Taming the Raven – Testing the Random Access, Visualization and Exploration Network RAVEN

Helmut Neukirchen

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
University of Iceland, Dunhagi 5, 107 Reykjavík, Iceland
helmut@hi.is

Abstract. The *Random Access, Visualization and Exploration Network* (RAVEN) aims to allow for the storage, analysis and visualisation of peta-bytes of scientific data in (near) real-time. In essence, RAVEN is a huge distributed and parallel system.
While testing of distributed systems, such as huge telecommunication systems, is well understood and performed systematically, testing of parallel systems, in particular high-performance computing, is currently lagging behind and is mainly based on ad-hoc approaches.
This paper surveys the state of the art of software testing and investigates challenges of testing a distributed and parallel high-performance RAVEN system. While using the standardised *Testing and Test Control Notation* (TTCN-3) looks promising for testing networking and communication aspects of RAVEN, testing the visualisation and analysis aspects of RAVEN may open new frontiers.

Keywords: Testing, Distributed Systems, Parallel Systems, High-Performance Computing, TTCN-3

1 Introduction

The RAVEN project aims to address the problem of the analysis and visualisation of inhomogeneous data as exemplified by the analysis of data recorded by a *Large Hadron Colider* (LHC) experiment at the *European Organization for Nuclear Research* (CERN). A novel distributed analysis infrastructure shall be developed which is scalable to allow (near) real-time random access and interaction with peta-bytes of data. The proposed hardware basis is a network of intelligent *Computing, data Storage and Routing* (CSR) units based on standard PC hardware. At the software level the project would develop efficient protocols for data distribution and information collection upon such a network, together with a middleware layer for data processing, client applications for data visualisation and an interface for the management of the system [1].

Testing of distributed systems, for example huge telecommunication systems, is mature and performed rigorously and systematically based on standards [2]. In contrast, a literature study on testing of parallel computing systems, such as high-performance cluster computing, reveals that testing is lagging behind

in this domain. However, as computing clusters are just a special kind of distributed systems[1], it seems worthwhile to apply the industry-proven mature testing methods for distributed systems also for testing software of parallel systems. The future RAVEN system is an example for such a parallel system. As testing and testability should already be considered when designing a system [3], this paper investigates the state of the art and the challenges of testing a distributed and parallel high-performance RAVEN system, even though RAVEN is still at it's initial state of gathering requirements and neither a testable implementation nor a design is available, yet.

This paper is structured as follows: Subsequent to this introduction, Section 2 provides as foundation an overview on the state of the art of software testing. Section 3 gives a glimpse of the standardised *Testing and Test Control Notation* (TTCN-3) which is suitable for testing distributed systems. As the main contribution, this paper discusses, in Section 4, the challenges of testing RAVEN. Final conclusions are drawn in Section 5.

2 An Overview on Software Testing

Software testing is the most important means to give confidence that a system implementation meets its requirements with respect to functional and real-time behaviour. Even though testing is expensive[2], it pays off as it is able to reveal defects early and thus prevents them from manifesting during productive use.

In his seminal textbook on software testing [4], G. Myers defines testing as "[...] *the process of executing a program with the intent of finding errors*". However, software testing is no formal proof. Hence, E.W. Dijkstra remarked that testing can be used to show the presence of bugs, but never to show their absence [5].

While Myers refers in his above definition to a *program* which is tested, a more general term for the object of test is *item under test*. The item might range from a single software component (*unit test*) to a whole software system[3] (*system test* – the item under test is here typically called *system under test* (SUT)) via a composed set of components (*integration test*). The possible levels (sometimes called scopes) of testing are just one dimension of testing as shown in Fig. 1. The second dimension refers to the goal or type of testing: *structural testing* has the goal to cover the internal structure of an item under test, for example, the branches of control flow. To achieve this, knowledge of the internal structure (for example, conditional statements) is required (*glass-box test* [4]). The goal of *functional testing* is to assess an item under test with respect to the functionality it should fulfil with respect to it's specification disregarding internal implementation details (*black-box test* [6]). *Non-functional testing* aims

[1] The main difference between distributed systems and parallel systems is probably that distributed systems focus on communication, whereas parallel system focus on computation.

[2] Up to 50% of the overall software development costs are incurred in testing [4].

[3] This can be even a large distributed or parallel system.

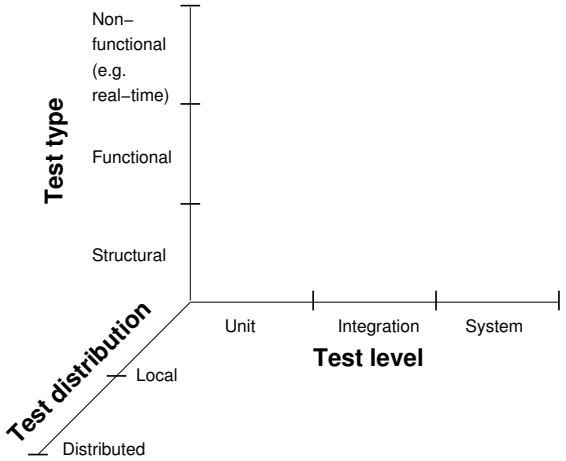


Fig. 1. Dimensions of testing

at checking the fulfillment of non-functional requirements. A variety of different non-functional properties exist, for example, performance with respect to real-time requirements, scalability, security, or usability. The third dimension of testing comes from the fact that the *tester* (or *test system* as it is often called) may be *distributed* (that is: the test runs on multiple nodes) or *local* (that is: the test runs on just one single node). In particular if the item under test itself is distributed, a distributed tester may ease testing or enable certain tests in the first place. The three dimensions are independent from each other, thus the different test types can be performed at all levels and in a local or distributed fashion.

Testing, in particular functional testing, is typically performed by sending a *stimulus* (for example, a function call, a network message, or an input via a user interface) to the item under test and observing the *response* (for example, a return value, a network reply message or an output at the user interface). Based on the observation, a *test verdict* (for example, *pass* or *fail*) is assigned. Testing may be performed manually by running the item under test, providing input and observing the output. Distributed tests, that require co-ordinated test actions, and also real-time performance tests are preferably automated.

Due to the composition and interaction of components, testing at different levels is likely to reveal different defects [7,8]. Hence, testing at just one level is not considered sufficient, but performed at all levels. To reveal defects as soon as possible, a component is unit tested as soon as it is implemented. In case of a class, for example, a unit test would aim at covering all methods. Once multiple components are integrated, they are subject to integration testing. Here, a test would aim at covering the interface between the integrated components. Once the whole system is completed, a system test is performed. In this case, usage scenarios would be covered. All these tests are typically performed within an

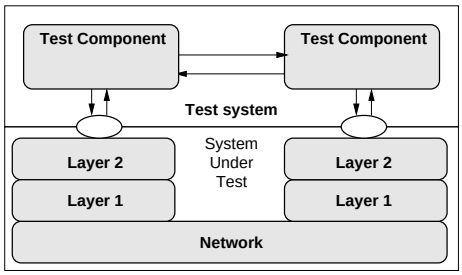
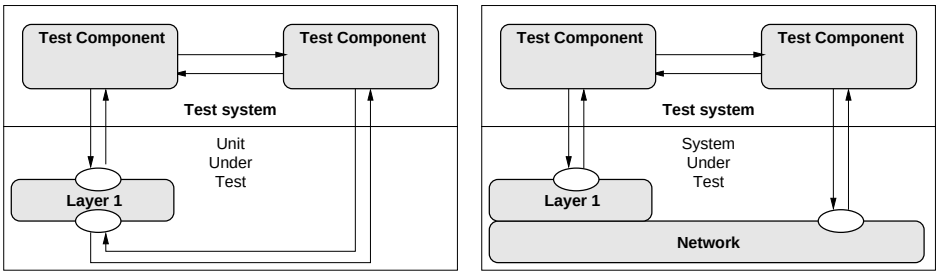


Fig. 2. System test



(a) Ideal (b) Pragmatic

Fig. 3. Unit test

artificial *test environment*, often at special test lab. However, a special kind of system test is the *acceptance test* that is performed in the final productive environment.

Testing in artificial test environments allows test on single or integrated components to be performed in isolation by replacing components on which the item under test depends by special test components. This allows for better control of the item under test and makes sure that really the item under test is tested and not implicitly any of the other components. A layered architecture of a network protocol stack shall serve as an example for this technique: At the system test level, all the involved components (network layer and higher layers 1 and 2 of the protocol stack on both sides) are tested (Fig. 2). At the unit test level, a single unit (such as a class of an object-oriented implementation or an implementation of a network protocol layer as in the example in Fig. 3(a)) is tested. As shown in Fig. 3(a), the environment of that unit under test is replaced by a test environment consisting of test components that act at the interfaces of the unit under test. In practice, this ideal approach may not be possible (or is too expensive): in a layered architecture, higher layers may have hard-coded dependencies on lower layers (thus the lower layer cannot be replaced) or the lower layers provide quite complex functionality that cannot easily be replaced by a test component. The ISO/IEC standard 9646 *Conformance Testing Methodology*

and Framework (CTMF) [2] suggests to circumvent this problem by testing first the lowest layer in an isolated unit test. Then, the next layer is tested together with the already tested underlying layer (Fig. 3(b)) and so on. As a result of this incremental approach, each layer (or unit) can be tested separately (assuming that the lower layers have been adequately tested) even if the ideal approach is not possible for pragmatic reasons.

3 Distributed Testing with TTCN-3

The *Testing and Test Control Notation* (TTCN-3) [9] is a language for specifying and implementing software tests and automating their execution. Due to the fact that TTCN-3 is standardised by the *European Telecommunications Standards Institute* (ETSI) and the *International Telecommunication Union* (ITU), several commercial tools and in-house solutions support editing test suites and compiling them into executable code. A vendor lock-in is avoided in contrast to other existing proprietary test solutions. Furthermore, tools allow to execute the tests, to manage the process of test execution, and to analyse the test results.

While TTCN-3 has its roots in functional black-box testing of telecommunication systems, it is nowadays also used for testing in other domains such as Internet protocols, automotive, aerospace, service-oriented architectures, or medical systems. TTCN-3 is not only applicable for specifying, implementing and executing functional tests, but also for other types of tests such as real-time performance, scalability, robustness, or stress tests of huge systems. Furthermore, all levels of testing are supported.

TTCN-3 has the look and feel of a typical general purpose programming language. Most of the concepts of general purpose programming languages can be found in TTCN-3 as well, for example, data types, variables, functions, parameters, visibility scopes, loops, and conditional statements. In addition, test and distribution related concepts are available to ease the specification of distributed tests.

As TTCN-3 is intended for black-box testing, testing a *system under test* (SUT)[4] takes place by sending a stimulus to the SUT and observing the response. In TTCN-3, communication with the SUT may be message-based (as, for example, in communication via low-level network messages) or procedure-based (as, for example, in communication via high-level procedure or method calls). Based on the observed responses, a TTCN-3 test case can decide whether an SUT has passed or failed a test.

In practise, testing a distributed system often requires that the test system itself is distributed as stimuli and observations need to be performed at different nodes. In contrast to other test solutions, TTCN-3 supports distributed testing – not only the SUT may be distributed or parallel, but also the test itself may consist of several *test components* that execute test behaviour in parallel. The parallel test components may even communicate with each other to co-ordinate

[4] Or any other test item depending on the test level.

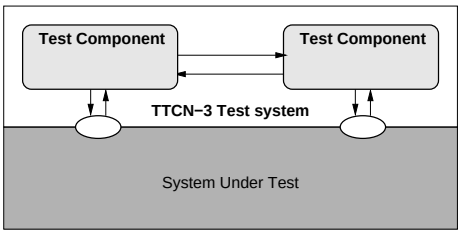


Fig. 4. A sample distributed TTCN-3 test system

their actions or to come to a common test verdict. Figure 4 shows an example of a distributed TTCN-3 test system.

TTCN-3 test cases are abstract, this, for example, means they do not care on which concrete nodes the parallel test components are distributed. It is therefore the responsibility of the TTCN-3 test execution tool to perform the mapping of the abstract distributed test onto a concrete distributed test environment [10]. Thus, the abstract TTCN-3 test cases can be re-used in different distributed environments.

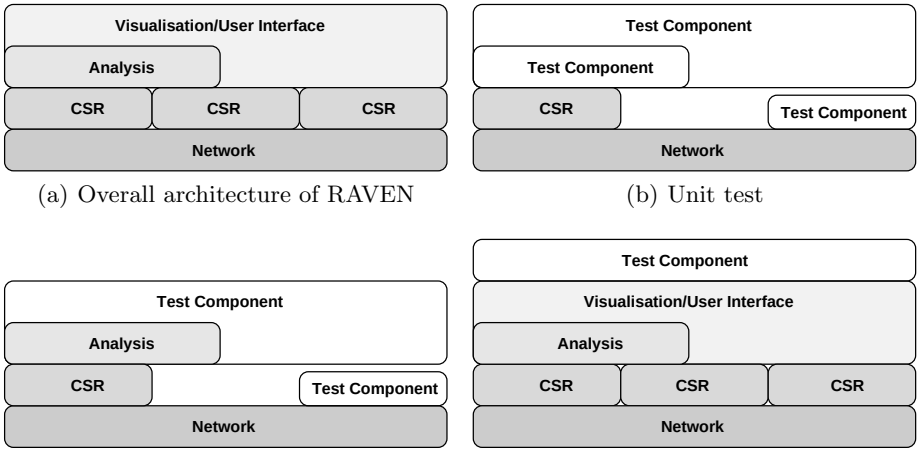
TTCN-3 and it's predecessors TTCN and TTCN-2 have been successfully applied by industry and standardisation for testing huge distributed systems (such as the GSM, 3G, and 3G LTE mobile telecommunication systems). In addition to pure functional tests, TTCN-3 has also been used for performance and load tests that involve testing millions of subscribers [11]. While these applications of TTCN-3 were mainly in the domain of testing "classical" distributed systems, only one work is known where TTCN-3 is used in the domain of "classical" parallel systems: Rings, Neukirchen, and Grabowski [12] investigate the applicability of TTCN-3 in the domain of Grid computing, in particular testing workflows of Grid applications.

More detailed information on TTCN-3 can be found in the TTCN-3 standard [9], in an introductory article [13], in a textbook [14], and on the official TTCN-3 website [15].

4 Testing RAVEN

The intended overall architecture of RAVEN is depicted in Fig. 5(a): the *Computing, data Storage and Routing* (CSR) nodes are connected via an underlying network. High-level data analyses are co-ordinated by an analysis layer that distributes the work load to the individual CSR nodes where the respective data to be analysed resides. The analysis results are then visualised by a corresponding visualisation layer that also provides the general graphical user interface. In addition, to support analysis by the human eye, a direct visualisation of the data is possible. To this aim, the visualisation layer accesses directly the CSR nodes.

For testing at the different test levels, the approaches described in Section 2 can be applied: In accordance to Fig. 3(b), Fig. 5(b) depicts how to perform a



(a) Overall architecture of RAVEN (b) Unit test

(c) Integration test (d) System test

Fig. 5. RAVEN and different test levels

unit test of a CSR node. An integration test of a CSR node and the analysis component that have been integrated together is shown Fig. 5(c). For the system (and acceptance) test, the whole system is covered using representative usage scenarios, however the system under test is interfaced via the user interface only as depicted in Fig. 5(d).

Functional testing of RAVEN should be performed at all test levels. For functional testing of the networking and communication aspects, the proven TTCN-3-based standard approach from protocol testing for telecommunication and network systems [2] is applicable including distributed testing. For testing the user interface and visualisation aspects, the standard *capture/replay* testing approach[5] may not work here as RAVEN aims at providing completely new, yet unknown kinds of graphical user interfaces that are not supported by current capture/replay tools. However, testing of the user interface parts that are based on standard user interface technology should be possible. Testing of the analysis aspect should be feasible as long as small "toy" examples are used, where the result is known in advance.

4.1 Challenges of Testing RAVEN

While functional testing of RAVEN seems to be feasible as described above, the non-functional test types (performance test, scalability test, load test) that seem to be crucial for a system such as RAVEN that aims at providing (near)

[5] In capture/replay testing, a correct interaction with the system under test via the user interface is recorded (user inputs as well as resulting system outputs). For testing, the recorded user inputs are replayed and the resulting actual system outputs are compared against the expected recorded system outputs.

real-time responses and scalability can be expected to be a challenge. While these tests may be possible at unit level, performance results from unit level may not be extrapolated to the system level as, for example, scalability at unit level does not imply scalability of the system as a whole due to management and communication overheads. Thus, the non-functional tests need to be performed at the system level. However, for system level testing of a system of this size, probe effects [16] may occur at the software (and hardware) scale: by observing (= testing) a system, we unavoidably influence it. For example, the communication overhead to co-ordinate distributed test components reduces the network capacity that is available for the RAVEN system that is being tested. Similarly, the actual observation requires CPU time that is lacking in the RAVEN system under test.

A further challenge is how to test analysis algorithms working on peta-bytes of test data. Comparison of the observed output with an the expected output may be difficult if the expected output is not known in advance as it can only be calculated by the analysis algorithm under test itself[6]. Furthermore, these peta-bytes of test data need to be generated and stored. However, as RAVEN itself will be challenged by storing and processing peta-bytes of data, the test environment will be challenged as well to manage the test data. Thus, testing of RAVEN will only be possible to a certain limit within an artificial test environment. RAVEN will require a huge amount of hardware and it will not be economically feasible to use a comparable amount of hardware just for setting up an artificial test environment. This fact inevitably results in the conclusion that performance and scalability beyond a certain size will only testable by productive use in the real environment. As such, a system test within a test environment will not possible. Instead, RAVEN can only be tested immediately in it's real hardware environment – that is, only acceptance testing will be possible[7]. However, performance and scalability assessments of RAVEN beyond a certain size may be evaluated by simulation or analytical techniques based on performance models [18,19].

5 Conclusions

We have considered the state of the art in software testing and at the distributed test language TTCN-3, and we investigated possibilities and challenges of testing a RAVEN system.

It seems that the state of the art in software testing is in principle mostly mature enough for testing a RAVEN system. However, it is the fact that the large scale of RAVEN itself opens new frontiers that poses problems. As a result, a

[6] This is a current research topic, see for example the "First International Workshop on Software Test Output Validation" 2010.

[7] The approach taken by others (for example the Apache Hadoop project for processing huge data sets [17]) confirms this: only small tests are performed in an artificial test environment – "big" tests (performance and scalability) involving huge amounts of data are essentially productive-use tests.

true system test in an artificial test environment will not be possible because beyond a certain limit (in terms of size and complexity), only acceptance testing in the final environment will be possible. To some extent, the problems to be expected concerning the test of RAVEN are unavoidable due to testing overheads and probing effects on the software, hardware and network scale. The challenges of testing RAVEN should not lead to the conclusion to perform no testing at all, but to the contrary: to test where possible.

TTCN-3 is a test language that has concepts for testing distributed and parallel systems, mainly by supporting distributed testing based on concurrently running parallel test components. While these concepts are successfully applied for testing distributed systems, there is a striking lack of applying them for testing parallel systems. One reason might be that most software in parallel computing is from the domain of scientific computing. Typically, this software is written by the scientists themselves who are experts in their scientific domain, but typically not experts in software engineering thus lacking a background in software testing. Another reason is probably that distributed systems and testing them has a strong focus on message exchange and communication, while in parallel systems this is only a minor aspect as the main focus is on computation. However, both kinds of systems can be considered as similar when it comes to black-box testing them; thus, TTCN-3 should be applicable for testing software of parallel systems as well as it is for testing software of distributed systems. However, it still needs to be investigated whether a generic test solution like TTCN-3 (and the implementing TTCN-3 tools) is sufficient and in particular efficient enough, or if specifically tailored and hand tuned test solutions are required.

Finally, as RAVEN aims at not being just deployable on a single cluster, but to extend to external computing and storage resources in the Internet such as cloud computing, a research project has just been started by the author that investigates testability issues in cloud computing environments.

References

1. Schmelling, M., Britsch, M., Gagunashvili, N., Gudmundsson, H.K., Neukirchen, H., Whitehead, N.: RAVEN – Boosting Data Analysis for the LHC Experiments. (This volume).
2. ISO/IEC: Information Technology – Open Systems Interconnection – Conformance testing methodology and framework. International ISO/IEC multipart standard No. 9646 (1994-1997)
3. Wallace, D.R., Fujii, R.U.: Software verification and validation: An overview. IEEE Software **6** (May 1989) 10–17
4. Myers, G.: The Art of Software Testing. Wiley (1979)
5. Dijkstra, E.: Notes on Structured Programming. Technical Report 70-WSK-03, Technological University Eindhoven, Department of Mathematics (April 1970)
6. Beizer, B.: Black-Box Testing. Wiley (1995)
7. Weyuker, E.: Axiomatizing Software Test Data Adequacy. IEEE Transactions on Software Engineering **12**(12) (December 1986)

8. Weyuker, E.: The Evaluation of Program-based Software Test Data Adequacy Criteria. Communications of the ACM **31**(6) (June 1988) DOI: 10.1145/62959.62963.
9. ETSI: ETSI Standard (ES) 201 873 V4.2.1: The Testing and Test Control Notation version 3; Parts 1–10. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2010)
10. Din, G., Tolea, S., Schieferdecker, I.: Distributed Load Tests with TTCN-3. In: Testing of Communicating Systems. Volume 3964 of Lecture Notes in Computer Science., Springer (2006) 177–196 DOI: 10.1007/11754008_12.
11. Din, G.: An IMS Performance Benchmark Implementation based on the TTCN-3 Language. International Journal on Software Tools for Technology Transfer (STTT) **10**(4) (2008) 359–370 DOI: 10.1007/s10009-008-0078-x.
12. Rings, T., Neukirchen, H., Grabowski, J.: Testing Grid Application Workflows Using TTCN-3. In: International Conference on Software Testing Verification and Validation (ICST), IEEE Computer Society (2008) 210–219 DOI: 10.1109/ICST.2008.24.
13. Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A., Willcock, C.: An introduction to the testing and test control notation (TTCN-3). Computer Networks **42**(3) (June 2003) 375–403 DOI: 10.1016/S1389-1286(03)00249-4.
14. Willcock, C., Deiß, T., Tobies, S., Keil, S., Engler, F., Schulz, S.: An Introduction to TTCN-3. Wiley, New York (2005)
15. ETSI: TTCN-3 Website. `http://www.ttcn-3.org`
16. Fidge, C.: Fundamentals of distributed system observation. IEEE Software **13** (1996) 77–83
17. Apache Software Foundation: Apache Hadoop. `http://hadoop.apache.org/`
18. Law, A., Kelton, W.: Simulation Modeling and Analysis. McGraw-Hill (2000)
19. Skadron, K., Martonosi, M., August, D., Hill, M., Lilja, D., Pai, V.: Challenges in Computer Architecture Evaluation. IEEE Computer **36**(8) (2003) 30–36 DOI: 10.1109/MC.2003.1220579.