# An Integration Framework for Heterogeneous Automatic Software Tests

Holger Schackmann, Horst Lichter, Veit Hoffmann

Research Group Software Construction
RWTH Aachen University
Ahornstraße 55
52074 Aachen
{Schackmann|Lichter}@informatik.rwth-aachen.de
Veit.Hoffmann@rwth-achen.de

**Abstract:** Developing and maintaining large software systems can require the usage of a variety of different automatic test tools. The complexity of the tools leads to considerable overheads for administration and maintenance of the test cases, as well as for the analysis of the test results. The necessary know-how to handle the test tools may limit their acceptance. This paper describes an integration framework for heterogeneous automatic test tools that unifies the test case administration, test execution, and reporting of the test results.

## 1 Motivation

The usage of automatic software tests has become indispensable for the development and especially for the maintenance of large software systems. A multitude of different test tools can be used for the implementation of test cases, their automatic execution as well as the analysis and presentation of test results [1][2].

For the development of complex software products a variety of those test tools is used. This may be due to different implementation languages of the software components, the needs of testing on different levels (module, integration, system test), the necessity for analysis of different aspects of the test execution (e.g. code coverage, memory usage, run time measurements), and the testing of non-functional qualities (e.g. load test, stress test) [3][4].

If a product family has to be developed and maintained several different configurations of a software product must be tested, since its customers may impose variant requirements to the functionality of the product and different constraints on the operating environment or the deployment the software product [5][6]. Thus a test often requires a complex setup of the environment, e.g. the setup of an underlying database with required test data, the setup of other applications, and the deployment of the software under test. Additionally there are typically several configurations to be tested, e.g. earlier releases of the software product that are still maintained. So it may be necessary to administrate several variants of each test case to be able to use the test in different environments.

Furthermore adequate sets of test cases must be provided for the different kinds of tests (e.g. developer test, integration test, system test, nightly build tests), so a test case can be part of several sets of test cases. Thus the administration of the test cases becomes a considerable task within the testing process [1].

Substantial know-how is necessary to use the different test tools, to develop new test cases and adapt existing tests, as well as to set up the correct test environment. This does impose a hindrance to testing early in the development process. Test results and additional information like coverage data and performance measurements can be scattered in several reports generated by the different tools. So it is difficult to get a general overview on the quality status of the software product. These difficulties can lead to considerable overheads in the testing process that limit the acceptance and the benefits of automatic testing.

Some of these problems were observed in the testing process of a software product that is highly configurable and has a large customer base. Parts of the product are implemented using C++ and Powerbuilder [7], while an underlying platform technology is developed in Java. Functional tests on the user interface are enacted using Mercury WinRunner [8]. The Java part is tested with JUnit-tests [9] that are integrated in a nightly build system. The test results are enriched by coverage reporting and analyses of code metrics. Run time measurements allow a comparison of several test runs.

The problems mentioned above were the motivation to develop a test framework that allows the integration of arbitrary automatic software tests. Section 2 describes briefly the requirements for the core of the test framework. Section 3 describes the structure of the framework. Section 4 reports on first experiences and gives an outlook on further work.

## 2 Requirements

Several basic assumptions led the development of the core framework. The framework should be flexible to integrate tests from arbitrary automatic test tools, so it may not depend on a specific testing approach. The only assumption is that each test case returns a result (passed or failed) and can optionally provide details on the test execution. The framework should allow a uniform administration of the tests, offer support for the setup of the test environment, control the test execution and merge the test results in a single report. The test cases integrated within the framework should remain executable independently from the framework. This enables a stepwise transition to the usage of the test framework by integrating the existing automatic tests.

Further major requirements are openness to integrate other test tools, and imposing a minimum of constraints on the management of the test cases and the handling of the test results.

Some commercial test suites, like Borland SilkCentral Test Manager [10] and Mercury TestDirector [11] also offer an API to integrate third party testing tools. But such a solution would impose to bind to a vendor specific tool chain. So these tools are not considered as an alternative since they restrict the flexibility of the solution. Furthermore the framework must be platform independent. The functionality of the framework should be kept simple, since unnecessary sophisticated functionality will impose a barrier for the adoption and reduce the acceptance of the framework

## 3 Structure of the Test Framework

Figure 1 shows the concepts of the framework and their relationships. Tests are hierarchically organized in the **Test Repository**. A **Test Plan** defines a test run by referencing tests in the Test Repository. The **Functional Repository** contains wrappers for automatic software tests and additionally utilities like scripts that are used during test setup. Each test case must reference a **Tool-Wrapper** to connect to an external automatic test tool. After the execution of a test case the tool wrapper sends information on the test results to a central **Test-Result-Collector**. Based on this information a **Reporting Unit** can generate a test report that merges the results based on different possible reporting strategies.

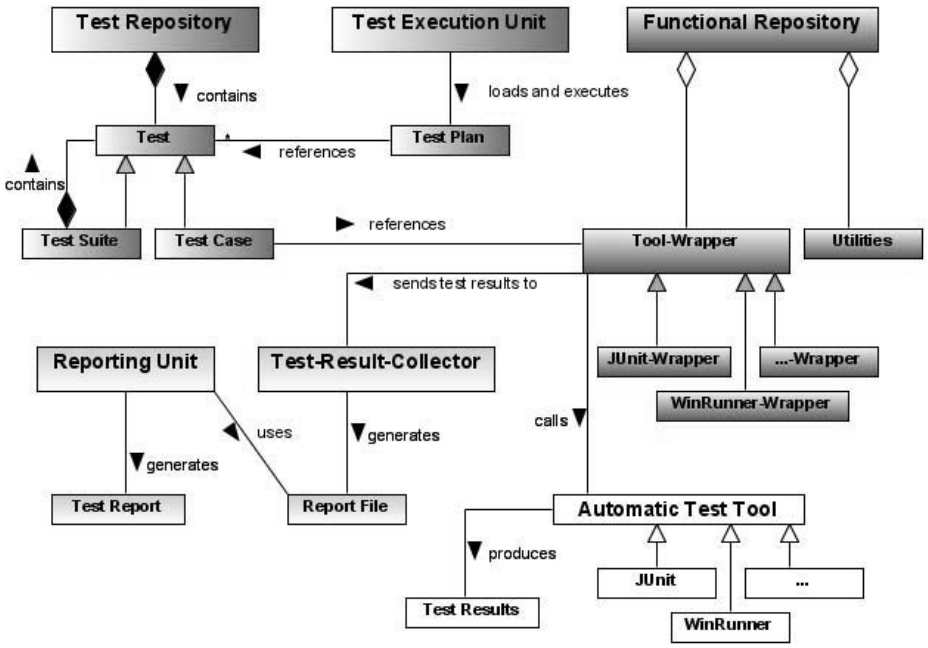


Figure 1: Conceptual overview of the test framework

### 3.1 Administration of Test Cases

The test repository is based on a directory structure in the file system. Hence a user can checkout a specific version of the test repository from a version control system into his local environment. Each directory represents either a **test suite** or a **test case**. A test case is defined by an XML-configuration file that consists of three sections: pre-processing for the test execution (e.g. environment settings), the execution of one or several tests by calling a wrapper of an external automatic test tool, and a post-processing section. Each call to a wrapper within the same test case is called a test step. Tests that are still under development can be marked by placing a file with a predefined name in their directory, so they are excluded during the test run. This facilitates to check in tests early to a central repository and make them visible to other developers. Excluded tests can easily be identified by searching for this file name.

It is assumed that each test case is side-effect free in relation to other test cases. Automatic tests that are dependent on each other must be defined within the same test case definition file. It was a deliberate decision that the framework should not handle dependencies between automatic tests, to prevent inconsistencies within the test repository and keep the usage of the framework simple.

### 3.2 Test Plans

While the test repository itself does just organize the tests in a systematic manner, a test run is defined by a **test plan** given in an XML file. The test plan describes which tests are selected from the test repository by listing the paths to test suites or test cases relative to the root of the test repository. In a pre-processing section the test plan defines environment settings for the whole test run. Additionally global variables can be defined and used in the test plan. Occurrences of global variables within the test plan and in the configuration files of the executed tests will be replaced by the value of the variable. This mechanism enables to keep the definition of the tests in the test repository independent from a specific environment (e.g. location of certain test tools, operating system, database settings, and characteristics of the program), and enables a basic configurability of the test cases. So a test plan can be reused for several configurations of the software product by adapting the environment settings and global variables.

The tests will be executed in the order given in the test plan. The framework identifies the test cases by scanning for the XML definition files in each directory and its subdirectories given in the test plan. Name and version of the test plan, date and time of test execution, the current user, and all environment variables will be recorded in a log file.

### 3.3 Recording and Reporting of Test Results

Each wrapper for a specific automatic test tool parses the output of the tool and sends the resulting information to the **Test-Result-Collector**. So the Test-Result-Collector is able to collect information on the executed test suites, test cases, test steps, their results and their execution time. Additionally the wrapper can provide comments or links to further information on the test results, like output files generated by an automatic test tool. The collected information will be recorded in a report file in a predefined XML format. The Test-Result-Collector is implemented as a TCP-Server. Since parallel tests are an upcoming requirement for the test framework, this enables a distribution of the test framework with a central interface for the collection of the results.

Test reports itself are generated by a reporting component that is independent from the test framework and just processes the report file. This enables to implement different reporting formats or strategies, e.g. an enriched report with information on test coverage and performance, or an executive report with a summary on the quality status. Hence concise reports for different target groups can be created. A basic HTML report is implemented by applying an XSLT transformation on the test result file. The report presents the information on the test execution in a tree view that is structured analogously to the tests given in the test plan.

## 4 Experiences and Outlook

The core framework was implemented in Perl, so it does not depend on a specific platform. It was possible to integrate the already existing large JUnit- and WinRunner test suites of the considered software product, without the necessity to modify the existing tests.

The test framework offers several possibilities to improve the maintainability of the test cases. By using global variables within the test plans and test cases, the tests can be adapted to different environments. Moving frequently used scripts sequences into the Functional Repository can help to improve the modularity of the tests. It still needs to be investigated how to reorganize the existing test suites to achieve better maintainability and reusability of the test cases.

Except for a small tool that supports the editing of test plan and test case configuration files, the framework does not offer a graphical user interface. According to our current experience, this does not limit the acceptance of the framework, since its target users are used to work with command line tools and other supporting tools like version control system clients.

The generated test report subsumes the information that was contained in a previous JUnit-test report, and presents the test results of all automatic tests in the same way. The usage of the XSLT transformation allows adjusting the details of the report to the needs of the test engineers.

Currently the following extensions are considered for the test framework. The administration of test cases and test plans can further be improved by providing tools to modify test plans and ensure their consistency, e.g. by checking if all global variables in a given set of test cases are defined in a test plan. Possibly more mechanisms are necessary to support the evolution of test plans, e.g. tracking the dependencies between several variants of a test plan, or allowing a test plan to include other test plans.

The usage of the framework will show whether the assumptions that were made to keep the framework simple can be preserved, e.g. the restriction on side-effect-free tests or the abandonment of managing dependencies between test cases.

A major enhancement will be the support for asynchronous tests and the execution of load tests. This may require that the framework has to be distributed and measurements of response times have to be integrated in the test reports.

## Literature


[1] Fewster, M.; Graham, D.: Software Test Automation. Addison-Wesley (ACM Press), New York, 1999.

[2] Pol, M.; Koomen, T.; Spillner, A.: Management und Optimierung des Testprozesses. dpunkt Verlag, Heidelberg, 2000.

[3] Spillner, A.; Roßner, T.; Winter, M.; Linz, T.: Praxiswissen Softwaretest – Testmanagement. dpunkt Verlag, Heidelberg, 2006.

[4] Nguyen, H.Q.; Johnson, B.; Hackett, M.: Testing Applications on the Web. Wiley. Indianapolis, 2003.

[5] Clements, P; Northrop, L.: Software Product Lines – Practices and Patterns. Addison-Wesley, Boston, 2002.

[6] Pohl, K.; Böckle, G.; van der Linden, F.: Software Product Line Engineering. Foundations, Principles, and Techniques. Springer, Berlin, 2005.

[7] http://www.sybase.com/products/development/powerbuilder/

[8] http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/

[9] http://www.junit.org

[10] http://www.borland.com/us/products/silk/silkcentral_test/

[11] http://www.mercury.com/us/products/quality-center/testdirector/