAM
- ◆ 30 Mb hard disk space for the programs, documentation, and examples
- ◆ Free disk space for work files (amount of free space required will vary by organizational needs)