TESSY
–
Yet Another Computer-Aided Software Testing Tool?

Joachim Wegener
Roman Pitschinetz
Klaus Grimm
Matthias Grochtmann
Daimler-Benz AG
Forschung und Technik
Alt-Moabit 91 b
D-10559 Berlin, Germany
Tel: +49 (0)30 39982-232 / -229 / -231 / -226
Fax: +49 (0)30 39982-107
email: wegener@dbresearch-berlin.de

Abstract

Until now, a lot of computer-aided software testing tools have been developed. Most of them aim at the special support of several distinct test activities like test execution, monitoring, and test evaluation. Tools which cover a wider range of activities and test phases have the decisive shortcoming that there is no methodological support for the systematic design of test cases, particularly for the functional test. In summary, there is no sufficient overall support for all test activities.

In order to overcome these shortcomings, the testing tool TESSY was developed. The most important strength of TESSY is that it provides support for the whole testing life cycle, with the main emphasis on test case determination, test data generation, test execution, and test evaluation. Furthermore, there is powerful support for test organization and test documentation. Substantial components of TESSY are, for example, special window-based editors for the provision of essential environment information, for the graphical visualization of the test object's interface as well as for the interactive input of test data and expected results. During test execution, several basic activities, such as test driver generation, coverage analysis, and regression testing, are performed automatically. For the test evaluation, different criteria can be defined for the comparison of actual and expected results.

However, the essential highlight of TESSY is given by the computer-aided test case design by means of the classification-tree editor CTE which was already presented at the EuroSTAR '93 conference. CTE supports the descriptive and systematic design of black-box test cases following the classification-tree method and has become an integral part of TESSY.

The first version of TESSY was finished in July 1994. Practical trials of the system will start in the fourth quarter of this year. Based on the promising experience with CTE, further improvement, particularly of the test efficiency, can be expected. Future work will focus on extensions of TESSY with respect to integration testing and automatic generation of test cases and test data.

1. Introduction

Dynamic testing is of major practical importance for the quality assurance of software systems. Up to 50 % of the overall development cost is incurred in software testing (MYERS and DAVIS). Investigations of various software development projects in several divisions of the Daimler-Benz Group showed that the costs for software testing mostly arose for unit testing, integration testing, and system testing. On average, 34 % of total testing expenses is spent on unit testing, 28 % on integration testing and approximately 27 % on system testing. The remaining 11 % is spent for specific tests like the examination of software-hardware interfaces.

A result of these statistics is that in practice up to 17 % of the overall development cost is allotted to unit testing. This significant quota of the entire development cost increases if integration testing is done bottom up and the integration is checked by testing higher level units. In this case the overall cost for unit and integration testing increases to an amount of 31 % of the entire development cost. This illustrates the importance of unit testing. Significant savings can be achieved, and the product quality can be improved by tools automating unit and integration testing. Moreover, system programmers will examine their units more carefully if they have powerful tools for efficient testing.

This paper describes the computer-aided software testing tool TESSY which provides general support for all test activities needed for unit testing. As an essential part of TESSY, the classification-tree editor CTE, presented at last year's EuroSTAR (GROC1), aids the tester in the systematic determination of test cases.

The following chapter contains a description of test activities required for unit testing. Chapter three gives a short overview of computer-aided software testing tools that are suited for unit testing. Afterwards the test system TESSY and its usage are described by testing an example module of TESSY itself. After some concluding remarks the paper closes with a short outlook on future work.

2. Test Activities

A systematic test comprises the following main activities: test case determination, test data selection, expected results prediction, test case execution, monitoring, and test evaluation as well as the accompanying activities of test organization and test documentation. This structure facilitates a systematic procedure and the definition of intermittent results. Figure 1 shows the test activities mentioned above and the relationships between them.

The *test organization* includes all activities involved in the management of test objects and their appertaining data. Therefore, a test database is required containing, for example, test cases, test data, expected results, the test object's output values, and technical settings. Furthermore, technical prerequisites for test case execution such as the implementation of test drivers have to be organized. Another task is to ensure the reproducibility of tests to support regression testing after program changes.

In the course of *test case determination*, the test cases, with which the test object should be tested, are defined. A test case defines a certain input situation to be tested.

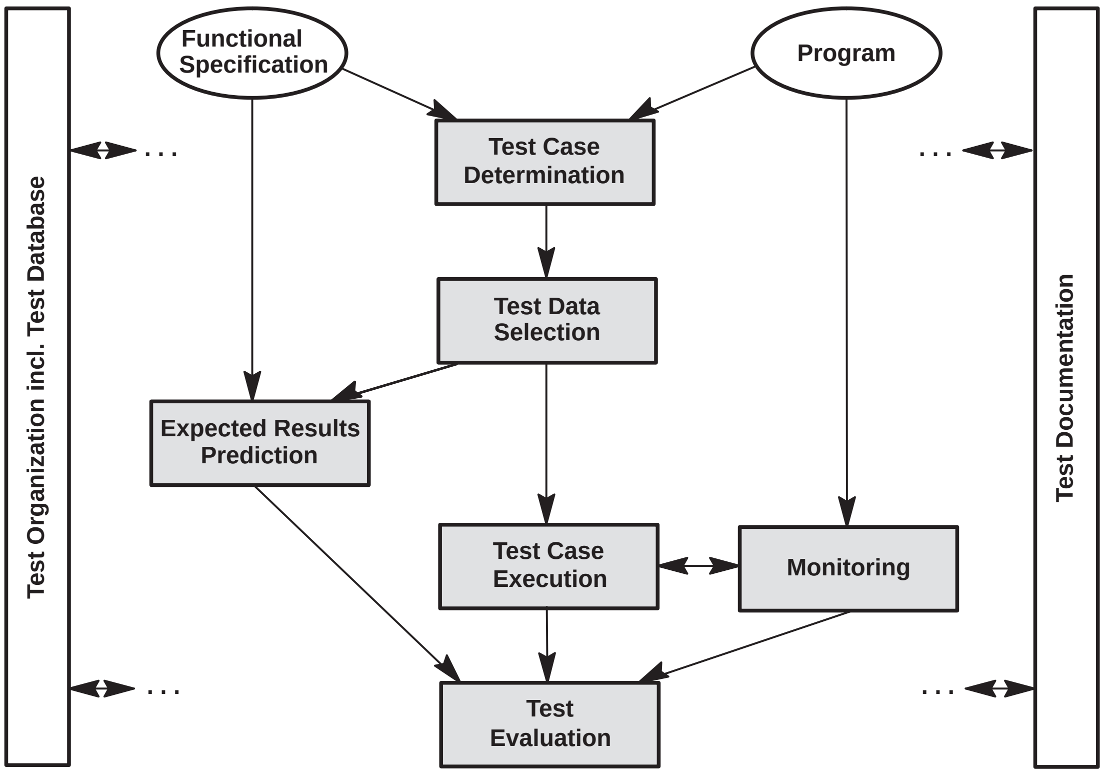


Figure 1: Test activities

It comprises a set of input data from the test object’s input domain. The test case determination is the most important activity for a thorough test, since it determines the kind and scope of the examination and thus the quality of the test.

A test case abstracts from a concrete test datum and defines it, only in so far as it is required for the intended test. During *test data selection* the tester has to choose a concrete element from each test case, i. e. from each set of input values, with which the test should be executed.

Determining the anticipated results and program behaviour for every selected test datum constitutes *expected results prediction*. If it is not possible to specify unequivocal output values or the expected behaviour, acceptance criteria or reference data have to be used for the predicition of expected results.

Subsequently, *test case execution* is performed. The test object is run with the selected test data. The output values and the program behaviour are thus determined. By implementing an appropriate test driver, extensive automation of test case execution is possible.

The behaviour of the test object can be observed and recorded during test case execution by means of *monitoring*. A common method is to instrument the program code according to a white-box test criterion. The source code is extended for that purpose by inserting statements at control-flow or data-flow relevant points of the program, which count the number of executions of the according program parts.

In the course of *test evaluation*, actual and expected values as well as actual and expected program behaviour are compared, and thus the test results are determined.

Finally, the test should be evaluated by comparing the test results achieved with the test objectives aspired to.

The objective of the *test documentation* is a clear, understandable, and comprehensive description of the test results and all other information produced during the test. Deviations between expected and actual behaviour have to be documented, discovered errors are to be summarized in error statistics, and the fulfillment of the test objectives should be assessed.

3. Tools for Unit Testing

About 30 tools for the automation of software tests were presented at the testing tools exhibitions of the EuroSTAR ’93 and STAR ’94 conferences. The largest fraction of tools were capture-and-replay tools, in particular supporting system testing of interactive programs. Approximately ten tools seem to be applicable for an employment in the field of unit testing, as there are Cantata and LDRA Testbed. A detailed study on these ten tools, examining their abilities to support the test activities mentioned above, reveals that none of the tools offers a general support for all test activities. Most of the unit testing tools concentrate on monitoring, test organization, and test documentation. Two software testing tools exclusively offer support for test case determination. Table 1 contains an anonymous survey.

	Test Activities							
TOOLS	Test Organization	Test Case Determination	Test Data Selection	Expected Results Prediction	Test Case Execution	Monitoring	Test Evaluation	Test Documentation
A						▨		
B		▨						
C	▨							▨
D	▨					▨		▨
E	▨					▨		▨
F						▨		▨
G	▨							▨
H	▨				▨	▨	░	▨
I						▨		
J		░						
TESSY	▨	▨	▨	▨	▨	▨	▨	▨

Table 1: Unit testing tools presented at EuroSTAR ’93 and STAR ’94

Due to the lack of software testing tools offering an overall support for all test activities, Daimler-Benz Research in Berlin developed the test system TESSY which provides general support for all test activities and aims at a considerable increase in unit testing efficiency for programs written in C. The costs for software testing are to be reduced, and the reliability of systematic and thoroughly tested software products is to be increased by using TESSY.

4. Test System TESSY

The most important strength of TESSY is that it provides support for the whole testing life cycle, offering a homogenous, object-oriented, and context-sensitive user interface which guides the tester through all the test phases. Accordingly, the use of TESSY is very simple and comfortable. TESSY contains separate tools for each test activity which are described successively in the following sections.

TESSY facilitates a combination of black-box and white-box tests. The emphasis is laid on black-box testing because only test cases derived from the functional specification allow to examine appropriately, whether all specified requirements have been transformed into the test object. Test cases are determined using the classification-tree editor CTE (GROC1), which is based on the classification-tree method (GROC2). During the functional test, the branch coverage can be investigated by means of instrumentation. If the degree of coverage is not sufficient to reach the test objectives, the test has to be improved subsequently by additional test cases.

Through the extensive test organization assistance provided by TESSY, comprising an integrated database for all test relevant data, regression testing can be totally automated in most cases. After the tester has edited the required data for the first test execution the data are available for further tests at any time. Normally, the test run simply has to be repeated.

The test system TESSY was developed on VAXstations with the operating system VMS for testing ANSI-C and VAX-C programs. It is implemented in C. The user interface is based on OSF’s Motif toolkit. A version for SUN/Solaris is planned.

4.1. Example

The application of TESSY shall be explained using a realistic example: The test object is the C-function

is_line_covered_by_rectangle (struct line line, struct rectangle rectangle)

which checks whether a line is covered by a given rectangle with its sides parallel to the axes of the coordinate system. Input parameters of the test object are two structures. The first one of type *line* describes the line by the positions of its two end points, the second one of type *rectangle* describes the rectangle using the position of its upper left corner, its width and its height. If the line is covered by the rectangle, the test object should return the value *yes*, otherwise *no*. Figure 2 illustrates the task of the test object *is_line_covered_by_rectangle* by means of an arbitrary rectangle and several sample lines. The figure also defines regions to describe the possible positions of the line end points with respect to the rectangle.

The function *is_line_covered_by_rectangle* is part of the classification-tree editor CTE. It is used to determine the need of redrawing connecting lines between the ele-

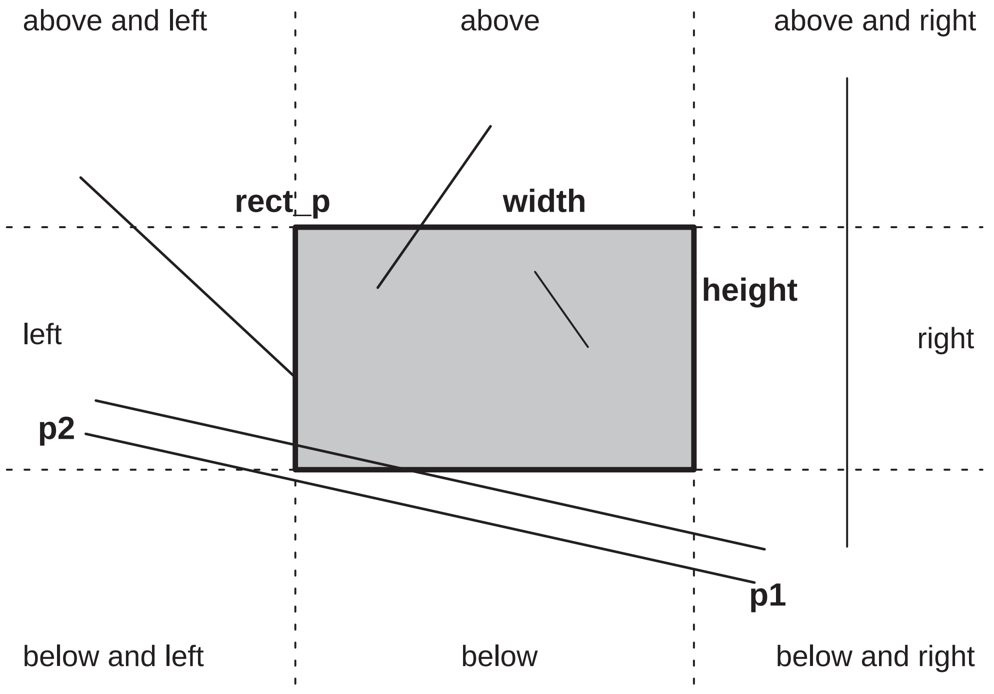


Figure 2: Rectangle with sample lines

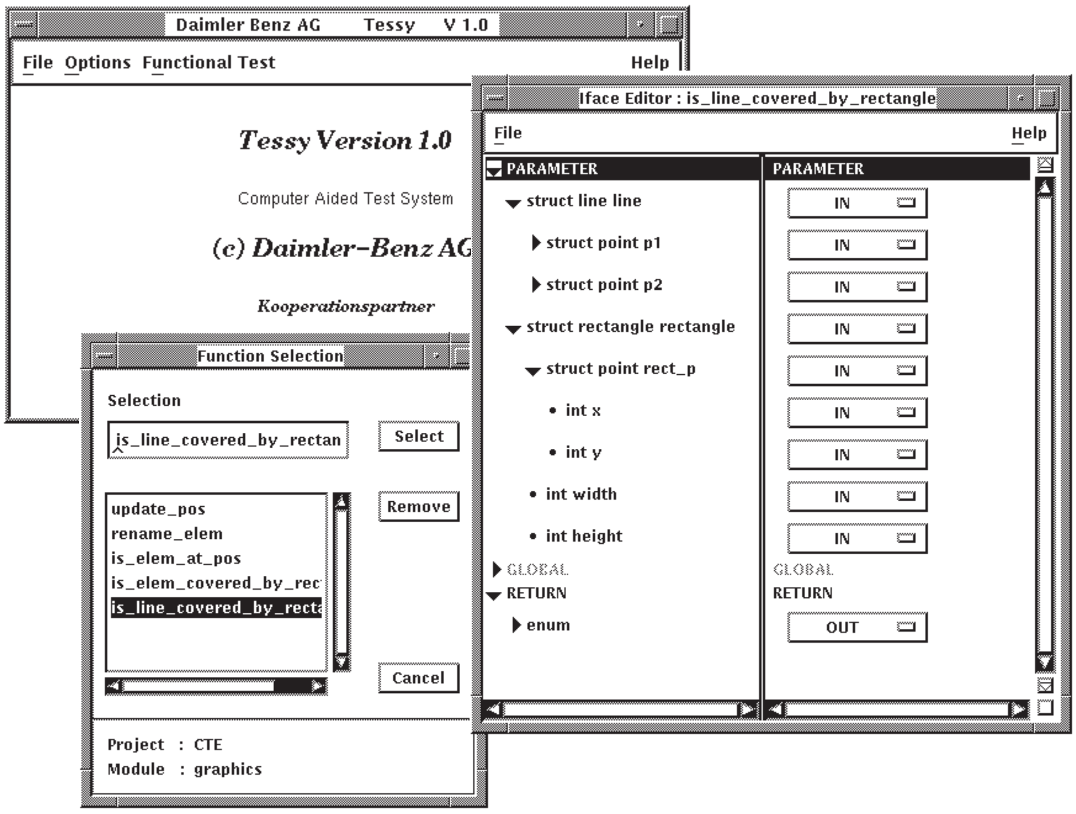
ments of the classification-tree in case of window exposure events. Similar functions can be found in many other software systems dealing with graphical presentations.

4.2. Test Organization

In TESSY a test is organized into projects. Each project contains at least one logical module. In preparation of the test, the user has to supply some information for each module using the *environment editor.* Among other data he enters the program sources for the logical module, the compiler to be used and, if necessary, compiler options and linker instructions. In case all information is complete, TESSY investigates the whole export interface of the module automatically by analyzing all program sources stated. The export interface consists of all exported C-functions and their interface, including global variables, parameters, function return codes and the corresponding data types. The export functions ascertained represent the actual test objects.

Before the tester can start with the central test activities for each function, he has to complete the interface description of the export functions by entering certain characteristics of each interface component for which an automatic identification was impossible. He must specify whether a component is only an input, an output, or input and output of the respective test object. Value parameters are always of the kind *IN*, the function return code always of the kind *OUT.* Global variables and dynamic values, like pointers, can be of the kind *IN*, *INOUT*, or *OUT.* This completion of the interface description is carried out in TESSY's *interface editor*, shown in Figure 3. It is possible to browse through complex data types down to the level of basic C data types. Figure 3 illustrates this for the second parameter *struct rectangle* of the example.

The type of every parameter of the illustrated test object *is_line_covered_by_rectangle* can be determined automatically. Both structures, *line* and *rectangle*, as well as



Selection : is_line_covered_by_rectan

Project : CTE
Module : graphics

struct line line	IN
struct point p1	IN
struct point p2	IN
struct rectangle rectangle	IN
struct point rect_p	IN
int x	IN
int y	IN
int width	IN
int height	IN
enum	OUT

Figure 3: Editor with interface description of *is_line_covered_by_rectangle*

their elements, are value parameters. Consequently, they are labelled *IN*. Naturally, the function return code is of the type *OUT*. Global variables do not exist.

When the definition of all interface components has been completed, the tester is able to begin with the main test activities.

4.3. Test Case Determination

The test case determination is carried out by means of the graphical classification-tree editor CTE which supports the application of the classification-tree method. The user edits the classification tree in a syntax-directed and object-oriented manner. In the course of the classification-tree method, the input domain of the test object is regarded under various aspects assessed as relevant for the test. For each aspect disjoint and complete classifications are formed. Classes resulting from these classifications may be further classified. The stepwise partition of the input domain results in the classification tree. Subsequently, test cases are determined by combining classes of different classifications. This is done by using the tree for the generation of a combination table in which the test cases are marked interactively. The CTE transmits these test cases to the test system TESSY where they are saved in the test database.

The classification-tree method was described by Grochtmann and Grimm (GROC2), the classification-tree editor CTE was already presented at the EuroSTAR'93 (GROC1). Therefore, they will not be explained in more detail here.

The test case determination for the example *is_line_covered_by_rectangle* leads to 49 test cases. Figure 4 shows the main window of the CTE displaying the appertaining classification tree and one part of the combination table. Test case number two, for instance, defines a test with the special case that the line end point P1 is the only point of the line which is covered by the given rectangle. P1 is located on the left side of the rectangle. The position of the line end point P2 is left from and above the rectangle. A sample line is shown in Figure 2.

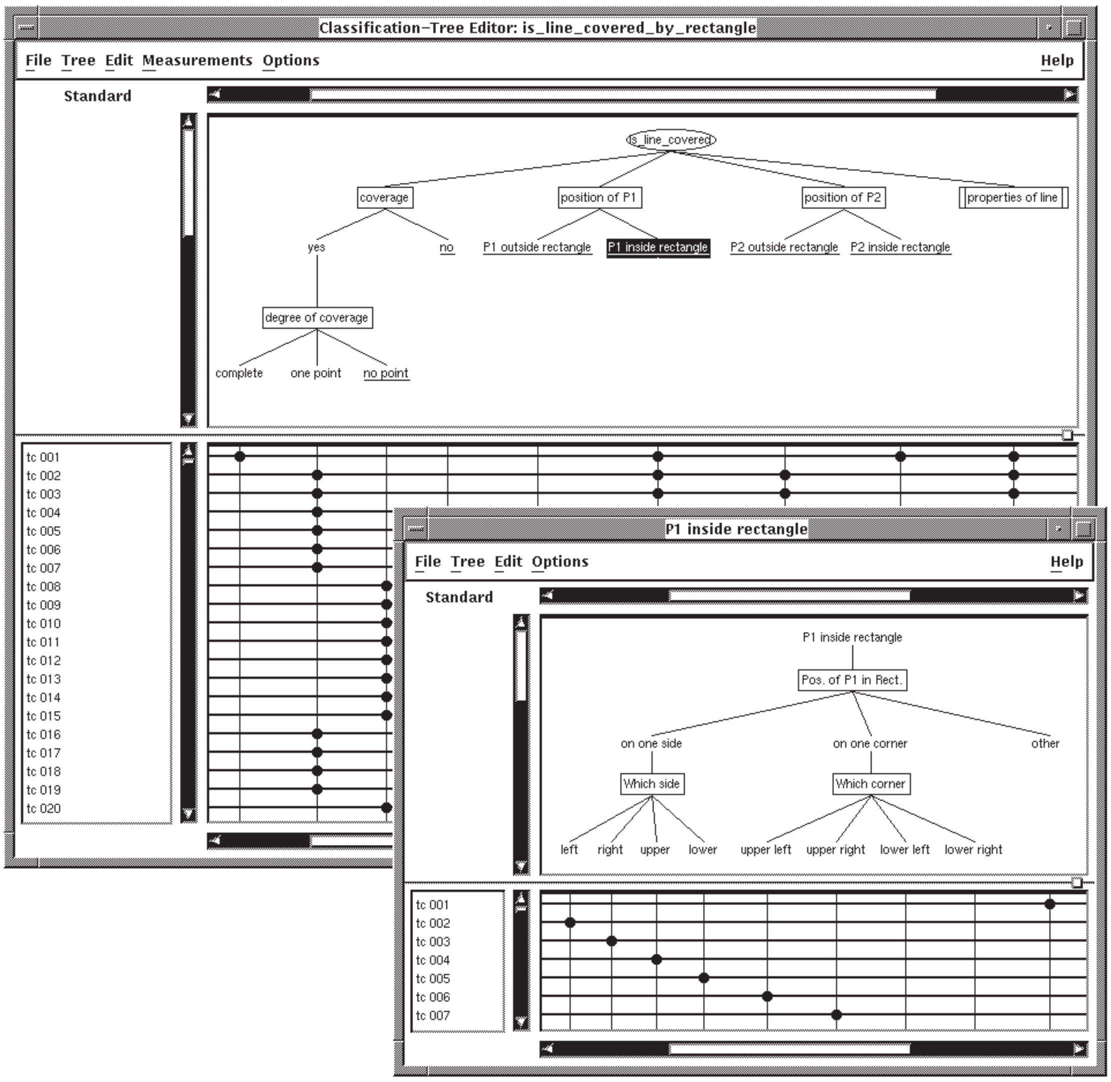
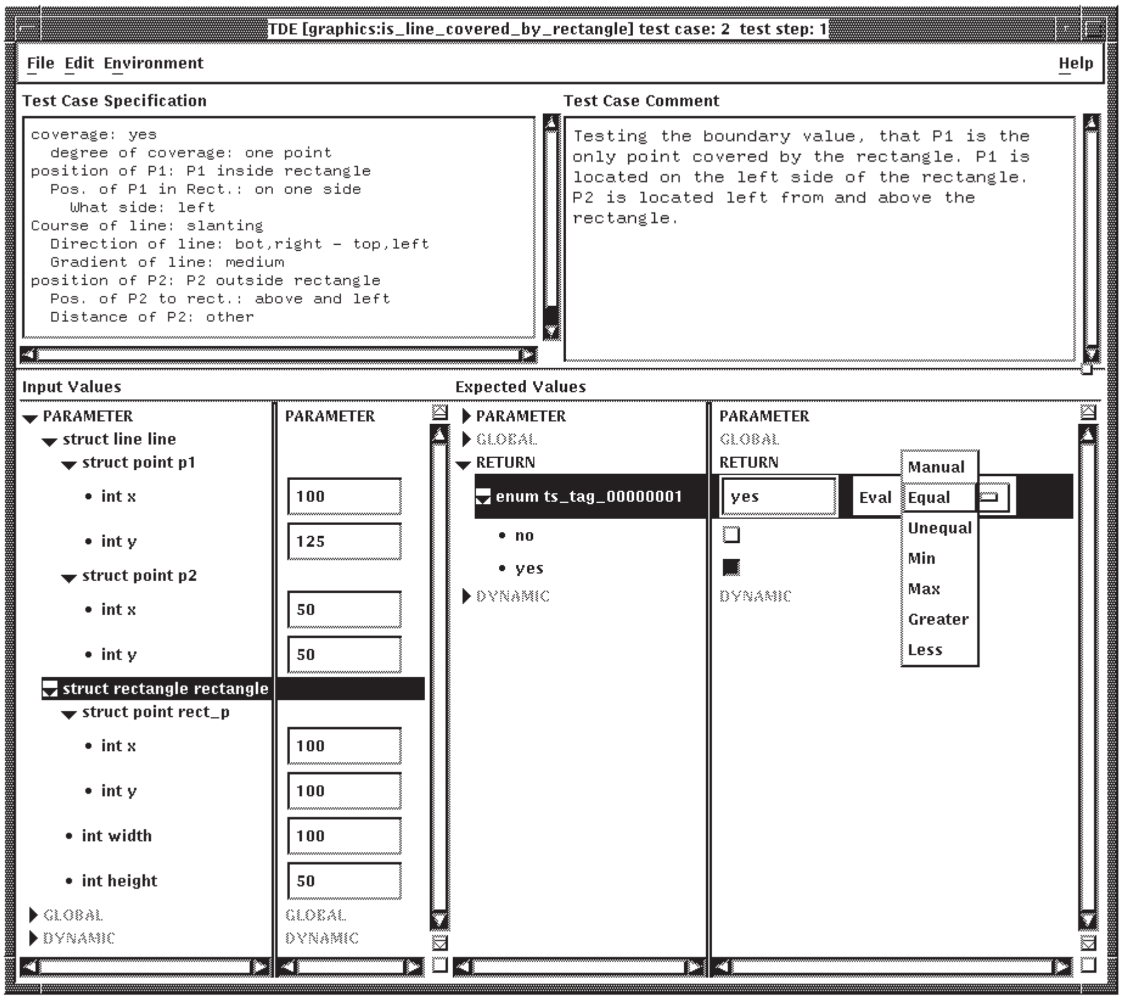


Figure 4: Test case determination for *is_line_covered_by_rectangle* using the CTE

4.4. Test Data Selection and Expected Results Prediction

Subsequent to the test case determination, the user calls the test-data editor (TDE) to enter concrete test data and to predict expected results for each defined test case. Based on the test case definition in the combination table of the CTE, a text version of the test case actually to be worked on is generated and displayed in the upper part of the editor. Additional test case comments are shown as well. Below these text areas the whole test object interface is presented in two browsers, one for editing test data, the other to enter expected results for the output parameters of the test object. Each component can be browsed through down to the level of basic types where input areas for editing concrete test data and expected results are provided. In addition, the tester has to select an evaluation mode for each output parameter, which determines the way the actual values produced by the test object shall be evaluated with respect to the expected results. Appropriate evaluation modes are offered to the user in pop-up menus depending on the particular data type handled. Evaluation criteria are, for example, equality, inequality, and lower or upper bounds. To enter pointer values the user can generate dynamic values. Thereby, it is possible to enter dynamic data structures like lists and trees as input values or predicted results.



TDE [graphics:is_line_covered_by_rectangle] test case: 2 test step: 1
File
Edit
Environment
Help
Eval
Manual
Equal
Unequal
Min
Max
Greater
Less

Test Case Specification

```
coverage: yes
  degree of coverage: one point
position of P1: P1 inside rectangle
  Pos. of P1 in Rect.: on one side
    What side: left
Course of line: slanting
  Direction of line: bot,right - top,left
  Gradient of line: medium
position of P2: P2 outside rectangle
  Pos. of P2 to rect.: above and left
  Distance of P2: other
```

Test Case Comment

```
Testing the boundary value, that P1 is the
only point covered by the rectangle. P1 is
located on the left side of the rectangle.
P2 is located left from and above the
rectangle.
```

Input Values

```
PARAMETER                          PARAMETER
  struct line line
    struct point p1
      • int x                      100
      • int y                      125
    struct point p2
      • int x                      50
      • int y                      50
  struct rectangle rectangle
    struct point rect_p
      • int x                      100
      • int y                      100
    • int width                    100
    • int height                   50
GLOBAL                             GLOBAL
DYNAMIC                            DYNAMIC
```

Expected Values

```
PARAMETER                          PARAMETER
GLOBAL                             GLOBAL
RETURN                             RETURN
  enum ts_tag_00000001             yes       Eval Equal
    • no                           ☐
    • yes                          ■
DYNAMIC                            DYNAMIC
```

Figure 5: Test data selection and expected results prediction for *is_line_covered_by_rectangle*

Figure 5 illustrates the test-data editor with its two browsers for the second test case of *is_line_covered_by_rectangle*. The tester enters test data in the left browser and expected results in the right one. The upper left corner of the rectangle is located at position (100,100). The width of the rectangle amounts to 100 pixels, its height to 50 pixels. The position of the first end point of the line, P1, is (100,125). Consequently, it is located on the left side of the rectangle as demanded in the test case specification. The second end point of the line has a position which is left from and above the rectangle. Its coordinates are (50,50). The only expected result to enter is a value for the function return code. Since the function return code is of the type enumeration, the expected results browser lists the possible values, and the user can select one value by simply clicking on the appertaining toggle button. For the actual test case, *is_line_covered_by_rectangle* has to detect a coverage of line and rectangle. Therefore, the expected output is *yes*. The evaluation mode is set to *Equal*.

4.5. Test Case Execution and Monitoring

If test data, expected results and evaluation modes for each test case were entered completely, the test object can be executed with the test data. TESSY provides a special tool for controlling the test case execution. The tester can select the test cases to be executed from a table visualizing the whole set of test cases. Additional editors are available to edit operating system-dependent settings, like the definition of logicals, or to prepare the test object's environment on programming language level, for instance, in order to open a file required by the test object. Both the operating system-dependent settings and the preparations on programming language level often are a prerequisite for a proper test execution. Based on the data in the test database and the information given by the tester, TESSY generates a test driver which runs the test object for every selected test case with its test data. This test driver also receives the actual values produced by the test object and saves them in the test system database. The generated driver concurs with ANSI-C restrictions. For test case execution, the test driver is linked to the test object and, if necessary, to other required files forming a so-called test rig. Optionally, the tester can export the test rig for test case execution on a target system. Afterwards it can be reloaded into the test system for test evaluation.

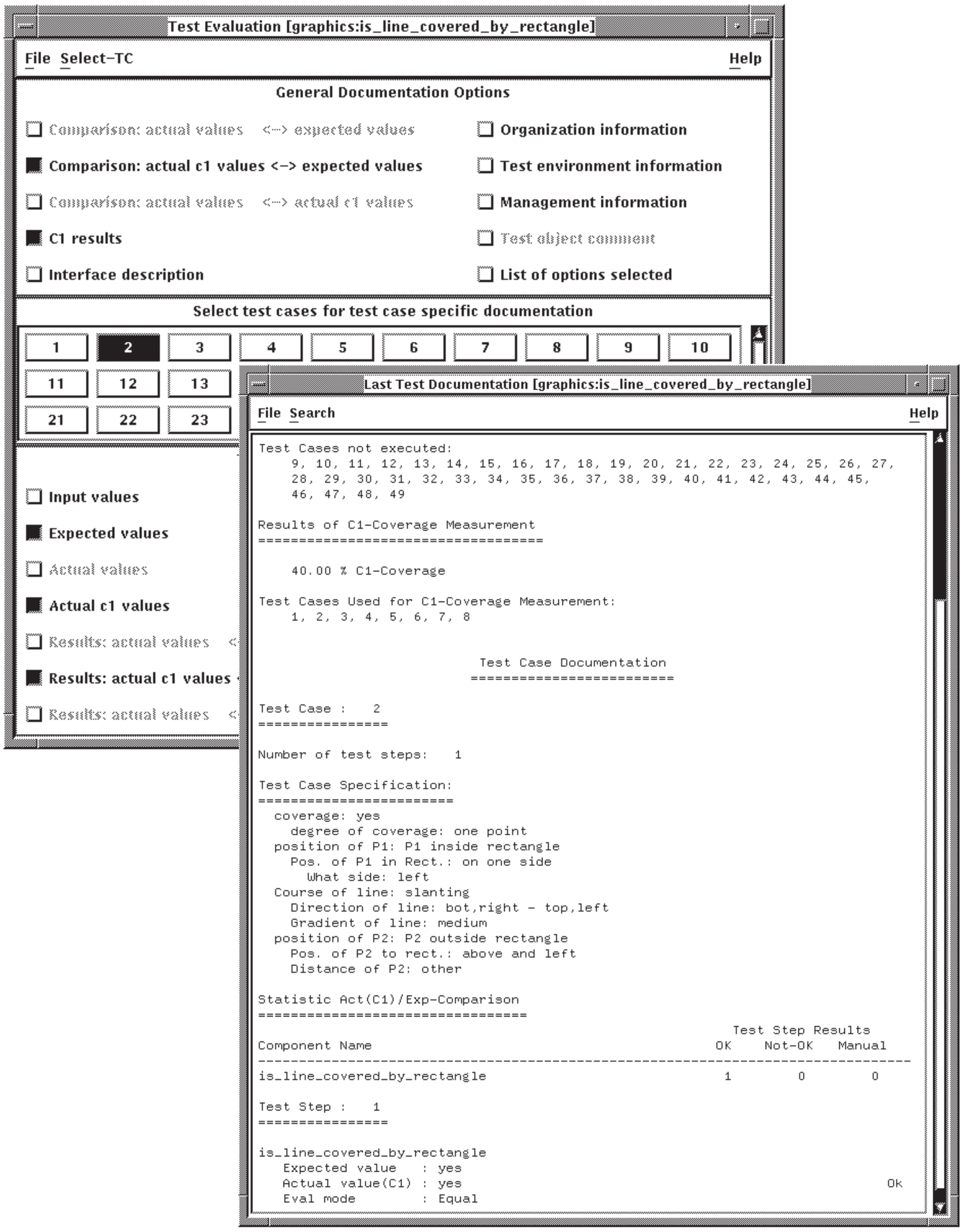
Monitoring of the test object can be done by measuring the branch coverage achieved. For this purpose, TESSY provides automatic instrumentation of the test object's source code. It is possible to instrument either only the export functions or the entire module.

4.6. Test Evaluation and Test Documentation

The last tool within TESSY is used for the test evaluation and the generation of a user determinable test documentation (Figure 6). The expected results and actual values are compared according to the evaluation modes specified. This yields the test results. If the test object was instrumented, the branch coverage can be computed and the program branches not executed can be shown.

The tester can specify the scope of the test documentation to be generated through a number of options. The documentation can vary from general error statistics for the complete test to very detailed tables for each test case executed. Generated documents can be saved in ASCII-files in order to import them into desktop publishing tools.

An exemplary test documentation for *is_line_covered_by_rectangle* is illustrated in Figure 6. The tester selected the first eight test cases for test case execution, and he activated monitoring. The execution of the first eight test cases did not discover any error in the test object, but only 40 % of the existing branches were executed. A detailed documentation is displayed for the second test case.


Test Evaluation [graphics:is_line_covered_by_rectangle]
File Select–TC Help
General Documentation Options

- ☐ Comparison: actual values <–> expected values
- ■ Comparison: actual c1 values <–> expected values
- ☐ Comparison: actual values <–> actual c1 values
- ■ C1 results
- ☐ Interface description
- ☐ Organization information
- ☐ Test environment information
- ☐ Management information
- ☐ Test object comment
- ☐ List of options selected

Select test cases for test case specific documentation
1 [✓ 2] 3 4 5 6 7 8 9 10
11 12 13
21 22 23

- ☐ Input values
- ■ Expected values
- ☐ Actual values
- ■ Actual c1 values
- ☐ Results: actual values <
- ■ Results: actual c1 values <
- ☐ Results: actual values <

Last Test Documentation [graphics:is_line_covered_by_rectangle]
File Search Help

```
Test Cases not executed:
    9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
    28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
    46, 47, 48, 49

Results of C1-Coverage Measurement
==================================

    40.00 % C1-Coverage

Test Cases Used for C1-Coverage Measurement:
    1, 2, 3, 4, 5, 6, 7, 8


                          Test Case Documentation
                         ========================

Test Case :   2
================

Number of test steps:   1

Test Case Specification:
========================
  coverage: yes
    degree of coverage: one point
  position of P1: P1 inside rectangle
    Pos. of P1 in Rect.: on one side
      What side: left
  Course of line: slanting
    Direction of line: bot,right - top,left
    Gradient of line: medium
  position of P2: P2 outside rectangle
    Pos. of P2 to rect.: above and left
    Distance of P2: other

Statistic Act(C1)/Exp-Comparison
================================
                                                         Test Step Results
Component Name                                         OK     Not-OK   Manual
--------------------------------------------------------------------------------
is_line_covered_by_rectangle                            1        0        0

Test Step :   1
================

is_line_covered_by_rectangle
   Expected value    : yes
   Actual value(C1)  : yes                                                Ok
   Eval mode         : Equal
```

Figure 6: Test evaluation and test documentation for *is_line_covered_by_rectangle*

5. Conclusion, Practical Experience, and Future Work

The test system TESSY provides interactive tools for test organization, test case determination, test data selection, and expected results prediction. They are specialized on the respective activities and conducive to a systematic test. The classification-tree editor CTE is of special importance, because it supports a thorough and well structured test case determination corresponding to the classification-tree method. Test case execution, monitoring, test evaluation, and test documentation are executed automatically by TESSY (see Table 2).

Test Activities	Degree of automation		
	manual	tool supported (interactive)	full
Test Case Determination		X planned	(X)
Test Data Selection		X planned	(X)
Expected Results Prediction		X planned	(X)
Test Case Execution			X
Monitoring			X
Test Evaluation and Documentation			X

Table 2: Degree of automation by TESSY

Practical trials of the entire test system will start in the fourth quarter of this year in several projects of Daimler-Benz divisions, hence results from practical applications will be available soon. The classification-tree editor has been used successfully in various projects for a systematic test case determination for two years now. In comparison with previous tests significant savings could be achieved: the number of test cases could be reduced and their quality was improved considerably. Thus the cost for the entire tests was diminished substantially. On average, savings up to 50 % could be achieved for the test case determination. Even more savings are expected for the remaining test activities due to the high degree of automation within TESSY. This refers especially to those activities which are totally automated by TESSY.

In the future, extensions for TESSY are planned to enable fully automated unit testing. Owing to the strengths and growing relevance of formal methods in software engineering the use of formal specification techniques is planned. This will enable computer-aided generation of classification trees and test cases as well as automatic generation of test data from formal test case specifications. The predicition of expected results will be supported by executable specifications.

Other fields of future work will be further support for integration and system testing as well as a test system version for Ada programs.

References

DAVIS Davis, C. G., *Testing Large, Real-Time Software Systems*, Software Testing, Infotech State of the Art Report, Volume 2, 1979, pp. 85 - 105.

GROC1	Grochtmann, M., Grimm, K., Wegener, J., *Tool-Supported Test Case Design for Black-Box Testing by Means of the Classification-Tree Editor*, EuroSTAR ’93 – 1st European International Conference on Software Testing Analysis and Review, 25 - 28 October 1993, London, UK.
GROC2	Grochtmann, M., Grimm, K., *Classification Trees for Partition Testing*, Software Testing, Verification & Reliability, Volume 3, Number 2, June 1993, Wiley, pp. 63 - 82.
MYERS	Myers, G. J., *The Art of Software Testing*, John Wiley & Sons, Inc., 1979.