

Using Stub Functions

Abstract

TESSY recognizes all external functions that are called by a test object and may provide stub functions as a replacement for the original function returning certain values. This document describes how to choose the right stub mechanism and where to enter the code or values needed.

Table of contents

1	Introduction.....	2
2	Defining and editing stub functions.....	3
2.1	TIE.....	3
2.2	Stub functions view	4
3	Advanced stubs.....	6
3.1	TIE.....	6
3.1.1	Passing directions	7
3.2	TDE	7
3.3	Test report.....	9
4	Providing stub functions using synthetic variables	10
4.1	Creating synthetic variables	10
4.2	Implementing the stub code	12
4.3	Entering test data	13
5	Using code from objects/libraries	14

1 Introduction

Within a C/C++ source code, a function may either be defined or just declared external. Because there is no code available for external functions, you need to provide a function to be called instead of the missing original external function your test object is calling. TESSY provides means to define missing external functions through its stub function mechanism. It is also possible to replace local functions defined within the source file by stub functions (instead of calling the original function implementation).

TESSY also provides a mechanism called **Advanced Stubs**. This allows to handle parameter and return values of external functions like normal input or output values of the test object. The parameter and return values will appear within the TDE and the test report, depending on their passing direction settings. In case of scalar types, it is possible to provide multiple different values for each invocation of the stub function.

Another approach for providing parameter or return values of stub functions uses **synthetic variables** as described within chapter 4. This is especially useful if you need to provide different values of complex type parameters or return value for each invocation of the stub function.

Yet another possibility to provide code for stub functions is to link object files or libraries containing stub functions to the test driver. In this way, you may also perform an integration test by linking against the (previously tested) underlying functions of your actual test object.

Each possibility of using stub functions will be explained in the following sections using the following sample source code:

```
#define MAX_ENTRIES 10

extern short getVal (short param1, short param2, short * errorCode);

short list[MAX_ENTRIES];

void setList (short value, short count)
{
    short i;
    short error = 0;

    if (count <= MAX_ENTRIES)
    {
        for (i = 0; i < count; i++)
        {
            list[i] = getVal (value, i, &error);

            if (error != 0) break;
        }
    }
}
```

Copy this code into a C source file and create a TESSY module in order to retrace the stub implementations described within this document.

2 Defining and editing stub functions

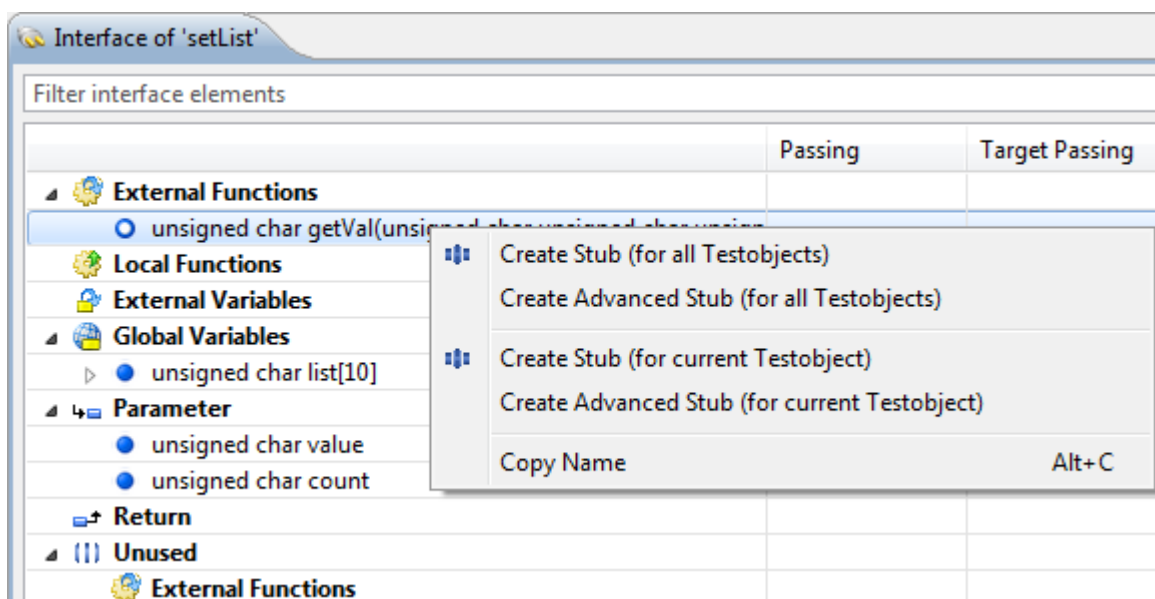
To define a stub function, the following two steps are necessary:

- Let TESSY define a stub function as replacement for the missing external function within the TIE perspective.
- Provide the stub function code within the respective stub functions view within the TDE perspective.

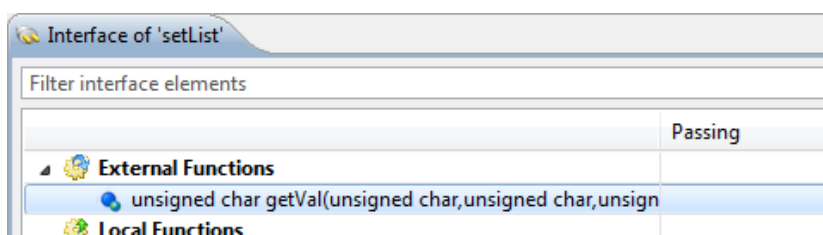
TESSY will then generate a function for each stub function to be defined, with the parameter list and return value as recognized during the source file analysis process. The body of these functions will contain the code entered within the stub functions view (initially empty).

2.1 TIE

All external functions are listed in a separate section **External Functions** within TIE. If you select an entry in that function list, you can choose to create a stub function using the context menu. The blue circle with the white center at the function name indicates, that no stub function will be defined by TESSY (which is the default).



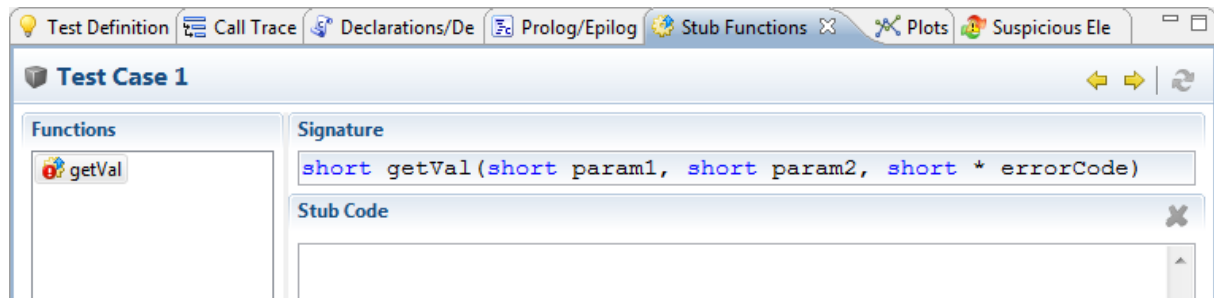
You can choose to create a (normal) stub or an advanced stub as described within chapter 3. When creating a normal stub for all test objects, the external function will be displayed like follows:



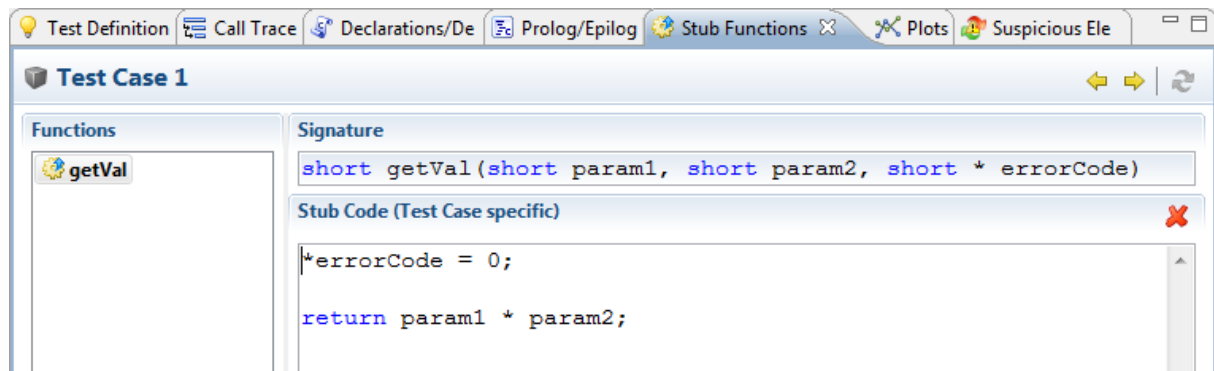
Editing of the stub function code will take place within the stub functions view as described below. Switch to the TDE perspective to edit a stub function.

2.2 Stub functions view

You will find all stub functions to be defined by TESSY within the stub functions view of the TDE perspective. Initially an error will be indicated because our stub function need to return a value but there is no code available (and yet no value is being returned).



The upper part shows the signature of the stub function with the individual parameter names. Add your stub function code in the lower part of the **Stub Code** text field. We will just set the error code being passed as pointer value and return the multiplication of the first two parameters:



You can return different values depending on the test case or test step. Refer to the user manual for details. With our example implementation, the test data and results of our test object could be like shown below:

Test Case 1			
Test Step 1.1 (Repeat Count = 1)			
Name	Input Value		
count	10		
value	2		
Name	Actual Value	Expected Value	Result
list[0]	0	0	✓
list[1]	2	2	✓
list[2]	4	4	✓
list[3]	6	6	✓
list[4]	8	8	✓
list[5]	10	10	✓
list[6]	12	12	✓
list[7]	14	14	✓
list[8]	16	16	✓
list[9]	18	18	✓

Please note: We recommend to use meaningful variable names for parameters within the **declaration of external function prototypes**. TESSY will then be able to show these names within the signature of the stub function. Otherwise you will find TESSY generated names in the stub function signature.

An example for a useful external function declaration and the kind of declaration to be avoided is shown below:

```
//  
// External function declaration with named parameters  
extern void add_values(int x, int y, int *result);  
  
//  
// Such declarations should be avoided  
extern void add_values(int, int, int *);
```

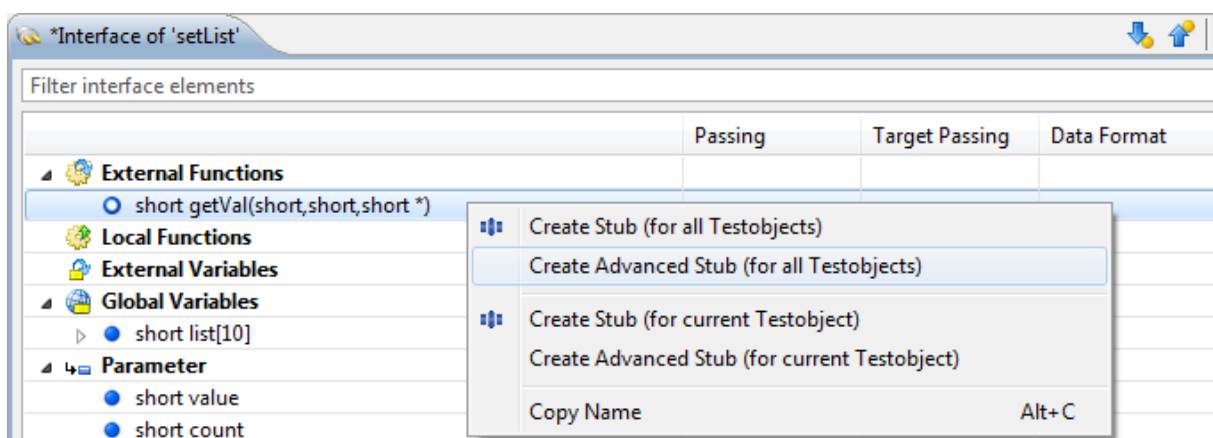
3 Advanced stubs

Advanced stubs may be applied to all external functions that have parameters or a return value. Unsupported types of parameter or return values will be indicated with a passing direction of IRRELEVANT which cannot be changed. In such cases you would need to implement a normal stub as described within chapter 2.

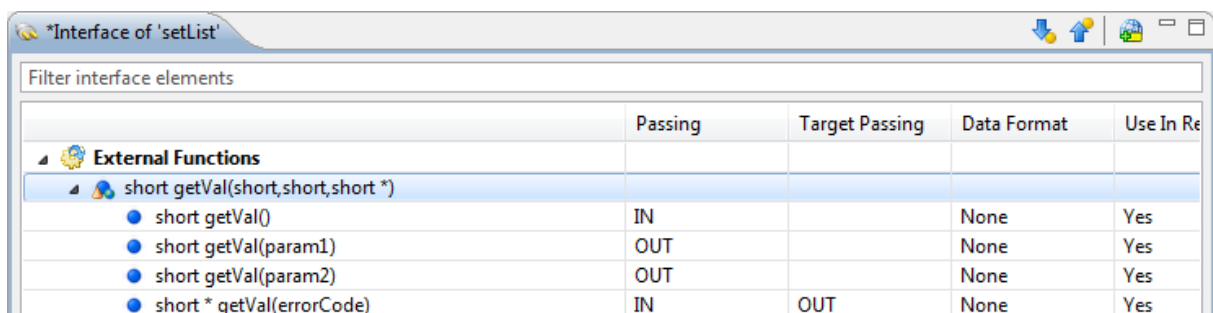
For scalar type parameters and return values you can enter vectors of values being used for subsequent calls to the stub function as described later within section 3.2.

3.1 TIE

From the context menu select **Create Advanced Stub** for the respective external function:



Your selection will be displayed with a little orange decorator indicating the usage of the advanced stub functionality. Also the parameters and return value (if any) will be displayed as children of the function entry, allowing to specify the passing directions.



Please review the settings of the passing directions. As a result, you will see “virtual” variables for each parameter and the return value in the global variables list within TDE and the test report. The advanced stub variables will be named as shown above with the name of the function followed by the parameter name in parenthesis.

3.1.1 Passing directions

The passing directions for parameters and the return value of an advanced stub function are handled slightly different than passing directions for normal variables of the test object's interface:

- If you want to return a value from a stub function, the passing direction for that parameter or return value would be IN, since the value will be passed to the test object and you need to provide a value within TDE.
- Checking if the correct parameter value was passed to the stub function requires the passing direction OUT for this parameter, since the value will be written from within the test object and should be evaluated and documented in the test report.
- Checking if a pointer parameter points to the right target will probably only be useful for pointers pointing to target objects within the test object interface. Otherwise you will always get the value *unknown* as result.

Please note: For technical reasons, parameters of type pointer require the passing direction IN, even if the pointer target value should only be evaluated (OUT).

3.2 TDE

The advanced stub functions require input values and expected values to be specified within TDE perspective. For each advanced stub function parameters and return value, you will find a “virtual” variable in the list of global variables. Whether they appear as input or expected values depend on the passing directions selected within TIE. Each variable will be displayed in parenthesis with the advanced stub function name as prefix.

It is also possible to provide vectors of values for parameters and the return value. The values of such vectors will be used for each subsequent invocation of the stub function. In the example below, the external function is expected to be called five times, so that vectors with five values are provided as expected values for the parameters as well as inputs for the return value.

Please note: Using vector values is only possible for scalar type parameters or return values.

Test Data of 'setList'	
type filter text	
	1.1
Inputs	
Globals	
short getVal()	{0, 2, 4, 6, 8}
short * getVal(errorCode)	target_getVal_errorCode
Parameter	
short value	2
short count	5
Dynamics	
short target_getVal_errorCode	0
Outputs	
Globals	
short getVal(param1)	{2, 2, 2, 2, 2}
short getVal(param2)	{0, 1, 2, 3, 4}
short list[10]	
short list[0]	0
short list[1]	2
short list[2]	4
short list[3]	6
short list[4]	8

For the given example, **each** call of the function `getVal()` within the test object, will return individual values from the given vector. The parameters `param1` and `param2` will be checked against different values from each of the given vectors for each invocation of function `getVal()`.

Because the `errorCode` parameter is a pointer, we have created a dynamic target object `target_getVal_ErrorCode`. We can only provide one value that will be used for **all** invocations of our external function. It is not possible to use vector values here.

Please note: Within the example data shown above, the return value of the stub function is according to the specification the multiplication of both parameters. But you are free to return an erroneous value here for e.g. error guessing tests that may check error handling within the test object.

3.3 Test report

Within the test report, the advanced stub function's “virtual” variables will be listed in the list of input values and expected/actual values like normal global variables.

Test Case 1 ✓			
Test Step 1.1 (Repeat Count = 1) ✓			
Name	Input Value		
count	5		
getVal()	{0, 2, 4, 6, 8}		
getVal(errorCode)	target_getVal_errorCode		
target_getVal_errorCode	0		
value	2		
Name	Actual Value	Expected Value	Result
getVal(param1)	{2, 2, 2, 2, 2}	{2, 2, 2, 2, 2}	✓
getVal(param2)	{0, 1, 2, 3, 4}	{0, 1, 2, 3, 4}	✓
list[0]	0	0	✓
list[1]	2	2	✓
list[2]	4	4	✓
list[3]	6	6	✓
list[4]	8	8	✓

4 Providing stub functions using synthetic variables

In case that advanced stubs and vector values cannot be used because there are complex types (e.g. pointers to structs/unions) being used for parameters or the return value and you need to provide different values for multiple stub function calls, this chapter describes an alternative approach using synthetic variables.

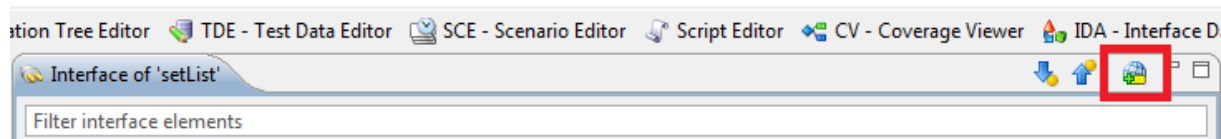
The following steps need to be carried out to implement such stubs:

- Switch to the TIE perspective to create synthetic variables for all parameters and for the return value.
- Switch to the TDE perspective to implement the stub code within the stub function view. The stub code will use the values from these synthetic variables.
- Fill the test data within the test data editor.

As a result, you will see all values used within the stub code as normal test data and listed within the test report.

4.1 Creating synthetic variables

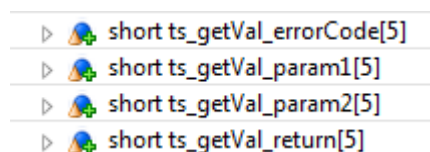
Within the TIE perspective create synthetic variables for each parameter and for the return value.



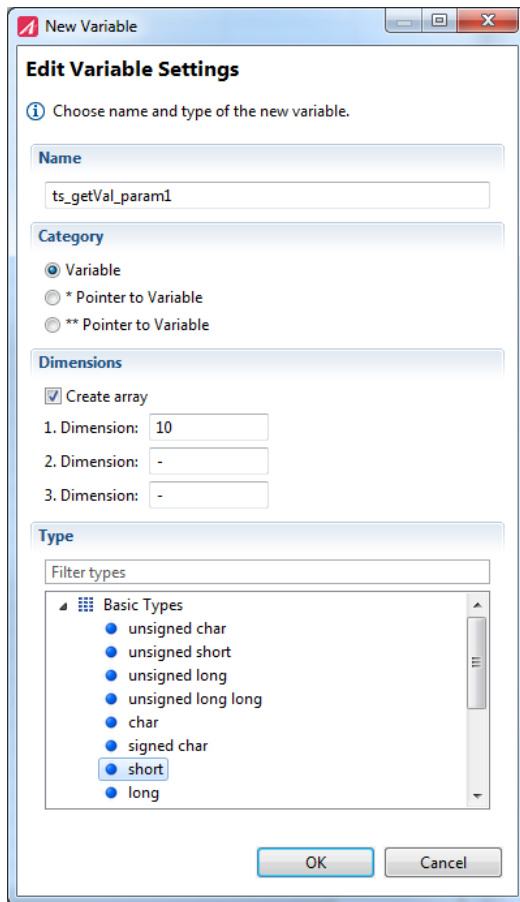
We suggest using a naming convention e.g.

```
ts_<function name>_<parameter name>
```

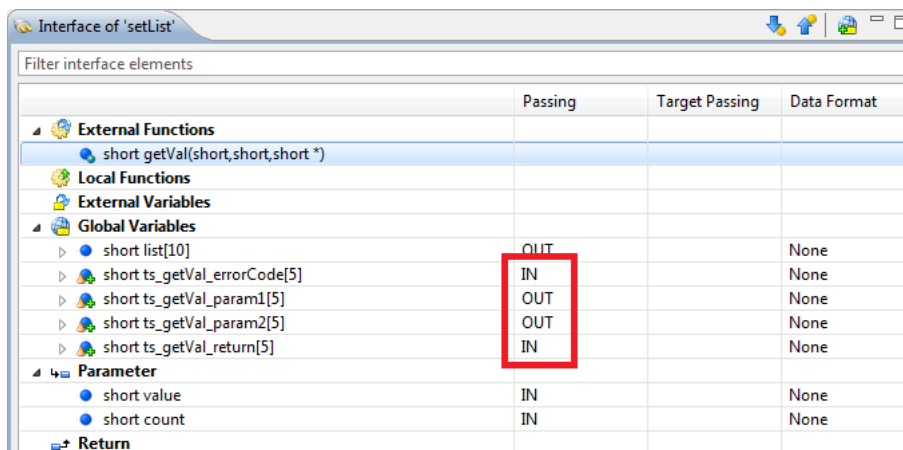
For our example, the following array variables should be created (with a size of five, because the variables will contain the test data of five calls to the stub function):



Specify the dimension and size (i.e the number of expected calls to the stub function) within the **New Variable** dialog:



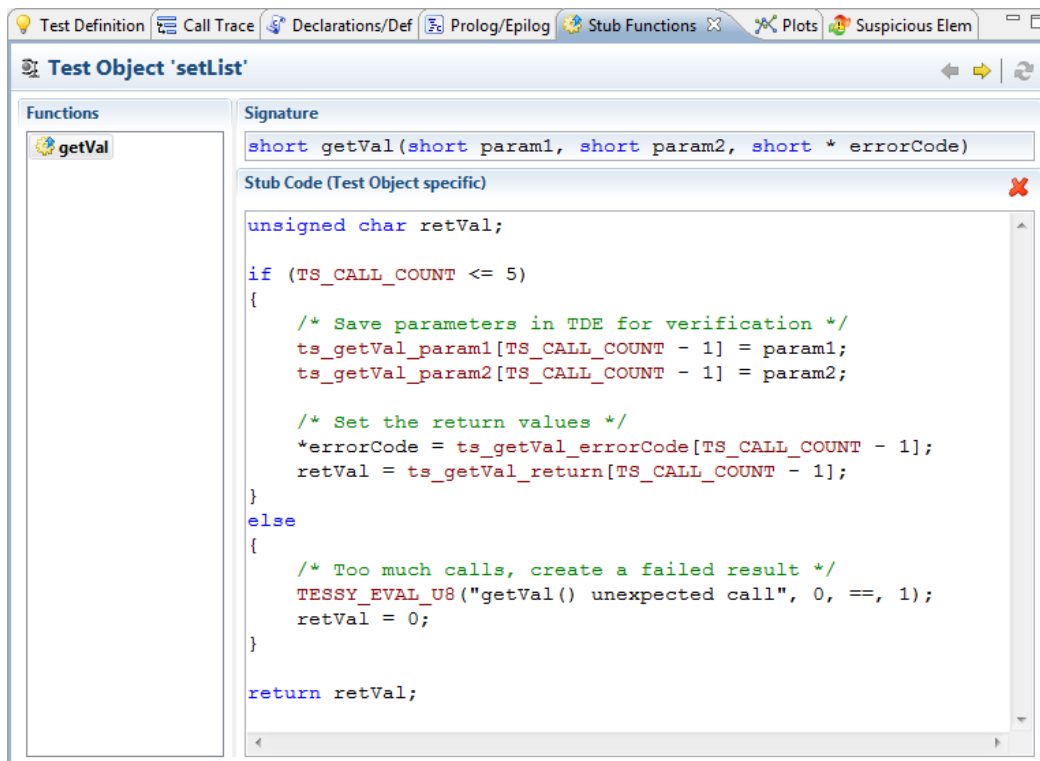
As a result, you will have the following variables being part of the test object interface:



Set the passing directions according to the desired usage as either input values or expected results.

4.2 Implementing the stub code

The stub code implementation uses the special `TS_CALL_COUNT` value in order to transfer the actual parameter and return values to the synthetic variables and vice versa. As a result, all values passed to the stub function can be evaluated within the synthetic variables and values to be returned by the stub function call are taken from the synthetic variables. The implementation for our example is as follows:



```

short getVal(short param1, short param2, short * errorCode)

Stub Code (Test Object specific)

unsigned char retVal;

if (TS_CALL_COUNT <= 5)
{
    /* Save parameters in TDE for verification */
    ts_getVal_param1[TS_CALL_COUNT - 1] = param1;
    ts_getVal_param2[TS_CALL_COUNT - 1] = param2;

    /* Set the return values */
    *errorCode = ts_getVal_errorCode[TS_CALL_COUNT - 1];
    retVal = ts_getVal_return[TS_CALL_COUNT - 1];
}
else
{
    /* Too much calls, create a failed result */
    TESSY_EVAL_U8("getVal() unexpected call", 0, ==, 1);
    retVal = 0;
}

return retVal;

```

Here is the code for reference:

```

unsigned char retVal;

if (TS_CALL_COUNT <= 5)
{
    /* Save parameters in TDE for verification */
    ts_getVal_param1[TS_CALL_COUNT - 1] = param1;
    ts_getVal_param2[TS_CALL_COUNT - 1] = param2;

    /* Set the return values */
    *errorCode = ts_getVal_errorCode[TS_CALL_COUNT - 1];
    retVal = ts_getVal_return[TS_CALL_COUNT - 1];
}
else
{
    /* Too much calls, create a failed result */
    TESSY_EVAL_U8("getVal() unexpected call", 0, ==, 1);
    retVal = 0;
}

return retVal;

```

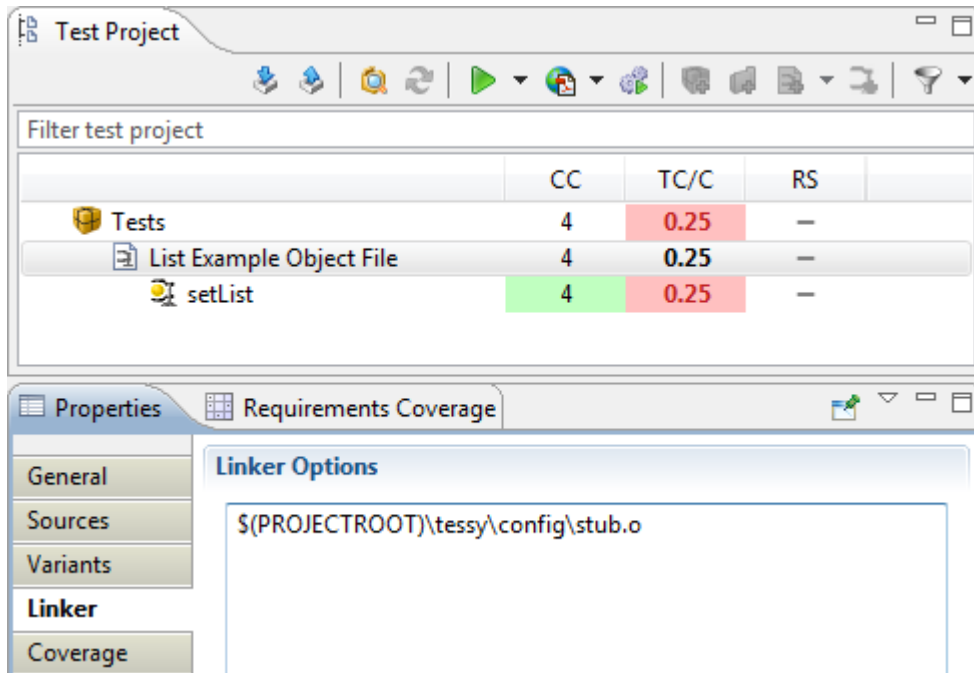
4.3 Entering test data

The test data for all synthetic variables can be filled within the TDE perspective.

Test Data of 'setList'		1.1
type filter text		
Inputs		
Globals		
short ts_getVal_errorCode[5]		
short ts_getVal_errorCode[0]		0
short ts_getVal_errorCode[1]		0
short ts_getVal_errorCode[2]		0
short ts_getVal_errorCode[3]		0
short ts_getVal_errorCode[4]		0
short ts_getVal_return[5]		
short ts_getVal_return[0]		0
short ts_getVal_return[1]		2
short ts_getVal_return[2]		4
short ts_getVal_return[3]		6
short ts_getVal_return[4]		8
Parameter		
short value		2
short count		5
Dynamics		
Outputs		
Globals		
short list[10]		
short list[0]		0
short list[1]		2
short list[2]		4
short list[3]		6
short list[4]		8
short ts_getVal_param1[5]		
short ts_getVal_param1[0]		2
short ts_getVal_param1[1]		2
short ts_getVal_param1[2]		2
short ts_getVal_param1[3]		2
short ts_getVal_param1[4]		2
short ts_getVal_param2[5]		
short ts_getVal_param2[0]		0
short ts_getVal_param2[1]		1
short ts_getVal_param2[2]		2
short ts_getVal_param2[3]		3
short ts_getVal_param2[4]		4
Parameter		
Return		
Dynamics		

5 Using code from objects/libraries

Last but not least, you may provide code for stub functions by linking objects or libraries to the test driver. You may add object or library files on module level into the **Linker Options** field of the **Linker** tab within the module properties.



You should specify relative path names using the `$(PROJECTROOT)` variable instead of using absolute path names for the object file or library to be linked.

To avoid multiple definitions of the stub function, make sure that there is no stub setting specified for the external function within TIE.