Quality Week 1995

TESSY
–
An Overall Unit Testing Tool

Joachim Wegener
Roman Pitschinetz

Daimler-Benz AG
Research and Technology
Alt-Moabit 96 a
D-10559 Berlin, Germany
Tel: +49 (0)30 39982-232
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 computer-aided test case determination, test data selection, expected results prediction, test execution, and test evaluation. Furthermore, there is powerful support for test organization and test documentation. An integral part of TESSY is the classification-tree editor CTE, a graphical editor for the descriptive and systematic design of black-box test cases following the classification-tree method. Several other 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.

Practical trials of the test system started in 1994. First promising results indicate that the test efficiency is enhanced significantly by using TESSY. 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. It typically consumes 30 % - 50 % of the overall software development effort and budget (e. g. Myers 1979). 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 on 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.

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. First results from practical trials are reported in chapter five. 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 necessary containing, for instance, test cases, test data, expected results, actual values produced by the test object, and global settings required for the test. Furthermore, technical prerequisites for test case execution such as the implementation of stubs and 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. It comprises a set

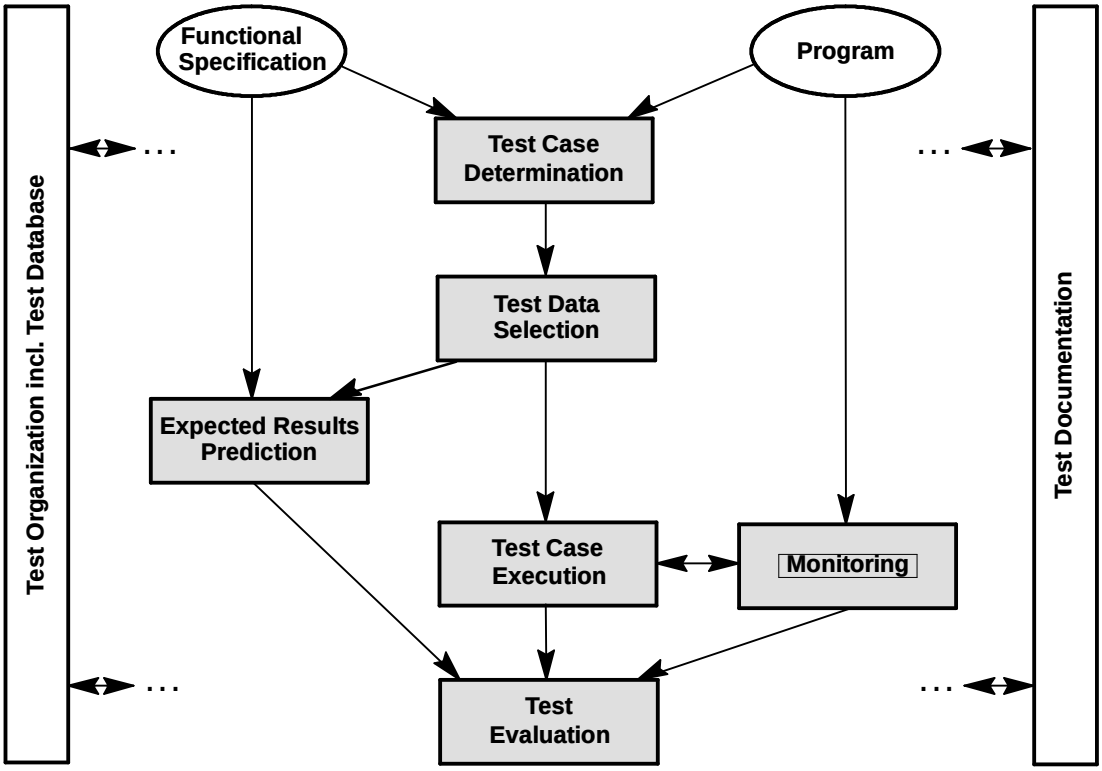

Figure 1: Test activities

of input data from the test object 's input domain. Each element of the set should, for example, lead to the execution of the same program functionality, the same program branch, or the same program state, depending on the test criteria applied. 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 with which the test should be executed.

Determining the anticipated results and program behavior for every selected test datum constitutes expected results prediction. If it is not possible to specify unequivocal output values or the expected behavior, 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 behavior are thus determined.

The behavior 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. Another kind of monitoring is performed by several capture-and-replay tools. They record the outputs produced by the test object on the screen and save them for regression testing.

In the course of test evaluation, actual and expected values as well as actual and expected program behavior 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 behavior have to be documented, discovered errors are to be summarized in error statistics, and the fulfillment of the test objectives should be assessed.

3. Survey of Existing Testing Tools

A large number of tools is available in the area of software testing all over the world. An important fraction of tools put the main focus on static analysis. However, more than 100 testing tools offer support for dynamic testing. A lot of them are capture-and-replay tools, in particular supporting system testing of interactive programs. Approximately 80 tools seem to be applicable for an employment in the field of unit testing (see e.g. Graham 1991, Wegener et al. 1994). A detailed study on these tools, examining their abilities to support the test activities mentioned above, reveals that none of the tools offers general support for all test activities. Nearly 50 % of the unit testing tools concentrate on monitoring. Additional 25 % combine test case execution and monitoring. Only five tools support test case determination but they have decisive deficiencies in the subsequent central test activities.

Table 1 contains an anonymous survey of existing unit testing tools and gives a rough classification. Dark coloured fields indicate activities fully automated by the tools, light coloured areas indicate activities partially automated, for example, using interactive tools. It has to be considered that the table does not regard the effectivness of the tools in practical use.

Due to the lack of software testing tools offering an overall and efficient 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 improved by using TESSY.

4. Test System TESSY

The test system TESSY was developed on VAXstations with the operating system VMS for unit testing of ANSI-C and VAX-C programs. Other C-compilers, especially for host-target cross-

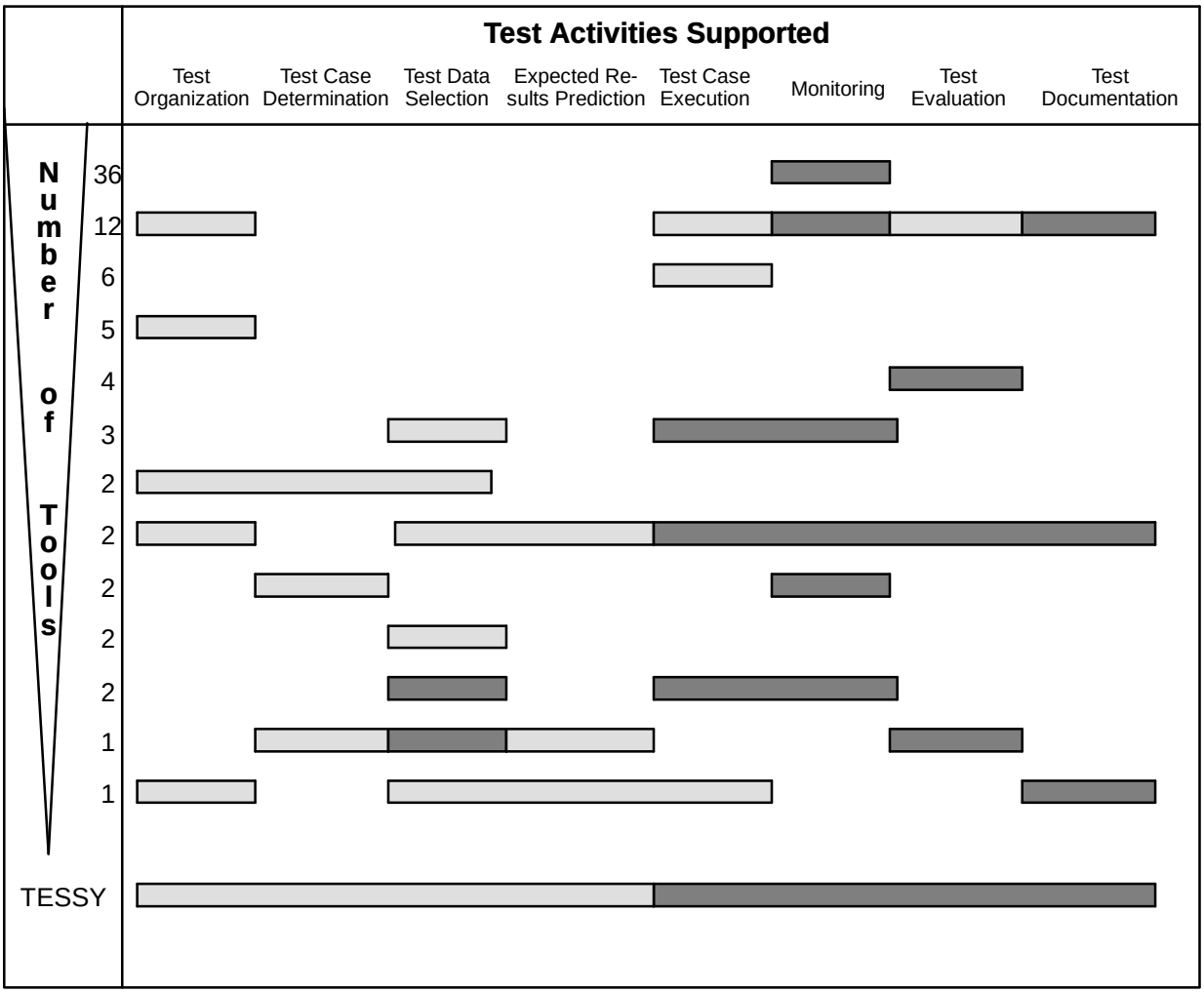

Table 1: Survey of unit testing tools

compiling, were added during the first half of 1995 corresponding to the needs of TESSY users. Transfer to SUN/Solaris systems is currently in progress.

The most important strength of TESSY is that it provides support for the whole testing life cycle, offering a homogenous, object-based, and context-sensitive user interface which guides the tester through all test phases. Accordingly, the use of TESSY is very simple and comfortable. As a result the test system can be applied by testers without detailed knowledge of the C programming language. This promotes the distinction between software development and quality assurance as it is demanded in various standards. The user interface is based on OSF 's Motif toolkit.

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 or not all specified requirements have been transformed into the

test object. Test cases are determined using the classification-tree method and the classification-tree editor CTE (Grochtmann et al. 1995). 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. Based on first results from practical trials TESSY is to be expanded in 1995 by mechanisms which support regression testing also in cases where the test object interface has changed.

TESSY contains separate tools for each test activity which shall be explained successively in the following sections using a realistic example.

4.1. Example

The sample function to be tested using TESSY is the C-function

```
t_answer    is_line_covered_by_rectangle (
                struct line line,
                struct rectangle rectangle )
```

which checks whether or not 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 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 elements 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 automat-

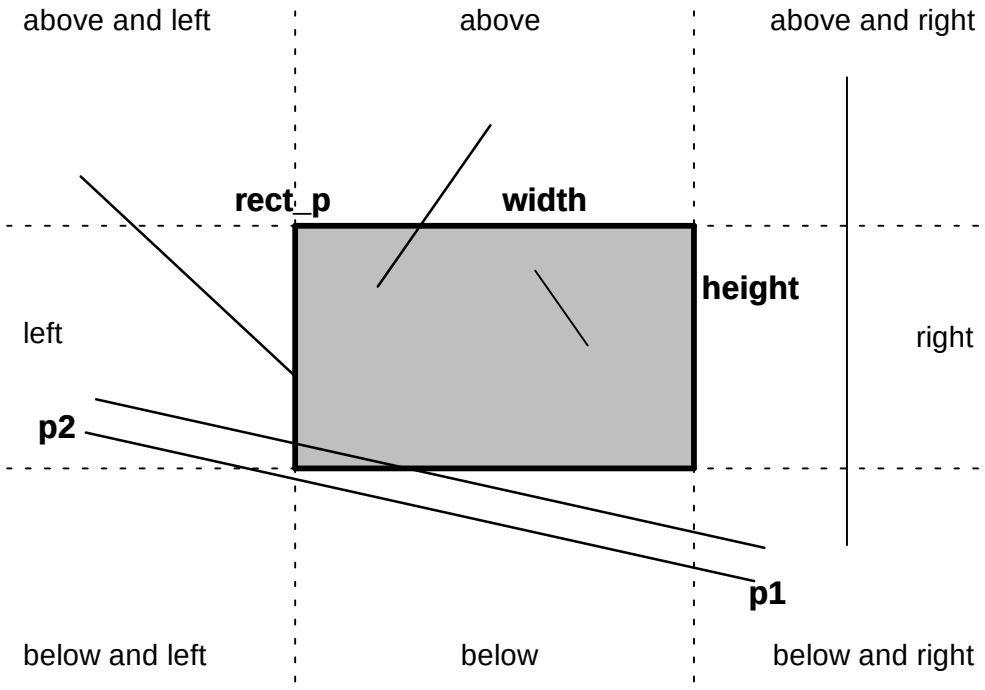

Figure 2: Rectangle with sample lines

ically 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 is always of the kind OUT. Global variables and dynamic values, like pointers, can be of the kind IN, INOUT, or OUT. The completion of the interface description is carried out using 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 sample function.

For the test object is_line_covered_by_rectangle the type of every parameter can be determined automatically from the source code. Both structures, line and rectangle, as well as 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.

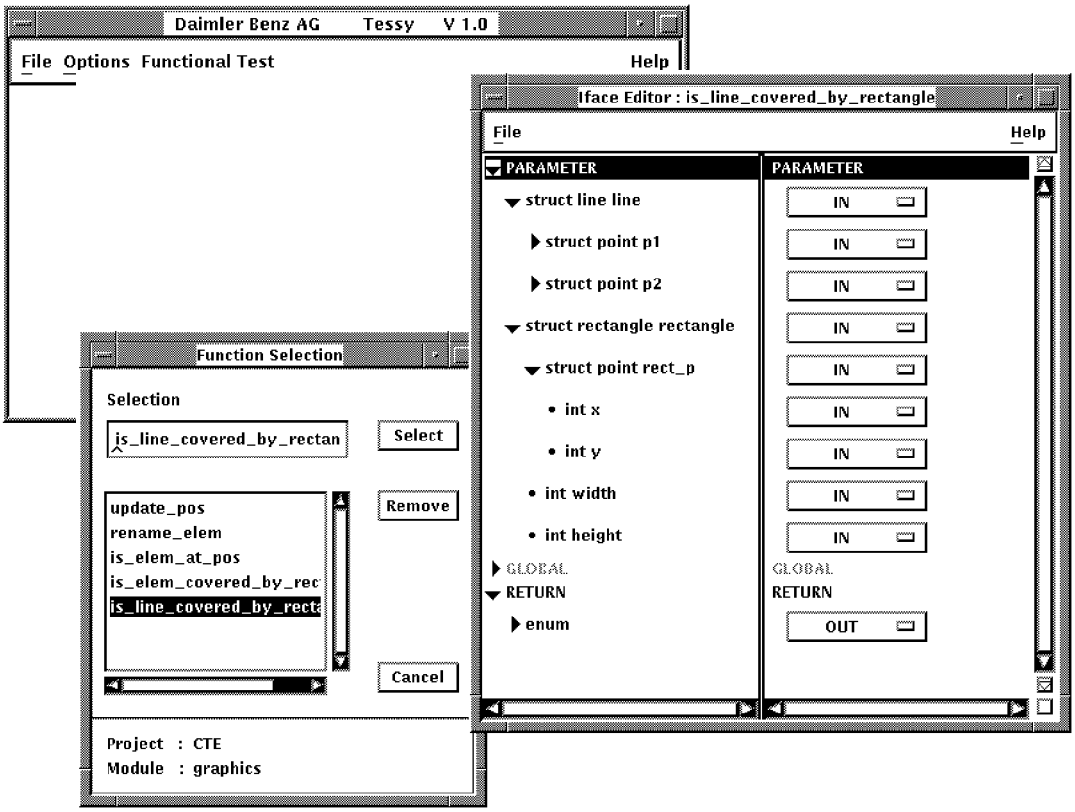

Figure 3: Editor with interface description of is_line_covered_by_rectangle

4.3. Test Case Determination

After having completed the interface information the tester is able to begin the main test activities with the test case determination. Test case determination is carried out by means of the graphical classification-tree editor CTE which supports the application of the classification-tree method (Grochtmann and Grimm 1993). The user edits the classification tree in a syntax-directed and object-based 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 and the classification-tree editor will be presented at the Quality Week 95, too (Grochtmann et al. 1995). Therefore, they are not explained in more detail here.

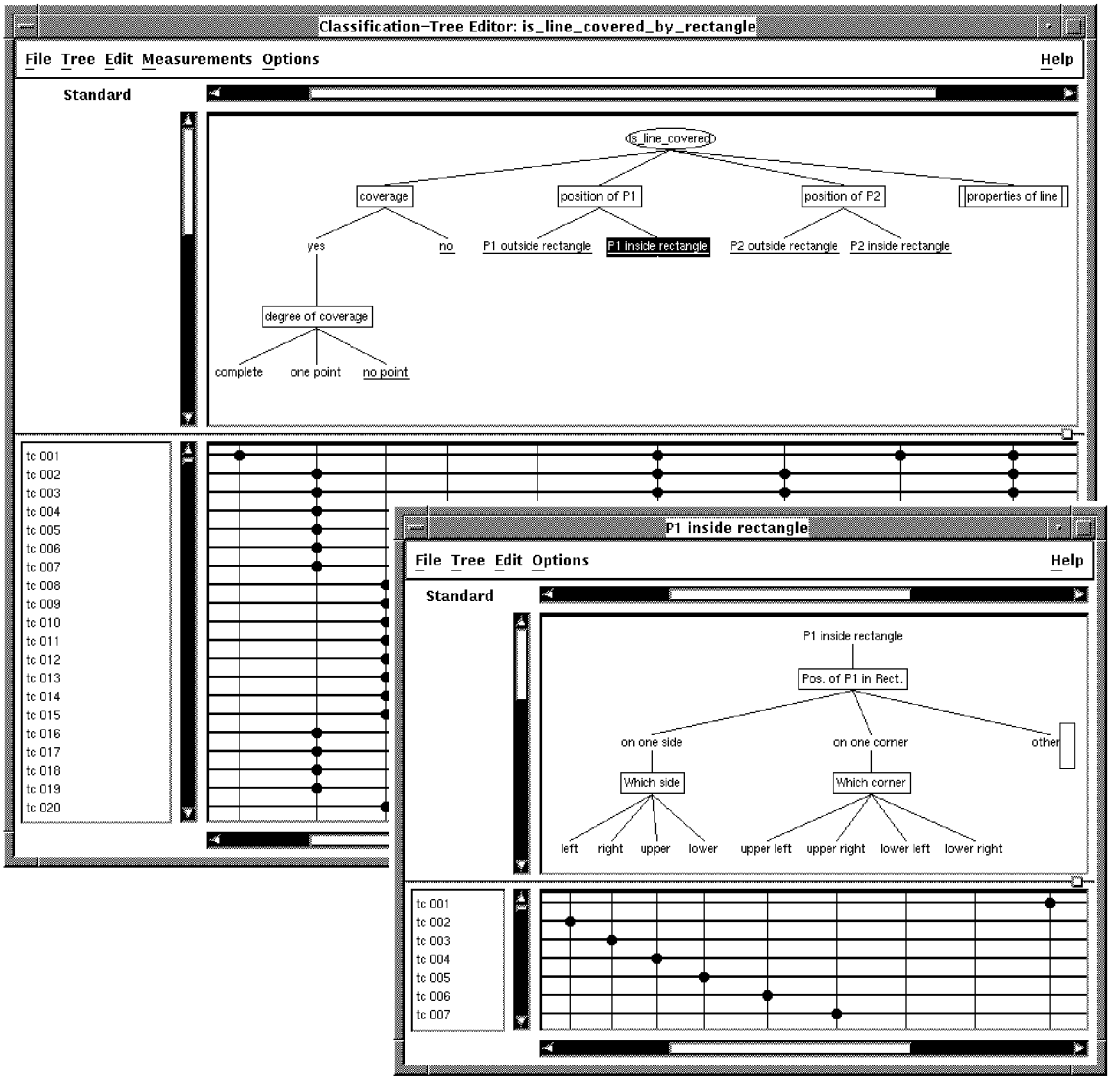

Figure 4: Test case determination for is_line_covered_by_rectangle using the CTE

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 a 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 corresponding sample line is shown in Figure 2.

4.4. Test Data Selection and Expected Results Prediction

Following the test case determination, the user starts the test-data editor TDE to enter concrete test data and to predict the 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 left part of the editor. An additional test case comment is shown on the right in case the tester has entered one in the classification-tree editor. Below these text areas the whole test object interface is presented in two browsers.

The left browser is used to edit test data for the input parameters of the test object, the right one to enter expected results for the output parameters. Each interface 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.

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 for the parameters line and rectangle in the left browser and the expected result for the function return code in the right browser. Coordinates are given as X-Window positions, beginning with (0,0) for the top left corner of the screen. 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 (Figure 6). 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

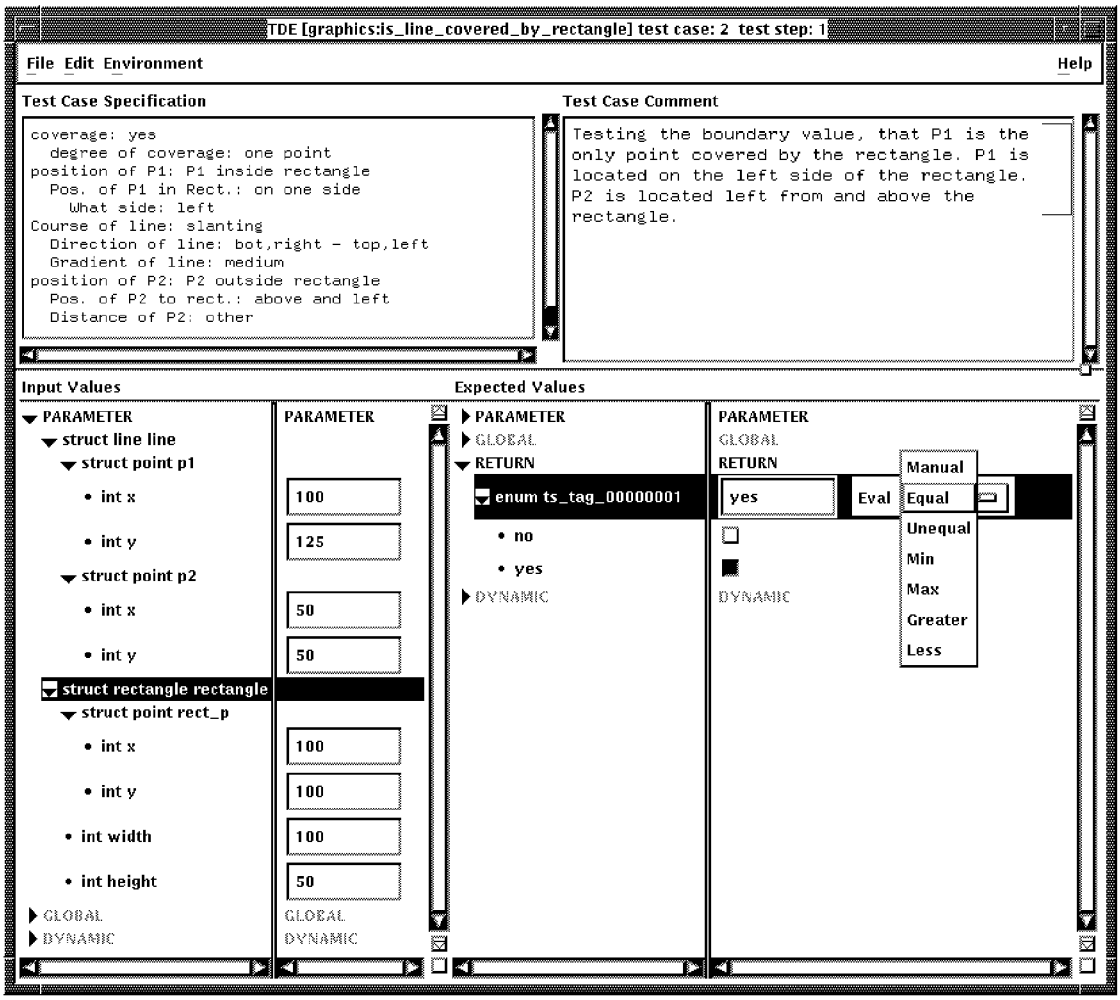

Figure 5: Test data selection and expected results prediction for is_line_covered_by_rectangle

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.

For the sample function is_line_covered_by_rectangle the first eight test cases are to be executed and monitoring is activated for 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 7). The expected results and actual values are compared ac-

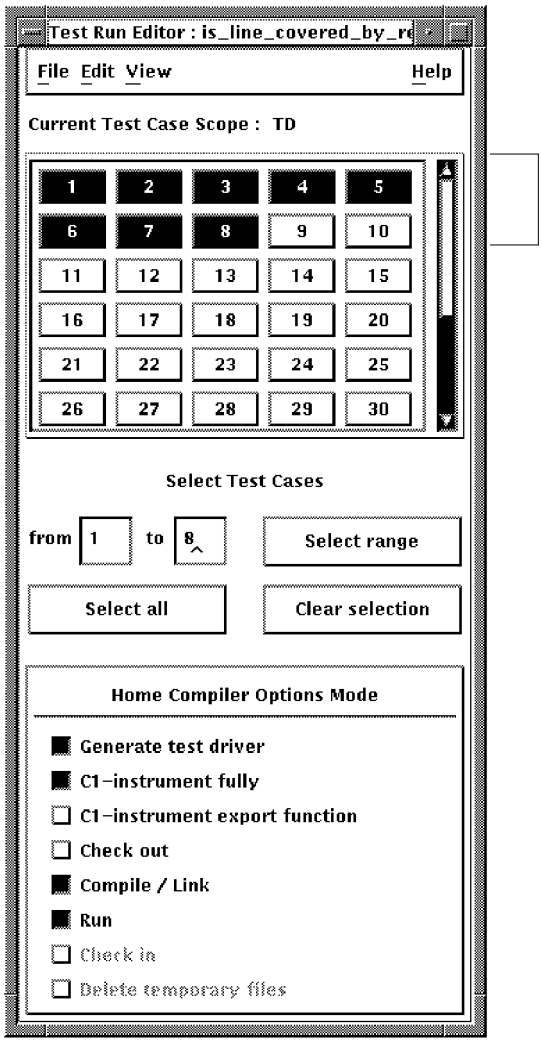

Figure 6: Controlling test case execution

cording 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 7. The first eight test cases were executed, and the branch coverage was measured. 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.

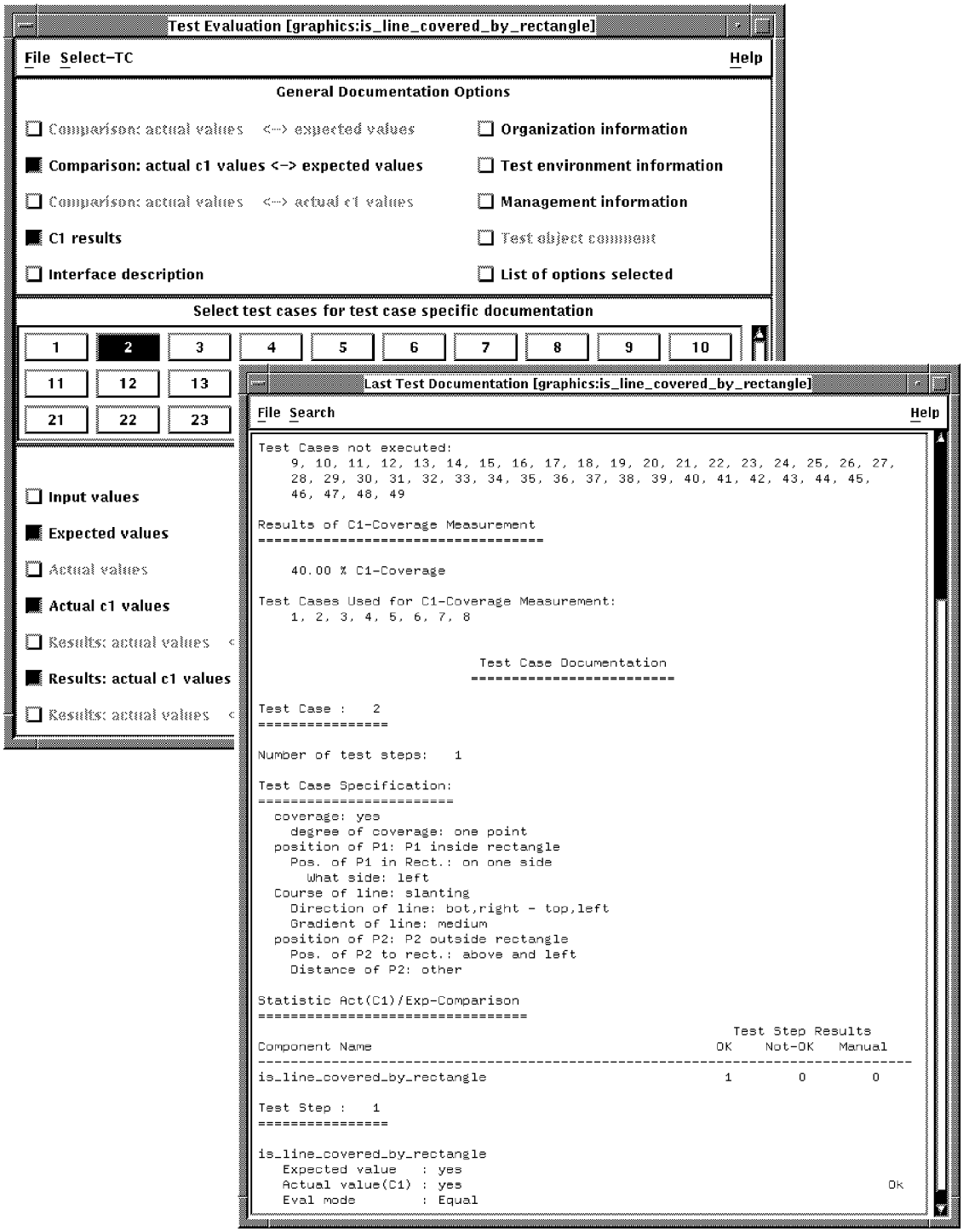

Figure 7: Test evaluation and test documentation for is_line_covered_by_rectangle

5. Practical Experience and Resulting Extensions

The classification-tree editor has been used successfully in various projects for a systematic test case determination for more than three years now. In comparison with previous tests perceptible savings could be achieved: the number of test cases could be reduced and their quality was improved considerably. Savings of up to 50 % could be achieved for the test case determination. Thus the cost for the entire tests was diminished substantially. Because of the positive user acceptance the CTE prototype has been developed to a product version, which will be available in summer 1995.

Practical trials of the entire test system started in the fourth quarter of 1994 in larger projects of Daimler-Benz divisions. The first application was just completed successfully. The test efficiency could be improved significantly. Savings of up to 70 % were estimated by the TESSY users regarding the fact that TESSY totally automates several time-consuming activities. Nevertheless some useful enlargements were proposed and TESSY was extended to offer new functionality for batch processing and host-target testing. For example, several target compilers used in the projects are now supported. Furthermore, two additional interactive tools are currently under development to complement the support for regression testing. They can be applied to reuse test information like test cases, test data, and expected results in cases where the test object interface was changed, or the classification-tree was enlarged by supplementary classifications and classes in order to increase the test quality for regression tests. It is also possible to use them to recycle existing test information for the examination of other test objects.

6. Conclusion and Future Work

The test system TESSY provides powerful editors for test case determination, test data selection, and expected results prediction, as well as interactive tools for test organization and regression testing. 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 (Table 2).

First practical trials of the test system were successful and gave several indications of further improvements. Next, TESSY has to be proven in usual business.

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.

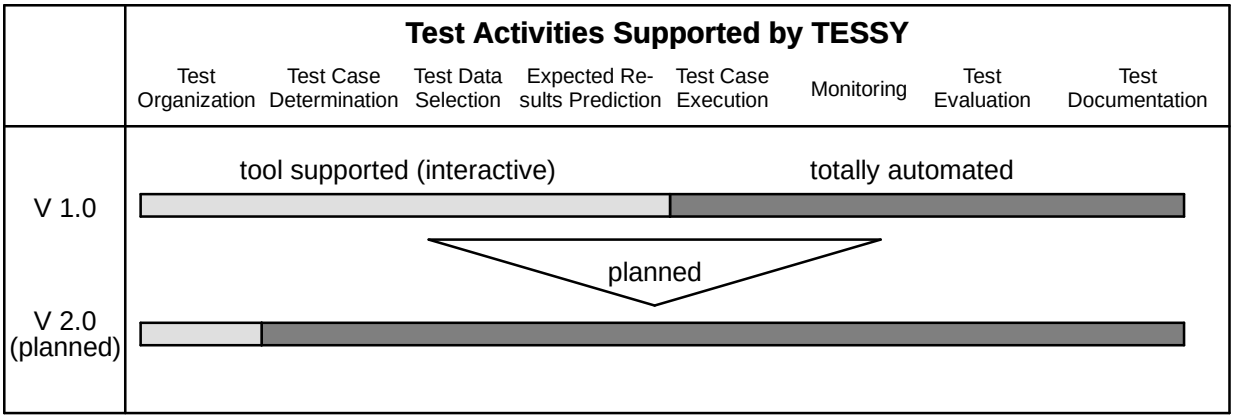

Table 2: Degree of automation by TESSY

References

Graham, D.R. (Ed.) (1991) Computer-Aided Software Testing: The CAST Report. Unicom Seminars Ltd., Middlesex, UK, 1991.

Grochtmann, M., Grimm, K. (1993) Classification Trees for Partition Testing. Software Testing, Verification & Reliability, Volume 3, Number 2, June 1993, Wiley, pp. 63 - 82.

Grochtmann, M., Wegener, J., Grimm, K. (1995) Test Case Design Using Classification Trees and the Classification-Tree Editor CTE. 8th International Software Quality Week, 30 May - 2 June 1995, San Francisco, California, USA, in this volume.

Myers, G.J. (1979) The Art of Software Testing. John Wiley & Sons, Inc., 1979.

Wegener, J., Pitschinetz, R., Grimm, K., Grochtmann, M. (1994) Tessy - Yet Another Computer-Aided Software Testing Tool? EuroSTAR '94 - 2nd European International Conference on Software Testing, Analysis, and Review, 10 - 13 October 1994, Brussels, Belgium, pp. 36/1 - 36/13.