Hitex White Paper

TESSY

Automated In-Target Testing as a Means to Achieve Software Quality

By Frank Büchner

April 2002 - 001

Embedding Software Quality

Hitex GmbH
Greschbachstr. 12
D-76229 Karlsruhe
Germany

Tel: +49 721 9628 - 0
Fax: +49 721 9628 - 149

E-mail: Sales@hitex.de
Support@hitex.de

Internet: http://www.hitex.de

April 2002 - 001

Embedding Software Quality

Contents

1. Introduction 4

2. Software Testing Issues 4

2.1. Practical Problems 4
2.2. Requirements of Common Software Quality Standards 5
2.3. Tessy – the Ideal Solution 5

3. Tessy's Basic Functionality 6

3.1. Problem Specification 6
3.2. Interface Analysis 7
3.3. Test Driver Generation 7
3.4. Test Data Acquisition 8
3.5. Automatic Results Prediction/Calculation 8
3.6. Running the Unit Test 9
3.7. Test Evaluation 9
3.8. Debugging 9
3.9. Test Documentation 10
3.10. Handling Externals 10
3.10.1. External Variables 10
3.10.2. External Functions 12

4. Regression Testing with Tessy 14

4.1. Test Case Adaptation 14
4.2. Further Use of Test Data 14

5. Test Coverage with Tessy 15

5.1. Line Coverage 15
5.2. Path Coverage 15

6. Integration Testing with Tessy 17

7. Conclusion 17

1. Introduction

The crucial issue in achieving software quality is for the software to have zero defects. This objective overrides others such as maintenance, portability and performance since if bugs are responsible for a state of disorder, it would most likely result in the entire product loosing any credibility for quality and value. Dynamic testing is the main method used to prove that software contains zero defects.

2. Software Testing Issues

2.1. Practical Problems

Although nobody denies the need for dynamic software testing, until now there has been virtually no tool support for this important task in the software development process. Therefore, dynamic software testing is mostly performed interactively. This means that the test object (the function under test) is loaded into a debugger and executed. The input values for the test object are entered manually, using the debugger. Then the test object is executed and its behavior monitored. After execution, the results (i.e. the output values) are checked against the expected values.

This raises some practical problems:

- Manually-driven dynamic software testing is very time-consuming.
- Besides being tedious and cumbersome, the process is error-prone. The entered test data is selected ad-hoc, and how carefully the results are checked depends on the human tester and is subject to variation.
- Normally neither the test data nor the test environment is saved so it is not possible to repeat tests exactly.
- If the software is changed, all previous tests should be repeated. This holds even true if the software was changed only slightly, e.g. for a bug fix. If the software was changed, a new compiler version is used to compile the software, or if the software is ported to another microcontroller architecture, repeating all tests is strongly recommended. If this has to be done manually, it often takes more time than is feasible and therefore only the most important tests are repeated.

Granted, most of the above listed problems are addressed in most software development departments because of the great importance of dynamic software testing. The general approach is to implement a proprietary test environment. Of course, the quality of this test environment depends on the effort employed. Therefore mainly in the big companies, which can afford it, there exist comprehensive test environments for embedded software. These test environments can be adapted to new requirements (new test objects, compilers, etc.) and are well maintained. However, the common situation is that a test environment is tailored to a specific test object; the database and scripting tools involved are the favourite tools of the tester and adapting the test environment during the development process to changing test objects (with steadily changing interfaces) takes as much effort as extending the test object itself. Adapting these home-grown test environments to new projects is normally so much effort that the current test environment is dropped and a new one is invented for each project.

2.2. Requirements of Common Software Quality Standards

Nowadays, more and more standards that address the quality of the software development process or methods are available to project managers. Examples of such standards are Bootstrap, the Capability Maturity Model (CMM), SPiCE (ISO/IEC 15504), various military standards like the British Defence Standard 00-55, etc. These standards were driven by the military and industries where reliability under all circumstances is essential (e.g. avionics, aerospace, medicine). The availability of those standards leads to their application in other industries like automotive and telecoms. There is a wide variety of standards with each industry / company tending to have its own and they normally cover a great number of issues. However when it comes to testing, all standards have more or less the same requirements, most of which are really common sense:

- Necessity of planning the tests in advance. This includes the determination of the test cases - how many, which and the provision of the required input and output test data.
- Necessity of documenting the tests. This should lead to full reproducibility of the tests; date of the test, software version tested, test data used, test outcome, version of test environment used, etc.
- Necessity of unit and integration testing. This means intensive testing of single units (functions in the C language) and then integrating these units step-by-step to eventually form the complete software system.
- Necessity of determining the test coverage. This reveals which parts of the source code were exercised by the tests and, more importantly, which were not.

Even if projects not conducted according to standards or standard methods, it is a good idea to address the issues mentioned above to at least some extent.

2.3. Tessy – the Ideal Solution

Tessy is a commercially available tool that facilitates the automatic dynamic testing of software for embedded systems. The functionality of Tessy addresses almost all problems and requirements mentioned above.

3. Tessy's Basic Functionality

Tessy can be brought into use as soon as a suitable compiler for the target hardware can compile the software module containing the C function to be tested. Tessy analyses the software module and lists all C functions it finds within it. The user can then select which function is to be tested.

3.1. Problem Specification

A very elementary problem specification forms the starting point of the survey of Tessy's basic functionality:

> A range of values is specified by a given start value and a given length. Is an input value within this range or not?

Or, more mathematically:

> range_start <= v1 < (range_start + range_length)

The following source code could be the implementation of that problem in C:

```
struct range {int range_start; int range_len;};

typedef int value;

typedef enum {no, yes} result;

result is_value_in_range (struct range r1, value v1)
{
   if (v1 < r1.range_start)
        return no;

   if (v1 > r1.range_start + r1.range_len)
        return no;

   return yes;
}
```

Fig. 1 A possible implementation in C according to the elementary problem specification

Taking a closer look at the C function is_value_in_range(), here the test object. Using the value 5 for range_start and the value 2 for range_len, the following table gives the expected result, depending on the input value v1.

Example: range_start = 5, range_len = 2	
Input value (v1)	**Expected result**
4	no
5	yes
6	yes
7	no
8	no

However, the implementation is (intentionally) erroneous: v1 == 7 results in “yes“ instead of “no".

3.2. Interface Analysis

Tessy analyses the source code of the function to be tested and determines the number and type of the variables of the test object's interface plus whether they are input, output, or both.

Tessy's internal Test Interface Editor (TIE) graphically displays information about the data direction flow, which can, if necessary, be easily modified or completed by the user.

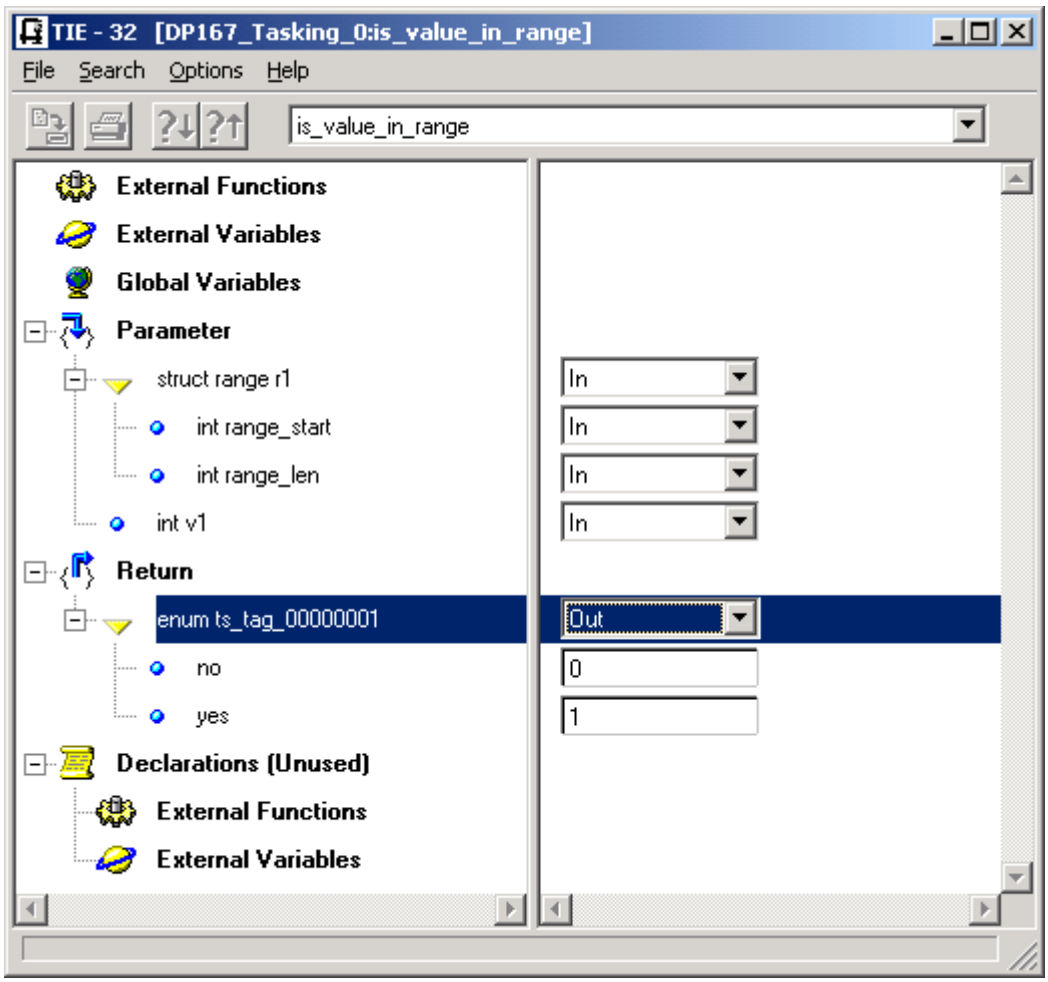

Fig. 2 The Test Interface Editor displays the findings of Tessy about the direction of the data flow for each variable of the function's interface

3.3. Test Driver Generation

From the information gained from the interface analysis, Tessy then automatically generates additional source code - the so-called "test driver". The test driver consists of the startup code for the microcontroller in question and code to call the function to be tested (the test object). Furthermore, the test driver may contain user-supplied source code or data e.g. for characteristic curves, sampled data from real observations and so on. The combination of these forms the test application, which is compiled and linked automatically by Tessy. The test driver also supplies the main() function of the test application.

3.4. Test Data Acquisition

A test case consists of the input values and the expected output values (results). These values can now be specified in Tessy using its built-in Test Data Editor (TDE). The TDE graphically displays the interface of the function to be tested and it is able to expand complex interface elements such as structures to elementary data types. Both input and output values are automatically stored in a database.

Since the inputting of test data is normally a major task for the user, Tessy's TDE allows this to be accomplished with particular ease and efficiency: e.g. it is possible to cut/copy&paste values and even test cases as a whole, pre-assign variables with values and conduct searches for variable names.

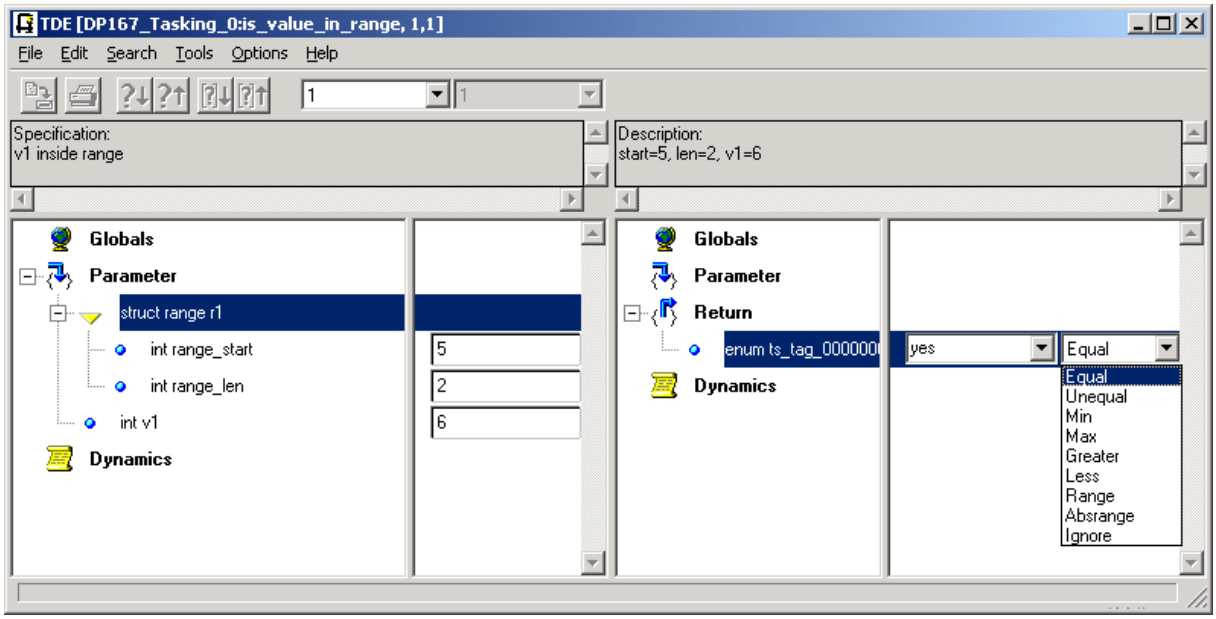

Fig. 3 Test data acquisition using the TDE

Also, the TDE lets you select the relationship to be evaluated between a result and its expected value (e.g. equal to, greater than, less than and others) in order to determine if a test was successful or not.

3.5. Automatic Results Prediction/Calculation

However, it is not mandatory for the user to pre-calculate expected test results and manually enter them into Tessy before a test case can be run. Tessy is able to run a test case without having the expected test results specified. After such a test case is run, the user can easily take the actual results of this run as expected results for subsequent runs. This relieves the user of manually determining a result and then entering it. This is particularly useful if the result is hard to determine in advance, but easy to validate if presented, e.g. consider a sorting function, which input is a large amount of unsorted data and which output is expected to be the same data, but sorted. It is much easier to simply check the sorting afterwards than to determine it manually in advance and enter it.

3.6. Running the Unit Test

To run the test, Tessy loads the test object via a debugger into the test system, e.g. an in-circuit emulator or simulator. Tessy then executes all test cases in sequence on the test system. For each test case executed, Tessy checks if the actual result corresponds to the expected result.

Test values are extracted from the database one by one - they are not included in the test application. Therefore the test application can be run even on 8-bit microcontrollers. Furthermore, the size of the test application is independent of the number of test cases. Hence in principle the number of test cases is unlimited.

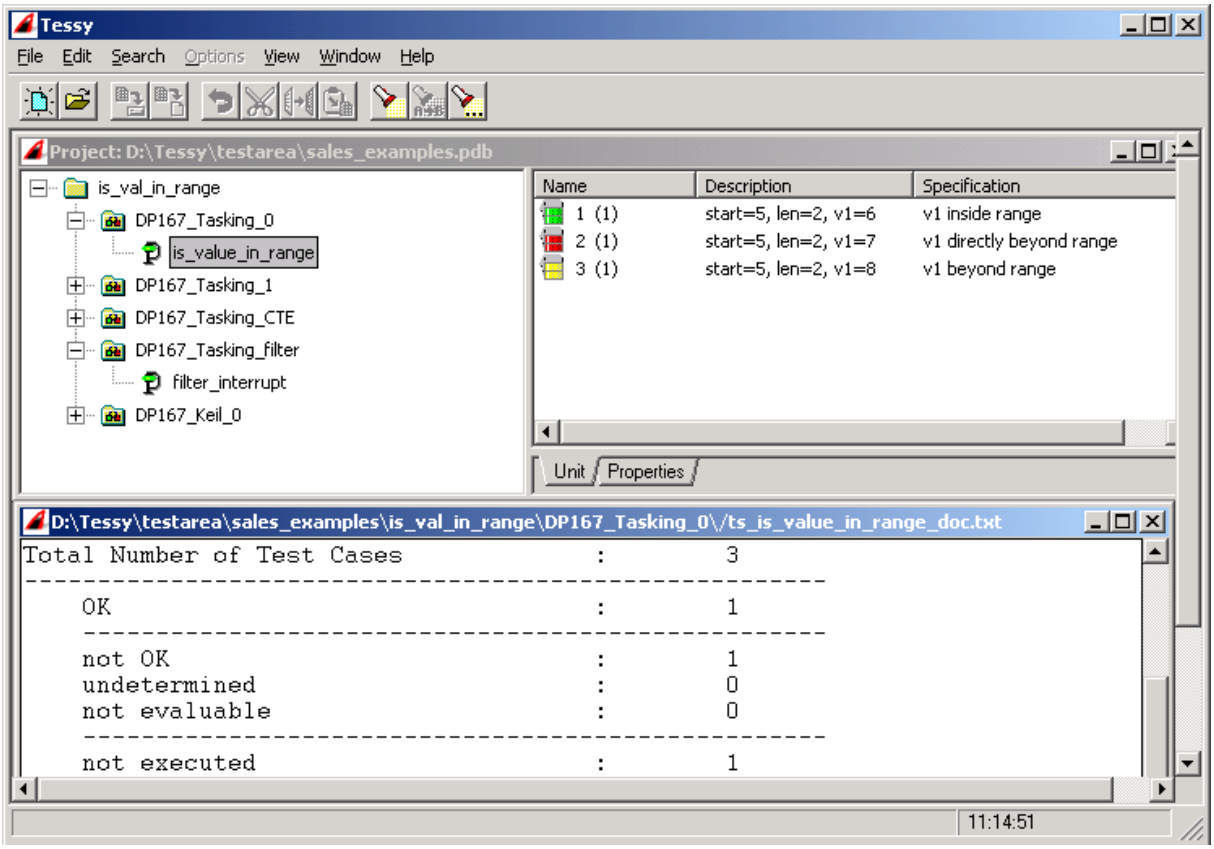

Fig. 4 Test evaluation and documentation

3.7. Test Evaluation

Tessy uses a set of colours to show whether a test case delivered the expected result (green) or not (red). Not yet executed test cases are marked yellow. Where a test result value differs from its expected value, it is displayed in the Test Data Editor (TDE) and documented in the test report.

3.8. Debugging

If executing a test case with a particular set of input values did not yield the expected result, the cause(s) of the failure needs to be determined. The tight integration of Tessy with the HiTOP debugger enables Tessy to re-run the test case, after Tessy sets a breakpoint at the entry point of the function to be tested. Test case execution is then interrupted automatically and control is passed to the debugger.

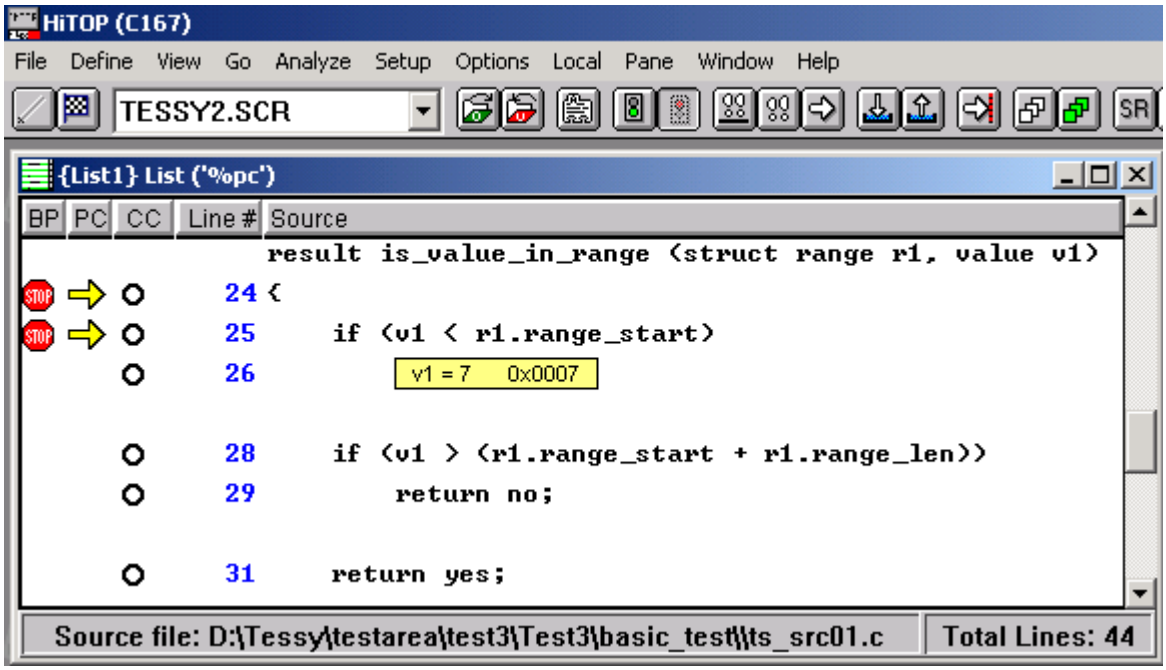

Fig. 5 Debugging starts just at the beginning of the function under test, with the parameters that caused the unexpected result

As the function is called with the exact data that caused the unexpected result to occur, the error debugging process can begin immediately. In the example above, two line steps in HiTOP quickly reveal that a missing ´=´ in line #28 caused the wrong result. As soon as the error is found, Tessy's internal editor can be used to correct the function's code and all tests can easily be re-run with the newly-modified test object.

3.9. Test Documentation

Tessy generates conclusive reports on test case results in user-definable levels of detail. Of course these reports also indicate if the execution of a test case yielded the expected result or not.

3.10. Handling Externals

What happens if the test object uses an external variable or an external function?

In principle, there are two possibilities:

1. An implementation of the external variable or function already exists in another source module: If this module can be compiled, it can be linked to the test application. In this case, the external variable or function can be considered like implemented in the module of the test object.
2. If the external variable or function does not yet exist, Tessy will take over and define them, as follows in the next section.

3.10.1. External Variables

Taking the latter case first where there is an external variable for which no implementation exists.

The variable "adj" is taken as example for such an external variable.

```
struct range {int range_start; int range_len;};

typedef int value;

typedef enum {no, yes} result;

extern unsigned int adj;

result is_value_in_range (struct range r1, value v1)
{
   if (v1 < adj + r1.range_start)
       return no;

   if (v1 > (adj + r1.range_start + r1.range_len))
       return no;

   return yes;
}
```

Fig. 6 The interface of is_value_in_range() extended by an external variable "adj"

Tessy recognizes that the interface of the test object was extended by an external variable "adj" and displays this variable in the Test Interface Editor (TIE), in the section "External variables". The context menu of the TIE allows the user to direct Tessy to define the external variable, i.e. to allocate memory for it. This will prevent the linker error message "undefined external" for "adj". The necessary source code is included in the test driver automatically by Tessy.

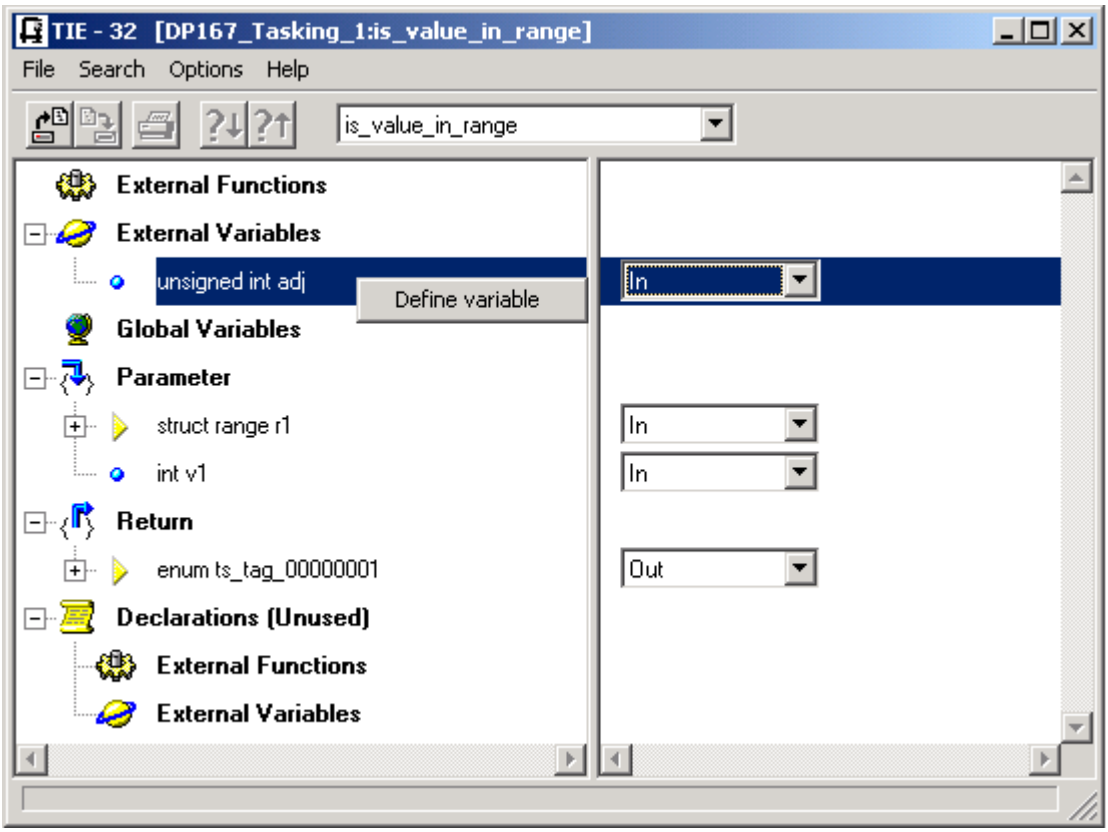

Fig. 7 Tessy can allocate memory for external variables

A variable defined by Tessy is marked by a red check mark in the TIE. Of course, Tessy lets this decision to be reversed: a previously-defined variable can be undefined also by the context menu of the TIE.

3.10.2. External Functions

A similar situation arises if the test object calls another function for which no implementation yet exists.

The following source code calls the external function absolute(), which serves as an example in the following.

```
typedef int value;
typedef enum {no, yes} result;
extern unsigned int adj;

extern unsigned int absolute(int);

result is_value_in_range (struct range r1, value v1)
{
   r1.range_len = absolute(r1.range_len);

   if (v1 < adj + r1.range_start)
       return no;
   if (v1 > (adj + r1.range_start + r1.range_len))
       return no;
   return yes;
}
```

Fig. 8 The interface of is_value_in_range() extended by an external function absolute(), for which Tessy is able to create a stub function

Again, Tessy recognizes that the interface of the test object was extended, this time by the call to the external function absolute(). This leads to the listing of the function absolute() in the section "External functions" of the Test Interface Editor (TIE).

Tessy is again able to provide a replacement function (a so-called "stub" function) for the missing implementation of the external function. Tessy can be directed to do so by using the context menu of the TIE.

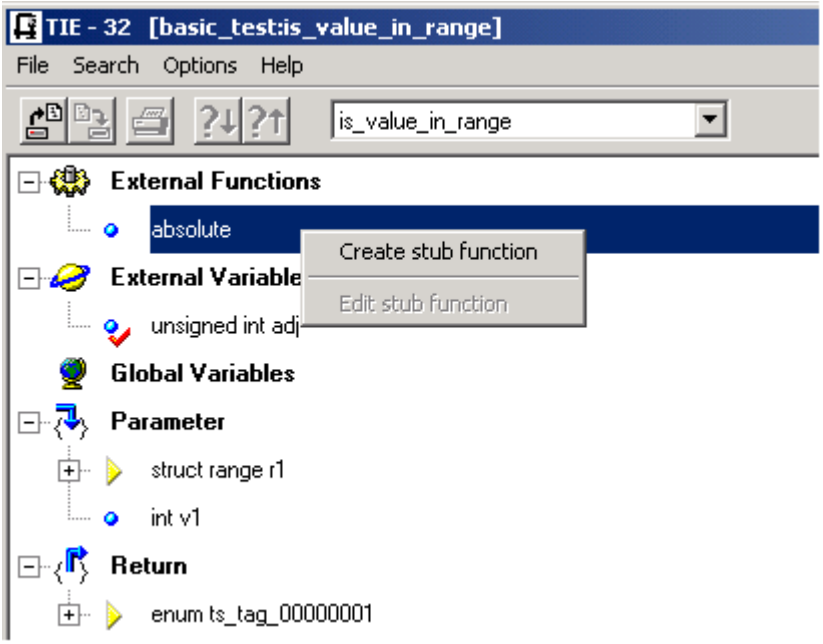

Fig. 9 Tessy can create stub functions for missing external functions - note the red check mark denoting that Tessy is allocating memory for the external variable "adj".

However, the situation is not as easy as with external variables because Tessy cannot guess what implementation the stub function should have. However Tessy allows the user to provide the source code for the body of the stub function. The stub function editor is opened by the context menu of the TIE.

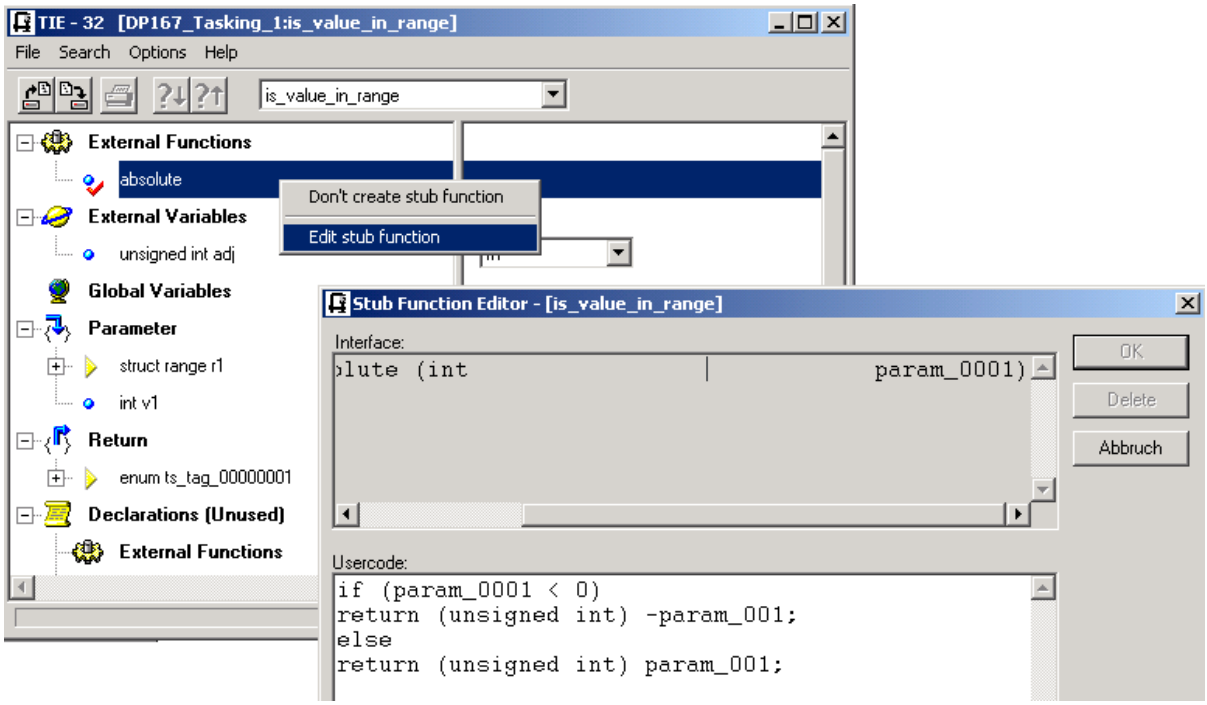

Fig. 10 The stub function editor allows the user to provide source code for the stubs

The source code provided by the user is automatically included by Tessy in the test driver. Tessy also takes care of the management of the source code for the whole group of stub functions that may exist. The user does need to worry about file names or linking issues - the user can always access and change the source code of a certain stub function using Tessy.

It should be noted that stub functions are only intended to serve as a temporary replacement for a function until the "real" implementation becomes available. Therefore the source code often consists only of a few lines of code, leading to a "reasonable" behaviour of the function (e.g. a stub function may return a predefined value instead of actually calculating one; or a stub function may just indicate that everything is in order but in fact does not perform any check.)

4. Regression Testing with Tessy

Regression tests provide verification that modifications and enhancements made to a program do not lead to undesired effects. Regression tests are thus re-runs of successfully completed test cases. If some cases did not yield a positive result and subsequent corrections of the source code have been implemented, regression tests ensure that these modifications have no undesired effects on test cases that were executed successfully. It is necessary not only to re-run failed test cases but *all* test cases. Tessy provides a batch feature that enables extensive regression testing to take place without any user intervention.

4.1. Test Case Adaptation

A situation could arise where further development of the program has resulted in the function under test being so modified that its test cases are no longer executable (e.g. due the introduction of an additional parameter). Since Tessy has stored information on existing interfaces, it is able to recognize alterations made to an interface and refers the user to them. Tessy's internal Interface Data Assign editor (IDA) allows efficient assignment of newly introduced interface elements to the elements of an existing interface. The IDA performs this allocation automatically wherever possible. This feature allows continued use of values from old interface elements in new ones. This is possible thanks to the concept of separating interface information from test data.

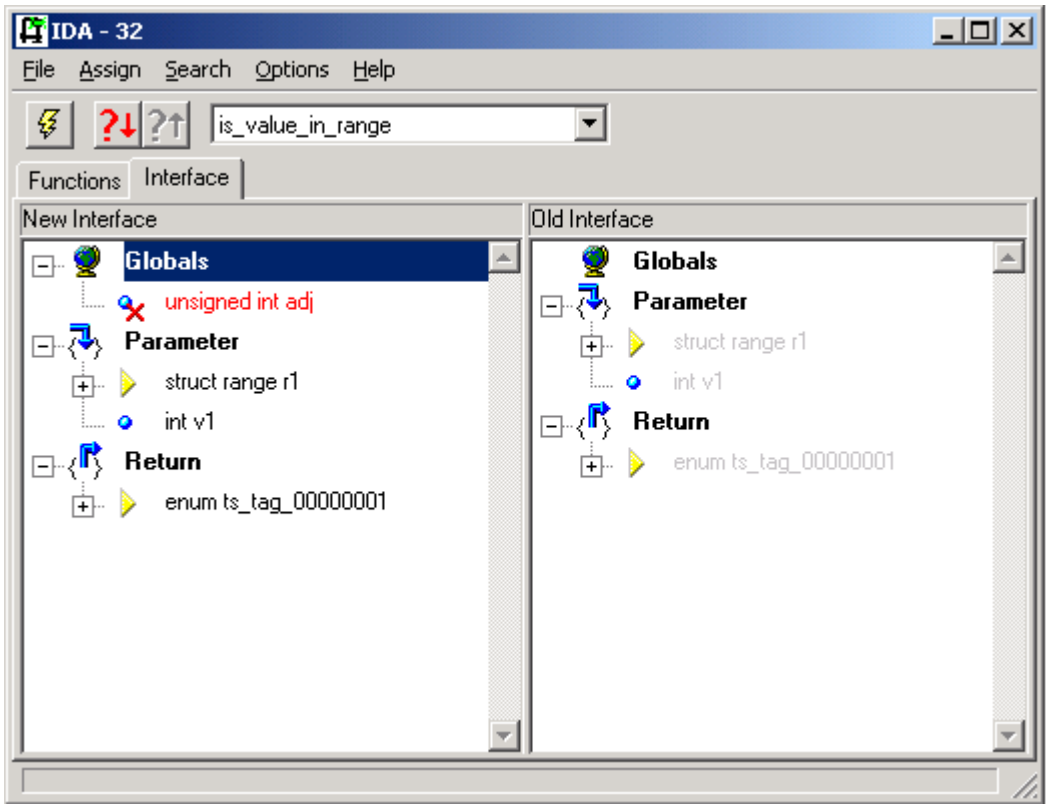

Fig. 11 Allocation of interface elements

4.2. Further Use of Test Data

The pre-requisite for ensuring test cases are compatible with the latest version of the software is that test data, including data for modified interfaces, is reusable in the most efficient and extensive means possible. The re-use of test data allows the repetition of tests at any time and re-running tests implies regression testing, this being an essential procedure in assuring software quality.

Regression testing often is necessary for software to that is subject to subsequent modification of its code size, performance, maintenance and reusability, while at the same time guaranteeing that existing functionality is preserved.

5. Test Coverage with Tessy

In general, the code coverage analysis reveals which portions of a program were executed. There are different kinds of coverage analysis, some of which have more value than others. Coverage analysis is based on the “white-box” approach to software testing because the analysis refers to the internal structure of the test object (e.g. the lines of a C function or the decisions in a C function). Code coverage is a measure for test coverage. Code coverage below 100% indicates that there is still something left to test.

5.1. Line Coverage

A very simple coverage measure is line coverage. This coverage measure indicates if a line of a test object was executed or not. Line coverage does not reveal how often this line was executed. Due to the simplicity of the line coverage measure, even achieving 100% line coverage guarantees neither zero-defect code nor sufficient testing. This simple example illustrates the point:

```
#1   int a = 0;
#2   if (cond)
#3   {
#4         a = 1;
#5   }
#6   a = 1 / a;
```

Fig. 12 Simple line coverage does not reveal the problem

Even if all lines were marked as executed one or more times, it is obviously possible that no test run was made in which the condition "cond" evaluated to 0, which in turn would prevent line #4 from being executed (i.e. the non-existent then clause would have been executed), which in turn would have caused a division-by-zero error in line #6.

To sum up, line coverage is a very weak measure. However, if 100% line coverage has not been achieved, it is strongly recommended to find out what is the cause. One reason may be "dead" code, which cannot be executed under any circumstances. Such code can be deleted. Furthermore, you may find code that was not exercised by any of the test cases, hence you have insufficient or the wrong test cases. In any case, the test coverage has been shown to be inadequate and that should prevent the testing from being completed.

Although line coverage is a weak measure, it is better than no measure at all and achieving 90% line coverage is always better than achieving only 80%.

Statement coverage is similar to line coverage, but covers assembler statements instead of C language lines. An alternative name for line / statement coverage is “C0” (C-zero) code coverage.

5.2. Path Coverage

The coverage analysis built-into Tessy does more than simple line coverage. It not only reveals whether a line was executed or not, but also how often a line was executed. This information is sufficient to state if a path through the code was taken or not. This kind of coverage is also known as “C1” code coverage.

Tessy's coverage analysis is activated by a simple click of the mouse. Tessy allows you to distinguish whether only the test object is of interest for the code coverage analysis or if the functions that are called by the test object should be included in the analysis as well.

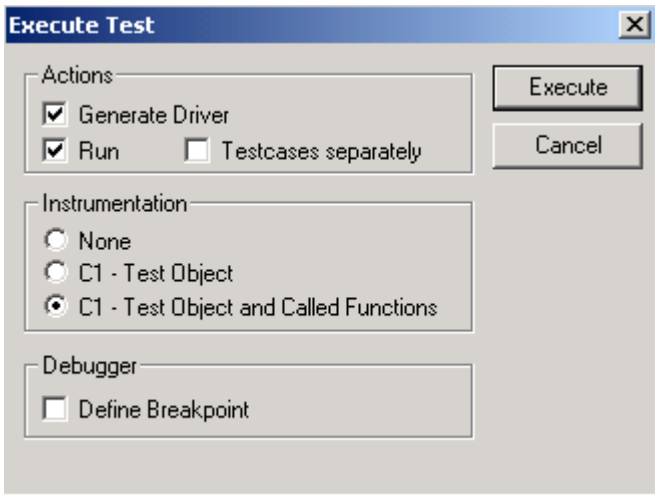

Fig. 13 Activating Tessy's C1 code coverage analysis

Tessy then automatically instruments the test object and executes the tests as usual. After that, the results of the coverage analysis are presented in a special report window.

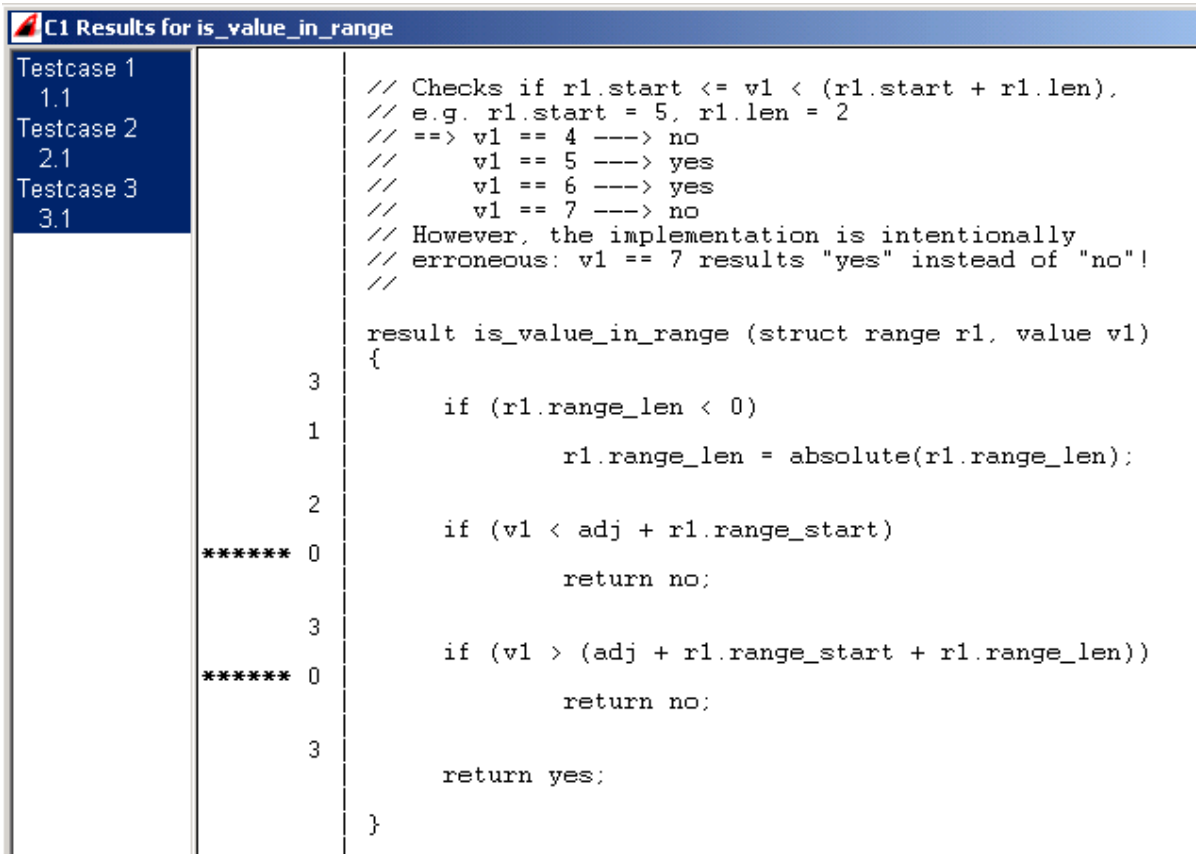

Fig. 14 C1 Coverage results

This report shows clearly, that for example, if the first “if”-statement was executed 3 times but part of it was executed only once, hence the condition of the “if” evaluated twice to false, which in turn means that the non-existent “else” part of the if-statement was exercised twice.

By clicking on a test case in the upper left corner of the report window, the report window shows the information for that test case only.

6. Integration Testing with Tessy

Tessy is normally regarded as a tool for doing unit tests where a “unit” is considered to be a C function. In fact, Tessy is also well-suited to integration testing of embedded software. Whereas unit tests concentrate on the test of the functionality of a single unit (i.e. a C function), integration tests concentrate on the test of the interfaces between units.

In an ideal world, integration testing starts with the unit test of a function “f1” that does not call any other function. However, f1 could interact with peripherals. When the unit test for f1 is finished, f1 is considered to work correctly under all circumstances, even in case of illegal parameters, etc. The integration test could then proceed by picking up another function “f2”, for unit testing. f2 may call f1, but no other function that has not passed its unit test already.

In case f2 calls f1, the (already tested) code of f1 should be linked to the test application for the unit test of f2. This means, the unit test of f2 uses the current implementation of f1. This automatically exercises the interfaces between f1 and f2. The test cases for the unit test of f2 should be tailored so that they mainly test the functionality of f2. It is not necessary that they again test the functionality of f1. Theoretically sufficient tests for f2 will automatically lead to an adequate test of the interface of f2 to f1 - at least to the extent that this interface is used by f2.

In practice, such a strict ordering of units according to the call hierarchy may not be possible because two functions may call each other or because the order in which the functions are implemented is different from the call hierarchy. That is why Tessy provides stub functions.

7. Conclusion

Tessy is a software tool that addresses unit testing, integration testing and regression testing. Also, Tessy allows the determination of the test coverage. However a part of Tessy, the Classification Tree Editor, addresses test planning and test-case specification. This is the topic of another paper.