Test Case Structuring and Execution Control in an Integration Framework for Heterogeneous Automatic Software Tests

Andreas Ganser[1], Holger Schackmann[1], Horst Lichter[1], Heinz-Josef Schlebusch[2]

[1]RWTH Aachen, Research Group Software Construction, Ahornstr. 55, 52074 Aachen
{Ganser|Schackmann|Lichter}@swc.rwth-aachen.de

[2]KISTERS AG, Charlottenburger Allee 5, 52068 Aachen
Schlebusch@kisters.de

Abstract: Dependencies between automatic test cases are considered as problematic, since they impair understandability and maintainability of these test cases. However, dependencies can not be fully avoided, when test cases require a time-consuming setup of the test fixture. This paper describes the handling of dependencies between test cases within an integration framework for heterogeneous automatic test tools that unifies test case administration, test execution and reporting of the test results. Two types of dependencies between test cases are identified. A simple approach is presented how one of these dependency types is utilized within the framework to gain control during test execution in order to reduce test execution time and unnecessary output.

1 Introduction

To develop and maintain complex software products different tools for automatic software tests are used. On the one hand this is caused by the need to pursue different kinds of tests like system tests, unit tests, or tests of non-functional qualities. On the other hand this is caused by the heterogeneity of the software under test in terms of implementations languages, technologies or underlying platforms. Hence, substantial know-how is necessary for applying these tools with a correct setup of the test environment, as well as for developing and maintaining the test cases. Moreover, test results can become scattered into several reports of different tools. As a consequence, this may lead to considerable overhead in the testing process that limits the acceptance and benefits of automatic testing. This was the motivation for the development of an integration framework for heterogeneous automatic test tools that unifies test case administration, test execution, and reporting of the test results [SLH07]. This framework was developed in cooperation with KISTERS AG. KISTERS AG offers software solutions for the energy markets and the management of the natural resources water and air, based on core software technologies for time series management of measurement values, modeling and forecasts. Due to the large volume of data needed to test these high capacity systems under realistic conditions this application

domain imposes additional challenges. To complete a test suite run in reasonable time, tests cases must share the same fixture or even be executed as chained tests [Mes07]. The resulting dependencies between test cases complicate test case maintenance. Thus, it is crucial to handle these dependencies within the integration framework.
In this paper several patterns for the integration of test cases into the central repository of the test framework will be presented. Then a characterization of dependencies between test cases is given. Further on, an approach will be presented how to utilize certain test case dependencies to reduce the run time of test suites and ease the analysis of test results. Finally, the capabilities of the framework will be discussed based on practical experiences with the integration of a representative test tool.

2 Structure of the Test Framework

This section briefly describes the basic concepts of the framework and introduces several recurring patterns of test structures. It is important to keep in mind that terms like test case and test plan have a slightly different meaning in the context of different test automation tools, as well as in the context of the test framework as described below.

2.1 Basic concepts

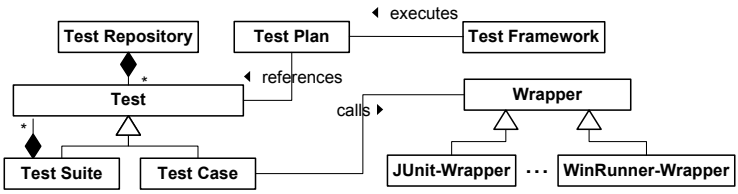

Figure 1: Basic concepts of the test framework

A sketch of the framework's basic concepts is given in Figure 1. The integration of an automatic test tool in the test framework is realized by a *wrapper* that executes an automatic test by a call to the external tool. Moreover, the wrapper is responsible for collecting the results of the tests. General information on the test result, like the test verdict or the execution time, is then delivered to the framework in a unified format. The wrapper can provide optional comments and links to additional files. The framework will merge this information into an overall report. The granularity of the results depends on the results reported by the wrapper. A wrapper must at least report a verdict like "passed" or "failed".
Tests are hierarchically organized into a *test repository* that is given by a directory structure of the file system and is put under version control. A *test suite* is represented by a directory that contains a script which defines pre- and post-processing actions that describe the

common setup and cleanup respectively. A *test case* is represented by a call to a wrapper within the test suite's script. While the test repository represents a static organization of all test cases, a test run is defined by the *test plan*. The test plan lists all related tests with their path within the test repository. Additionally pre- and post-processing sections can be defined. To ease portability and configurability of test cases, a test plan can define global variables. During a test run the tests will be executed in the order given in the test plan. All test results reported by the wrappers will be collected in an xml file and imported into a quality reporting system. This system archives each test run in a database, offers a unified presentation of the results, and enables access through a web interface.

2.2 Patterns of Test Structures

The following patterns are applied within the test framework in order to enable the integration of existing tests from different test tools and facilitate the maintenance of the tests. These patterns can be subdivided into patterns which describe the static structure of tests during design time, and patterns which describe the dynamic structure in order to control test execution.

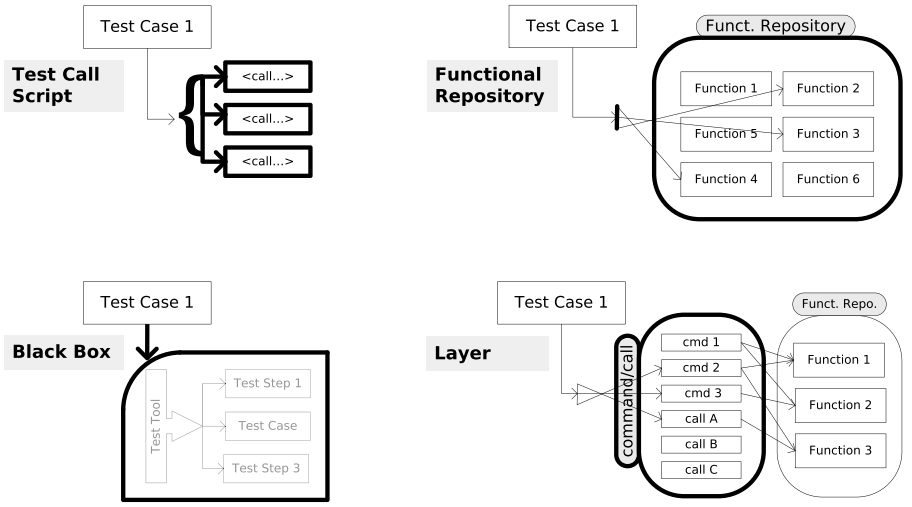

Figure 2: Patterns of static test structures

The patterns of static test structures comprise what we call *test call script*, *black box*, *functional repository* and *layer* pattern. Some sketches of the patterns are given in Figure 2. The names and ideas of the patterns are taken from literature [FG99] [Bin00] [Vig05].
A *test call script* implements one or more test cases by calls to the wrappers. Moreover it can encapsulate the static structures mentioned after.
In case the framework regards a test tool as a *black box*, it calls the test tool which takes on responsibility for the entire execution, the further selection of test cases or test suites,

and, most important, bears responsibility for the entire reporting and for the verdict.
A pattern of *functional repository* comprises a repository of functions from which test cases can be constructed. This repository works as a tool box with reusable functions and delivers no evaluation of the functions. Accordingly, the wrapper is responsible for the reporting and the verdict.
The *layer* pattern works similarly to the functional repository but includes the evaluation of the test. The main difference between both patterns is the abstraction the layer represents. The idea is to encapsulate complex calls in simple names and, furthermore, provide an abstract code for tests.
The only pattern of a dynamic structure is the *test plan* as introduced in section 2.1. This pattern builds upon test call scripts.

3 Dependencies between Tests

A test case sometimes has dependencies to other test cases, for example *chained tests*, where a subsequent test case relies on data outputs or the system state that has been created by the preceding test. Such dependencies are considered as problematic, since understandability and maintainability of these test cases is impaired [Mes07] [Bla03]. But there are pragmatic reasons to run some tests as chained tests. If the setup of a test's fixture is very time consuming, like importing large data volume of measurement values, the setup time can be saved by using the data output or system state created by another test.
Hence we had to face the problem how to handle these dependencies within the test framework. Further investigation showed that two types of dependencies between tests can be identified (see section 3.1). In the following we introduce these types of dependencies and describe how the existing framework was enhanced with a lightweight controlling mechanism with respect to a verdict dependency between tests.

3.1 Chained-Test Dependency and Verdict Dependency

Examining dependencies, the point of view and the order of tests need to be mentioned. Firstly, tests are regarded how they are presented to the framework. This means a test case is a call of a wrapper as illustrated in Figure 1 and, therefore, the smallest element the framework is able to recognize and control. Notwithstanding, it might later be necessary to inspect test cases of a test tool which are called by the wrapper. Secondly, test cases are always ordered in a linear sequence. Therefore the perspective from a single test case can be forwards and backwards which describes the preconditions and the postconditions respectively. No matter what perspective is taken the relationships between two test cases are subsumed and referred to as dependency. A backwards perspective is usually described as follows:

$$\text{if } X \text{ then } proceed \text{ else } omit$$

But what is condition X? Simply put, it can be the status of the system or an aggregation of the test's outputs, e.g. the test verdict.

Whenever the status of the system is the condition, the tests are chained tests. This means the test executed beforehand produced the preconditions for the upcoming test. This needs to be avoided because chained tests are difficult to maintain and understand [Mes07]. Nevertheless those chained tests are found in reality very often. This is because bringing the entire system in the required status is often very tedious and time-consuming. This means test chains are intended to save time and money. The other side of the coin is, the overall status of the system can not be assured due to the amount of data you have to ensure.
In case the condition is an aggregation or interpretation of some output, the most condensed form is a verdict like “passed” or “failed”. Such verdicts are the pieces of interest in this paper, because the decision to execute or to omit a test case will be made based on the verdicts. A verdict is usually determined after the execution of a test case by comparing actual outputs with expected outputs. Hence the status of the system is not involved with how the decision is made.

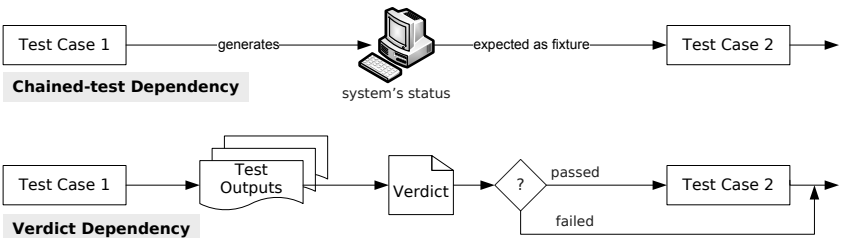

Figure 3: Dependencies between test cases

Talking in terms of dependencies, there are two distinguishable kinds as shown in Figure 3. To begin with, the *chained-test dependency* subsumes all conditions based on the status of the system; next, the *verdict dependency* comprises all conditions concerning verdicts.

Merging chained-test dependencies and verdict dependencies, the latter is a subset of the former. Indeed testers often like implying from verdicts to the status of the system. Hence chained-test dependencies are often applied in the context of test case design without evaluating the overall status of the system, but simply using some verdicts.

3.2 Verdict Dependencies in the Test Framework

The verdict dependencies need to be examined at different levels of test design. First, a look at the level of test cases is needed and, second, a look inside a test case is needed to regard test steps. At the level of test cases the verdict dependencies are quite useful. In fact verdicts open up an elegant way to gain control about test runs. This control can be done with a little help of knowledge about previously run tests. So, it might be possible to deduce whether the upcoming test case is likely to succeed. If there is no way it will succeed it can be omitted. For example, a failed login in the first test case makes a second useless which logs in and tries to change the user’s name. Omitting the second test case reduces the execution time and keeps the tester away from inspecting outputs which deliver

no further insights. Therefore, the verdict dependencies became the idea the framework was improved with.
Since the framework bases test runs on test plans (section 2.1) and should evaluate verdicts during run time, the verdict dependencies need to make an impact on the dynamic pattern mentioned in section 2.2. Hence, the description of a test element was extended with a keyword "requires_passed" as shown in Figure 4. This keyword models the verdict dependency of the current test case and takes a list of test cases which define its precondition.

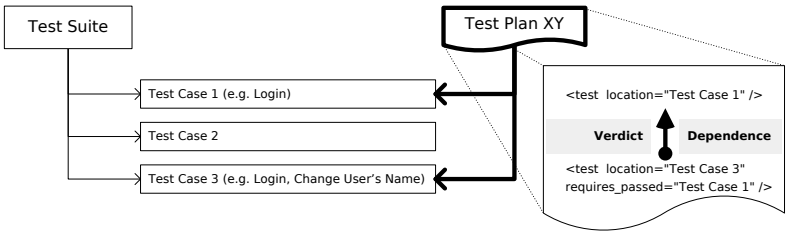

Figure 4: Sketch of a verdict dependency in a test plan

4 Integration of a Representative Testing Approach

The main intention for considering verdict dependencies lies in optimizing test runs and outputs. But testing approaches need to be well structured and designed to gain full benefits from these dependencies. Therefore, a representative testing approach was integrated into the framework in order to assess the usefulness of the concept. A model for evaluating tests in some sort is presented in [ZVS+07], which bases on the ISO model for software quality [ISO]. An appropriate language to name problems, in terms of test smells, was found in the area of object oriented xUnit test patterns [Mes07]. The above mentioned chained tests therefore suffer from interacting tests in terms of test smells.
Such and alike problems were investigated during integrating an application specific test tool and the existing test suites into the framework [Gan07]. This tool implements a typical testing approach referred to as *reference version approach* [FG99]. It compares expected outputs, the references, with the obtained outputs in order to find differences. Whenever a difference is found, the test is assumed to have failed. Therefore the expected outputs need to be prepared beforehand. These references can be of many different types: text files, screen shots, dumps of databases et cetera.
With respect to the above mentioned patterns, the approach was transformed from a black box pattern to a test call script pattern. In detail the testing approach was split up in controllable parts, what is one call of the application specific test tool per test case. This enabled the integration of this approach in the framework with the test call scripts necessary for the wrapper calls. Moreover, the verdict dependencies could be applied to the testing approach, and the results of the comparison were improved by parsing these results into xml-formatted documents. These documents can now be imported to the company's

reporting system which increases the acceptance of the framework.
Regarding the quality model, the evaluation was done slightly altered, because the evaluation was not of a test specification but a testing tool. Most important were the strong improvements of understandability, analyzability and the improvement of the test repeatability. By contrast, the time behavior is affected negatively due to the overhead the framework imposes. Besides the usage is more complex since the framework requires more configuration settings. But we think that the improved controllability, the embedding of the framework in the application domain and the better understandability of the test outweigh the mentioned disadvantages.
In terms of test smells, we were able to eliminate a smell called "manual result verification" in consideration of condensing the test run to a single verdict. This was achieved because on the one hand the verdicts exist at every stage of abstraction, from test steps over test cases up to test suites and test plans. On the other hand a parser transforms variances in the comparisons into verdicts. Moreover, the framework's ability to ease automatic test runs and result analysis reduced the danger of "infrequently run tests".

Over all the improved controllability of the testing approach finally has facilitated the usage of the verdict dependencies and, the benefits and the simplicity of the concept lead to a positive acceptance. Therefore, the verdict dependencies are expected to be employed increasingly in test suites with tedious fixtures.

5 Experiences and Outlook

With the initial assumption of side-effect free test cases, some existing tests could only be integrated as black box. The usage of verdict dependencies enables the integration on a more fine-grained level and facilitates detailed control of the test case execution by the test framework. Describing verdict dependencies as an optional precondition for a test makes existing dependencies explicit within a test plan. Hence understandability of the test plan is substantially improved.
Moreover, verdict dependencies can be utilized to run very coarse-grained tests as regression tests, especially of high capacity systems. But in case something fails, more details about the defect should be found out. This is what we call a *top-down test strategy*. For this purpose we introduced an attribute named "requires_failed" which works the opposite way of the "requires_passed". That is the test is only executed in case the named tests in the attribute failed or were not executed. Of course both attributes can be combined.

Further development might involve verdict dependencies for parallelizing the execution of test plans; in other terms "load-balancing" the test plan or test run. Therefore our verdict dependencies are important as a constraint how it is reasonable to distribute the execution of test cases. For example, consider a given test plan with a very long run time, e.g. one day. But this test plan must be done within four hours. In case the test plan can be distributed along verdict dependencies on several machines, maybe four hours will be enough.
Note that parallelizing test plans is different from running test cases for a distributed system. An enhancement of the framework that enables the description of such distributed test scenarios has been implemented as a prototype [Ada07].

6 Summary

In a nutshell, we introduced a framework for automated testing in a heterogeneous testing environment and showed how this approach on high-level test tool controlling can be optimized. In order to come up with a clean concept, we examined possible dependencies between tests and distinguished between chained-test dependencies and verdict dependencies. Though chained-test dependencies are a common way to reduce effort in fixtures, there is no way to gain proper control for the framework. Instead the verdict dependencies open up the opportunity for controlling a test run. For this purpose a test plan must include information about preconditions in terms of verdicts about previously run tests. In case the preconditions are not fulfilled, the execution of a test case is omitted. Using verdict dependencies relies on a certain fine-grained structure of a test case with respect how the test tool works. Therefore some structures were examined and some hints about how to include a testing approach were given.

References

[Ada07] Sofia Adamanova. Distributed execution of heterogeneous automatic software tests. Master Thesis, RWTH Aachen, 12 2007.

[Bin00] Robert V. Binder. *Testing Object-Oriented Systems*. Addison-Wesley, 2000.

[Bla03] Rex Black. *Critical Testing Processes: Plan, Prepare, Perform, Perfect*. Addison Wesley, 2003.

[FG99] Mark Fewster and Dorothy Graham. *Software Test Automation - Effective use of test execution tools*. Addison Wesley Logman Limited, ACM New York, 1999.

[Gan07] Andreas Ganser. Testdesign für automatische Softwaretests. Diplomarbeit, RWTH Aachen, 11 2007.

[ISO] ISO/IEC 9126. *Software engineering Product quality Part 1: Quality model, Part 2: External metrics, Part 3: Internal metrics, Part 4: Quality in use metrics*. International Standards Organization.

[Mes07] Gerard Meszaros. *XUnit Test Patterns*. Addison-Wesley, 2007.

[SLH07] Holger Schackmann, Horst Lichter, and Veit Hoffmann. An integration framework for heterogeneous automatic software tests. In Bleek, Schwentner, and Züllighoven, editors, *Software Engineering (Workshops)*, volume 106 of *LNI*, pages 107–112. GI, 2007.

[Vig05] Uwe Vigenschow. *Objektorientiertes Testen und Testautomatisierung in der Praxis*. dpunkt Verlag, 2005.

[ZVS+07] Benjamin Zeiß, Diana Vega, Ina Schieferdecker, Helmut Neukirchen, and Jens Grabowski. Applying the ISO 9126 Quality Model to Test Specifications. In *Software Engineering 2007 (SE 2007, Lecture Notes in Informatics (LNI), volume 105)*, pages 231–242. Köllen Verlag, Bonn, März 2007.