

TestCaseGenerator

A Quick Start Tutorial

© 2007 M.Bulmahn

Contents

Introduction	2
Features at a glance	3
Test-Definition	3
Test case generation and management.....	3
Integration	3
Use Cases	3
Tutorial: How to use it.	4
The Specification for the SUT.....	4
The Test Model: One simple question	4
Generating test cases – A small step for a man but	6
Declaring and assigning Results	7
Adding results	7
Rules rule!	9
Creating a rule.....	9
Checking the test model – or the specs?	11
What’s more?.....	13
Rules for obsolete test cases	14
Null Condition	14
Example Data and XML-Export (1.2)	14
Filetypes and command line	14
Further development.....	14

Introduction

TestCaseGenerator is basically a tool for test case generation in combinatorial test scenarios. Since most test scenarios are somehow of combinatorial nature or can be transformed into it by breaking down to equivalence classes in defined states, this tool will help you in many situations where one has to build test cases in a systematical and reproducible way.

The tool itself implements a simple combinatorial algorithm which will lead to so called “exhaustive” test cases. Because we do not have time to run all these test cases in the real world, there are different ways to reduce the number of testcases:

- Certain combinations can be defined as “not possible” or “invalid” cases and so don’t have to be executed.
- The number of combinations can be dramatically reduced by using so called “n-wise” combinatorial algorithms. The most famous one is “pair wise”. In simple words this means that every pair of combinations will be tested but not every permutation of pairs. You can find much more on this under <http://www.pairwise.org> . Though this procedure will reduce the number of test cases very much it will not decrease the effectiveness of the test very much.

TCG makes use of both of these methods to reduce test cases. The second one, the “n-wise” combinatorial algorithms, is in fact not implemented in TCG itself but is achieved by instrumenting a tool by Microsoft which also is not included in the binary distribution of TCG. But the good news is, that this tool is for free and so you can download it and use it with TCG (Download is available under <http://download.microsoft.com/download/f/5/5/f55484df-8494-48fa-8dbd-8c6f76cc014b/pict33.msi>). As far as I can see there is no restriction in use of this tool. Technically spoken the tool is instrumented by TCG via its standard command line interface and redirection of stdout to get the result for further processing in TCG.

Features at a glance

Test-Definition

- The SUT can be modeled by its test input parameters and their equivalence classes
- Expected or predicted results can be expressed as boolean results.
- Rules can be defined with logical expressions using the previously defined parameters and equivalence classes linking them to expected results.
- Null conditions: this is something with which you can define unwanted combinations to reduce the number of resulting test cases

Test case generation and management

- A test model as described above along with its resulting test cases is handled by the application like a document in e.g. MS WORD or EXCEL. It can be saved to a file and loaded from it.
- Based on the test model the application will do the dirty work for you. According to the current settings it will generate all possible combinations of parameters and equivalence classes or it will generate only pair wise combinations or “n-wise” combinations using PICT.
- After generation it will apply the rules on the model to link the expected results with the several combinations and to eliminate unwanted combinations.
- You can still override these rules by simply checking or unchecking a result and so it is even not necessary to have result rules defined to use the tool. You can do this all manually. Very handy for reviewing specifications or thinking about logical problems while implementing something.

Integration

- The only integration into the development or test environment at the moment is the export of data. The combinations can be exported to CSV or to a database. The database export is very useful to use the combinatorial data in a data driven unit test scenario.
- Further integration is planned:
 - As a Visual Studio Add-In
 - Analyzing method signatures to build the test input parameters and equivalence classes automatically.

Use Cases

There are several use cases for this tool. I had the following in mind while building it:

- Creating test cases for an already implemented module, mostly without any documented user requirements or any valid specification. Of course this includes the case where everything is well documented.
- During review or walkthrough of specifications. While writing this it comes to my mind that it can also be used for code review, when thinking about the behavior of logical branches inside a piece of code.
- Creating combinatorial data for data load tests

Tutorial: How to use it.

And now for something completely different – How to use it? The following tutorial will show how to use this tool instrumenting a very simple SUT (System under Test): A login dialog.

The Specification for the SUT

Our SUT for this Tutorial is a very simple and well known one: a login dialog for an application. The only purpose of this login dialog is to authenticate a user and let him start the application or block him out from starting it. We assume that there is a user management component behind this login dialog which handles a number of users and a password for each user. So the specification for this example may look like this:

- A textbox for the user name: The user can enter his user name here and if he clicks OK it will make the program look, whether he entered a password and will check if the password was correct. If all these conditions are fulfilled the application will be started
- A textbox for the password: Here the user will enter his password. If the user gave a valid user name and if he clicks on OK the program will compare the entered password with the stored password and the application will start. Otherwise the user will be blocked out.
- A button for “OK” and another one for “Cancel”: OK means: “Yes, I think I entered the correct values and please let me in!” and “Cancel” means “Ok, I give up. Quit and forget it!”.

The Test Model: One simple question ...

The above specification of our SUT gives us all the information we need for creating the test model. The test model is the back bone of the test case generation and is mandatory for any further steps in the tools. But before we lay hands on the tool we have to do some mental work by finding answers for one simple question:

What are the test input parameters and what are their equivalence classes?

Unfortunately the tool is not able to find answers for this – not now ;-) So we have to go back to the specification and have to analyze it. In this case it is very simple: We have three (some might say four) parameters:

- Username
- Password
- Button

Some might say there are four: OK and Cancel. This would also work but we will take this simplified version because essentially the OK and the Cancel button do the same: they initiate a state change in the application that will generate the result: Program starts or Program will not start.

So the next step is to describe the possible equivalence classes for the parameters. In our case these would be:

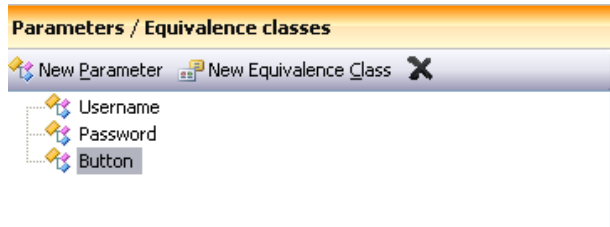
- Username: [Known username, Unknown username, Empty username]
- Password: [Correct password for given user, Incorrect password for given user, Empty password]
- Button: [OK, Cancel]

You see the equivalence classes are somehow the logical possible states of the parameters. Theoretically a text box with a size of 255 characters can have some hundred thousand possible states. By adding specified application logic we can reduce it to three possible states. This is actually the first step in test case reduction!

So now back to the tool: simply start the application and create your first test model with it.

Adding the test input parameters

The first thing we have to do is adding the parameters. To do this simply click on the “New Parameter” – Button and the program creates a “Neuer Parameter” which has to be renamed to “Username”. Do the same for the other two parameters. You should have this at the end:

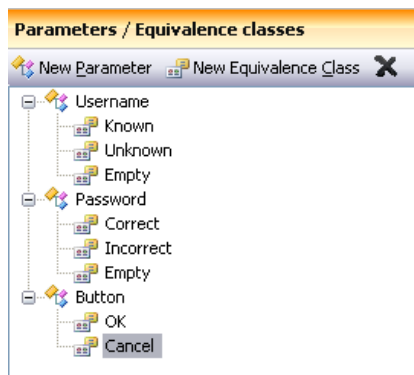


Hints:

- With the fat “X”-Button you can delete the currently selected parameter or equivalence class.
- You can rename the parameters the same way you would do this in an explorer tree. Simply click on the parameter and wait a second.

Adding the equivalence classes for each parameter

Again select the parameter the Username and click the button “New Equivalence class”. This makes the program add a new entry under the username for a new eq. class which then has to be renamed to the desired value – In our case: “Known”. Do this according to the description above. In the end you should have the following:



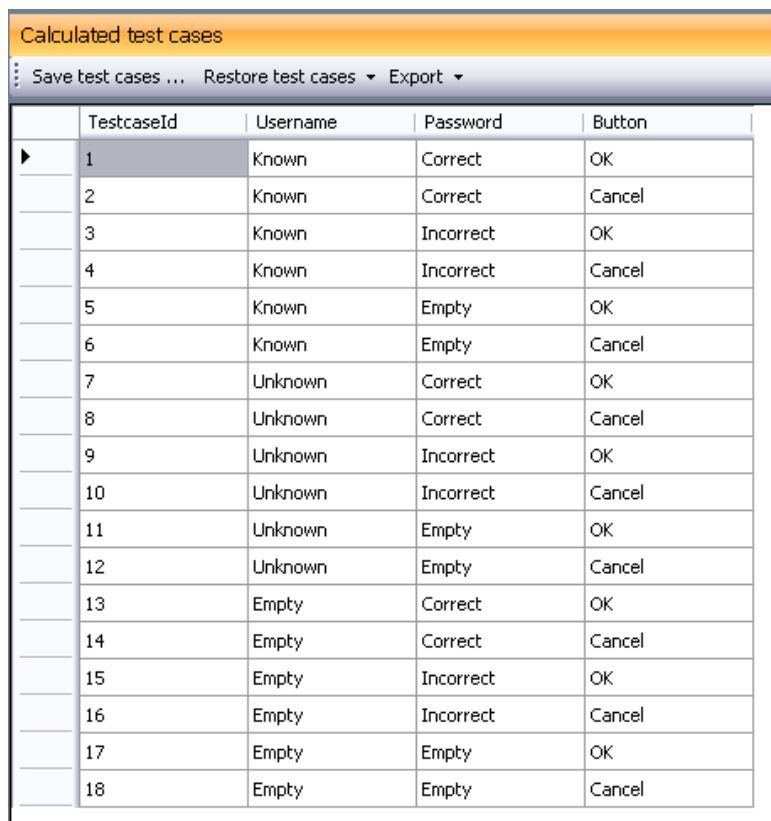
Hints:

- Use keyboard shortcuts to work faster: ALT+P for a new parameter and ALT+C for a new class. This will make you very fast entering your data.
- The naming of the parameters and equivalence classes is arbitrary. The names don't mean anything for the tool and you can type what you want.

Generating test cases – A small step for a man but ...

So you made it until here and I think you should get a little reward. Essentially everything is done to generate test cases! So simply go the "Testmodel"-Menu and select the function "Generate Test Cases". The program will warn you about something you will understand later so you can click on OK. And voila: there appears a grid on the right side which will show you all possible combinations of your test parameters and equivalence classes:

You should see something like this if you build your model like it was described above.



The screenshot shows a window titled "Calculated test cases" with a menu bar containing "Save test cases ...", "Restore test cases", and "Export". Below the menu is a table with 5 columns: "TestcaseId", "Username", "Password", and "Button". The table contains 18 rows of test cases, numbered 1 to 18. The first row is highlighted with a mouse cursor.

	TestcaseId	Username	Password	Button
▶	1	Known	Correct	OK
	2	Known	Correct	Cancel
	3	Known	Incorrect	OK
	4	Known	Incorrect	Cancel
	5	Known	Empty	OK
	6	Known	Empty	Cancel
	7	Unknown	Correct	OK
	8	Unknown	Correct	Cancel
	9	Unknown	Incorrect	OK
	10	Unknown	Incorrect	Cancel
	11	Unknown	Empty	OK
	12	Unknown	Empty	Cancel
	13	Empty	Correct	OK
	14	Empty	Correct	Cancel
	15	Empty	Incorrect	OK
	16	Empty	Incorrect	Cancel
	17	Empty	Empty	OK
	18	Empty	Empty	Cancel

There should be 18 combinations. This is what we normally call "exhaustive": every possible combination will be tested.

Now if you downloaded the PICT.EXE and copied it to the correct location you can start playing with the combinatorial algorithms. E.g. you want to create pairwise test cases. To achieve this click on the

Generator-Button in the tool bar and select “Microsoft Pairwise” from the drop down menu. Then change the Parameter “Combinatorial depth” in the toolbar to the value of “2”. This will make PICT generate pairwise combinations. Of course in a more complex model you can vary this parameter to 3 or 4 to get more test cases. But also with “only” pairwise the chance will be very high that you will find the same number of bugs you would find with the exhaustive way. You should really spend a little time on this topic and google around it since it can really help you save time (and money) in your software testing.

Declaring and assigning Results

So we can have a first result very early but if we look at a single test case then we see that it lacks a very important information: what should the expected result be? For a complete test case description there has always got to be an expected result.

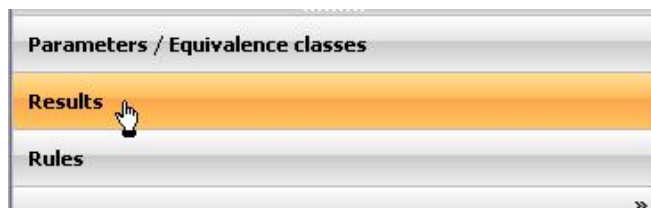
The reason why we don’t find any expected results here is simple: we didn’t even declare the possible results for this test model. This is one thing we can do right now. Taking everything into account after reading the specification there can only be three results:

- The program will start after the user has authenticated with the correct credentials
- The program will quit because the user clicked on cancel
- The program will inform the user that he entered an unknown username or wrong password.

And that’s exactly what we will enter into our testmodel:

Adding results

Change to the “Result”-Area of the tool by selecting the appropriate navigation button:



Then you can enter any expected result by clicking “New Result” or pressing ALT+E and rename the new created result to the desired value, e.g. “Program start” or “Program quit”. In the end you should have this:

Results		
New Result X		
Result	Description	Resulttype
Start		Allgemein
Quit		Allgemein
Message Bad User or Password		Allgemein

Hints:

- You can double click on a result item to edit more details in it: a further description and a type. The type is not supported at the moment but will be implemented in later versions.
- Again the fat “x” will delete the current selected item.

If you now repeat the steps to generate the test cases you will see additional columns in the grid on the right side for each expected result, like this:

Calculated test cases							
Save test cases ... Restore test cases ▾ Export ▾							
	TestcaseId	Username	Password	Button	Start	Quit	Message Bad User or Password
▶	1	Known	Correct	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	2	Known	Correct	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	3	Known	Incorrect	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	4	Known	Incorrect	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	5	Known	Empty	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	6	Known	Empty	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	7	Unknown	Correct	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	8	Unknown	Correct	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	9	Unknown	Incorrect	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	10	Unknown	Incorrect	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	11	Unknown	Empty	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

The big question is: what can I do with this?

Well, it depends:

- Let’s assume that you are in the design phase of your login dialog. You have not completely thought about every aspect of your somehow drafted business logic. This is the point where you can clear it up. Just go through every case and consider what the expected result should be. E.g. first line: “The username is known and the password is correct and the user clicked ‘OK’ – the expected result is that the program will start”. So just activate the checkbox in the start column. Do this for every case (NO! Not really – I’ll show you a smarter way!!).
- Let’s assume you are “QAMan / QAWoman” – then simply take the list and ask the developer to fill in the expected result for every case. I am sure the developers will discuss the need of complete specification after you did this some times ;-)

So enough sarcasm. The important thing is this: you can define the expected results here and they will be saved when you go to “File/Save” or “File/Save as”. They will even be kept when you regenerate the model via “Testmodel / Generate Test cases and keep results”.

Rules rule!

No, the author is not a representative of some new law and order mentality. But rules in this context build the glue between input parameters and expected results. If we look at our specification we can extract some rules:

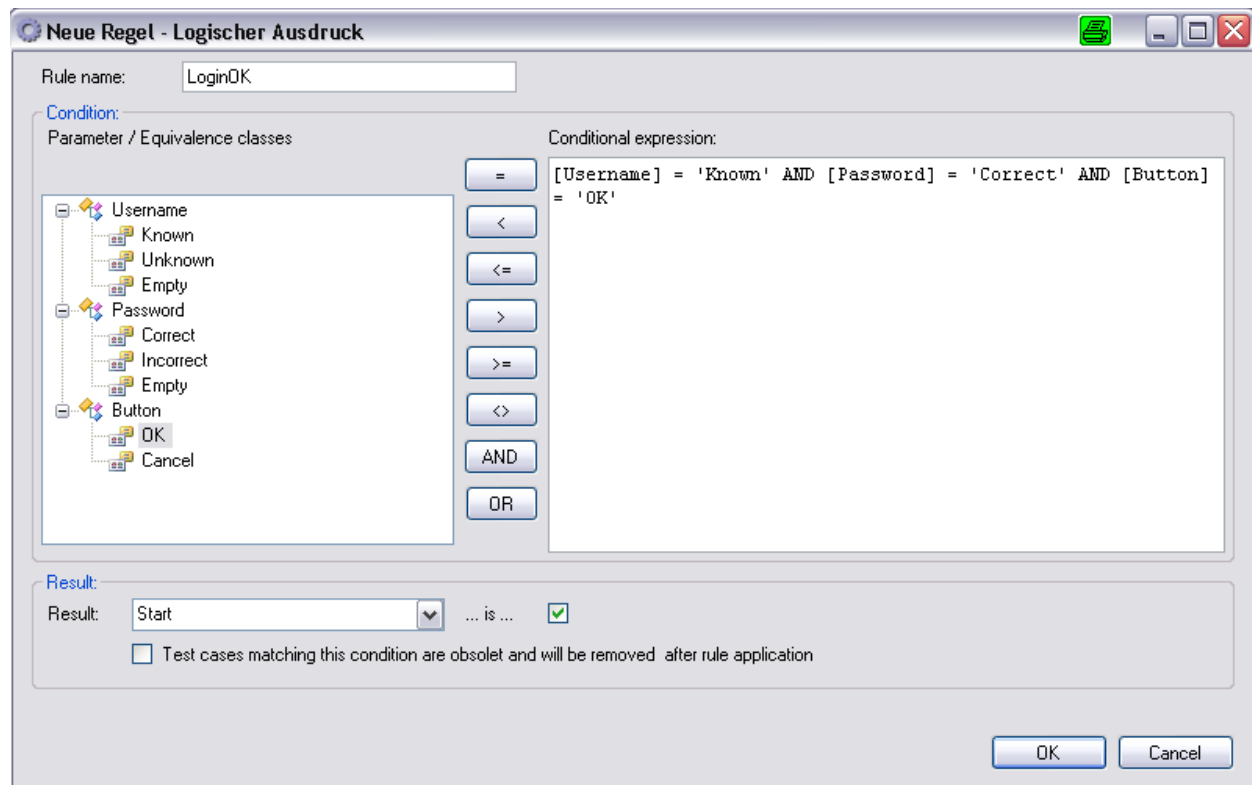
- If the user is known and the password is correct and the OK-Button was clicked the program should start
- If the user is unknown or the password is incorrect the program should display an adequate message.

Now it would be a good thing if we could add these rules to our model because the TCG would be able to assign the correct expected results in every test case. Amazingly enough this is exactly what we will do in the next step 😊.

Creating a rule

First navigate to the rules area within the left hand navigation bar. Then click “New Condition” and double click the new created condition to edit it. Our first rule will be expressed like this:

“If the user is known and the password is correct and the OK-Button was clicked the program should start”:



As you can see the rule has to have an arbitrary name and the conditional part of our rule is expressed using the defined parameters and equivalence classes. You can either write this expression by hand in

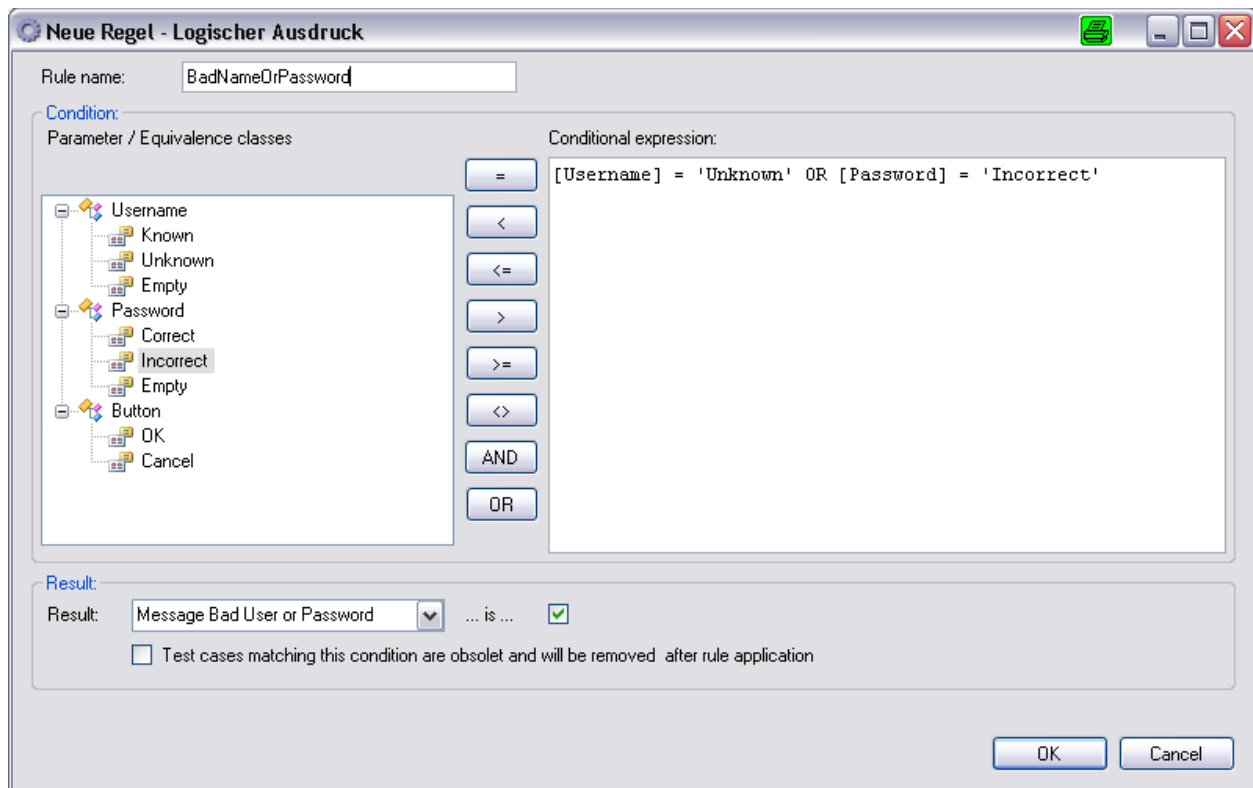
the textbox on the right side or just double click on the item in the tree and click on the operators in the middle. Using this you would create the above expression by these steps:

- Double click "Username"
- Click "="-Operator-Button
- Double click "Known" und "Username"
- Click "AND"-Operator-Button
- Double click "Password"
- ... and so on.

The imperative part of our rule is expressed via a result and its nominal boolean value. In our case we say, "if this condition is fulfilled" then "Start" shall be "True".

Create the other expressions the same way:

"If the user is unknown or the password is incorrect the program should display an adequate message."



So now we have three different rules which should connect all possible input combinations with all expected results. Let's try it out by generating the test cases and applying the results automatically using the rules. This can be achieved by these two steps:

- Testmodel / Generate Test Cases
- Testmodel / Apply results

After both operations have completed you should see something like this in the result matrix:

	TestcaseId	Username	Password	Button	Start	Quit	Message Bad User or Password
▶	1	Known	Correct	OK	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	2	Known	Correct	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	3	Known	Incorrect	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	4	Known	Incorrect	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	5	Known	Empty	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	6	Known	Empty	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	7	Unknown	Correct	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	8	Unknown	Correct	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	9	Unknown	Incorrect	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	10	Unknown	Incorrect	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	11	Unknown	Empty	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	12	Unknown	Empty	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	13	Empty	Correct	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	14	Empty	Correct	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	15	Empty	Incorrect	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	16	Empty	Incorrect	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	17	Empty	Empty	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	18	Empty	Empty	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

We can see that TCG assigned expected results to some of the test cases based on the given rule set – though not for all. This is an indicator that our test model is somehow incomplete. How to systematically find “wholes” in the model will be shown below.

Checking the test model – or the specs?

Now the interesting part begins. The last result was, that there are wholes in our test model because expected results were not assigned for all test cases. The tool gives you some help at this point. Just select the function “Testmodel / Check model” and you will get some further information:

Calculated test cases

Save test cases ...
Restore test cases
Export

	TestcaseId	Username	Password	Button	Start	Quit	Message Bad User or Password
▶	1	Known	Correct	OK	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	2	Known	Correct	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	3	Known	Incorrect	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	4	Known	Incorrect	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	5	Known	Empty	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	6	Known	Empty	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	7	Unknown	Correct	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	8	Unknown	Correct	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	9	Unknown	Incorrect	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	10	Unknown	Incorrect	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	11	Unknown	Empty	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	12	Unknown	Empty	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	13	Empty	Correct	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	14	Empty	Correct	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	15	Empty	Incorrect	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	16	Empty	Incorrect	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	17	Empty	Empty	OK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	18	Empty	Empty	Cancel	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

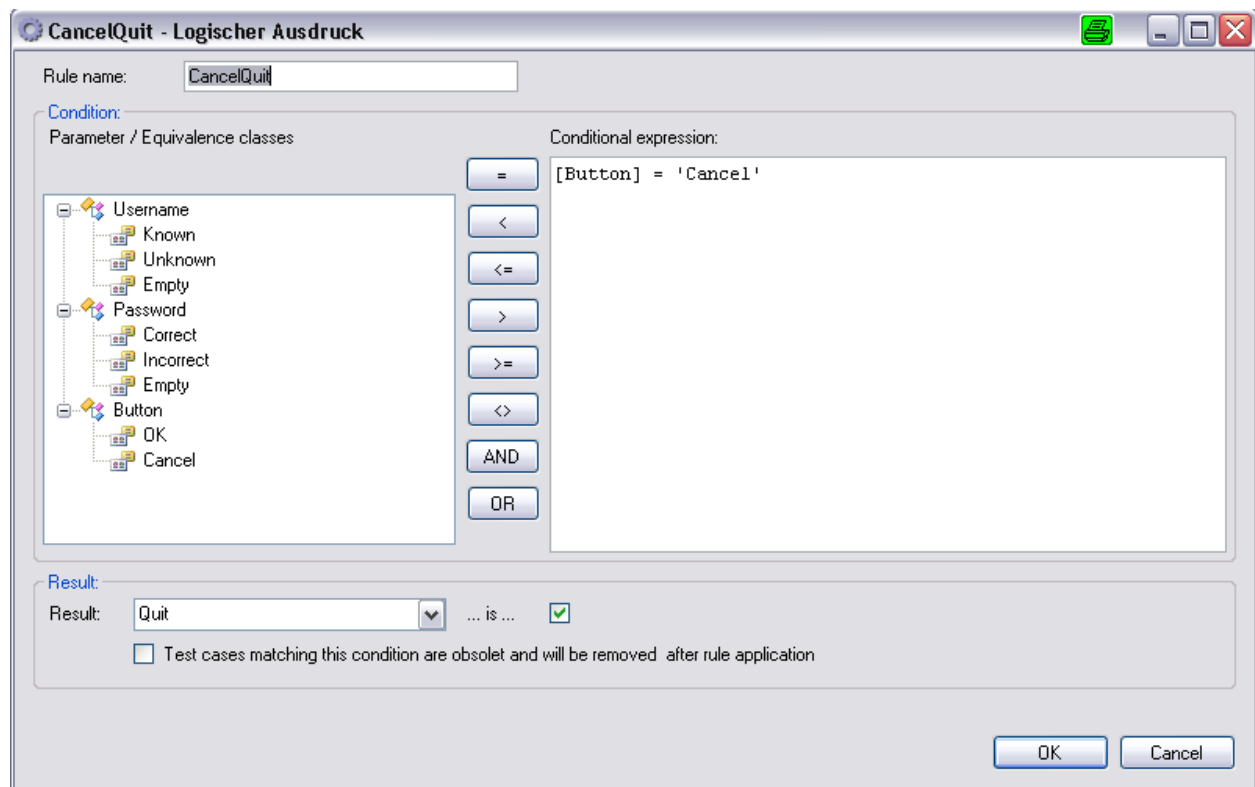
Messages

No results assigned to tEst case No.5!
No results assigned to tEst case No.6!
No results assigned to tEst case No.13!
No results assigned to tEst case No.14!
No results assigned to tEst case No.17!
No results assigned to tEst case No.18!
Result 'Quit' is never used in the model!

The program shows you all test cases that have no assigned results by marking these line with red background color. In the messages window you can see the same information (“No results assigned for test case No. XY”) and one more – very important – information: “Result ‘Quit’ is never used in the model”. So we declared a result in the model that was never used / assigned. This means that either the result is obsolete or that we have to declare on more rule for this result.

In our case we have to do the latter:

“If the user clicked the cancel button the program should quit.”



This will bring us one step closer to a complete model, but just closer. The reason is that our model still lacks some conditional information: What shall the expected result be if either username or password are left blank? Looking at our specification we can see that there is no expected result for this condition. So what we actually have here is a bug in the specification. This is very important because it shows us that using this tool at specification time can be a very good way to prove assumptions.

To complete the model we will modify the “BadNameOrPassword”-Rule:

[Username] = 'Unknown' OR [Password] = 'Incorrect' OR [Username] = 'Empty' OR [Password] = 'Empty'

So our program should give the same error message for empty input as it gives for wrong input. Of course in a real world application this decision should not be made while test case design but on specification level.

What's more?

So we walked through nearly every corner of this small tool in this tutorial. There are some things left that have to be mentioned here. Mainly these features are for test case reduction. So let's look at them:

Rules for obsolete test cases

As we have seen before a rule will define what nominal state a certain result shall have if the rule's condition is fulfilled. It is also possible to say "Hey, whenever this conditions applies, the whole test case is not interesting / waste of time / does not make sense / is obsolete". So you can check the option "Test cases matching this condition are obsolete". If you do this – and you should really know what you are doing! – the program will remove all test cases that fulfill the rule's conditional expression.

Null Condition

There are situations when a certain state of one parameter causes another parameter to be obsolete. Perhaps our little login dialog could be designed so that the buttons are disabled as long as one of the text boxes are empty. So one could say Parameter "Button" is obsolete if "Username='Empty' OR Password='Empty'". This would be the exact expression you have to enter when defining a null condition on parameter "Button". If you do so, all test cases matching this condition would exclude parameter "Button" from the test.

This feature is not very userfriendly at the moment, because you have to care for the correct syntax of the expression and there is no input assistance. This will change in the future because I think this is a very important feature to express dependencies between input parameters. And of course it can also help reducing test cases without scarifying the correctness of the test.

Unlike the "obsolete rules" the null conditions are applied at generation time. So again here you should know what you do when you do something here ;-)

Example Data and XML-Export (1.2)

Data driven unit tests in VS 2008 are a very efficient way to test complex functions. A testmodel in TestCaseGenerator can function as the input generator for such a test. For this scenario you can assign example data for each equivalence class. By using the function "Testmodel / Fill in example data" you can transfer these example values to the test case table. Then a Export to XML (new export option) generates XML, that can be used directly as a data source for VS 2008 data driven unit tests.

Filetypes and command line

Under Help there is a new function for registering filetypes "TestModel" with the application in the current installation directory. Another new option in the same context is the possibility to give a filename on the command line to be opened on startup.

Further development

In the last few weeks there were only little changes. The most fundamental was the introduction of the parameter null condition. The program fulfills my personal needs at the moment and so there are no big plans for it. Two things I would like to do next are:

- Integrating it in to VS 2008 as an Add-In with the ability to manage the testmodel-Files in the project structure.
- Building the parameters and equivalence classes from Method signatures. This can be interesting for generation of test cases in unit tests.
- Perhaps a integration with TFS2008 with its own item type?
- ...

So you see, feedback is needed! Feel free to play / work with this tool and let me know what your wishes are! Feedback can either be given by posting on codeplex or sending me an eMail to

mBulmahn<AT>web.de

Of course I would be happy if somebody wants to work together with me on this project!