Hochschule
Bonn-Rhein-Sieg
University
of Applied Sciences

Fachbereich Informatik
Department of Computer Sciences

In Zusammenarbeit mit
Deutsches Zentrum
für Luft- und Raumfahrt
German Aerospace Center

Abschlussarbeit

im Studiengang Master Informatik

Automated testing of distributed systems using on-demand virtual infrastructure

von

Phillip Kroll

Erstbetreuer	Prof. Dr. Sascha Alda
Zweitbetreuer	Prof. Dr. Manfred Kaul
eingereicht am	19.07.2013

Automated testing of distributed systems
using on-demand virtual infrastructure

Master's Thesis
Author: Phillip Kroll
Submission: July 19, 2013

Bonn-Rhein-Sieg University of Applied Sciences
Department of Computer Science
Grantham-Allee 20, 53757 Sankt Augustin (Germany)

Supervisor: Prof. Dr. Sascha Alda
Supervisor: Prof. Dr. Manfred Kaul

German Aerospace Center (DLR)
Simulation and Software Technology
Distributed Systems and Component Software
Linder Höhe, 51147 Cologne (Germany)

Supervisor: Robert Mischke

Contents

List of Figures

List of Tables

Listings

Abstract

Distributed systems comprise distributed computing systems, distributed information systems, and distributed pervasive systems. They are often very complex and their implementation is challenging. Intensive and continuous testing is indispensable to ensure reliability and high quality of a distributed system. The testing process should have a high degree of automation, not only on lower levels (i.e. unit and module testing), but also on higher testing levels (e.g. system, integration, and acceptance tests). To achieve automation on higher testing levels virtual infrastructure components (e.g. virtual machines, virtual networks) that are offered as a Service (IaaS) can be employed. The elasticity of on-demand computation resources fits well together with the varying resource demands of automated test execution.

A methodology for automated acceptance testing of distributed systems that uses virtual infrastructure is presented. It is founded on a task-oriented model that is used to abstract concurrency and asynchronous, remote communication in distributed systems. The model is used as groundwork for a domain-specific language that allows expressing tests for distributed systems in the form of scenarios. On the one hand, test scenarios are executable and, therefore, fully automated. On the other hand, test scenarios represent requirements to the system under test making an automated, example-based verification possible.

A prototypical implementation is used to apply the developed methodology in the context of two different case studies. The first case study uses RCE as an example of a distributed workflow-driven integration environment for scientific computing. The second one uses MongoDB as an example of a document-oriented database system that offers distributed data storage through master-slave replication. The results of the experimental evaluation indicate that the developed acceptance testing methodology is a useful approach to design, build, and execute tests for distributed systems with high quality and a high degree of automation.

“automating chaos just gives faster chaos”
Mark Fewster & Dorothy Graham

1. Introduction

This chapter briefly introduces the problem statement, goals and non-goals, deliverables, and the overall structure of the thesis.

1.1. Problem Statement

Distributed computing introduces an array of issues that add to the complexity of the overall system. In [38] these are summarized as follows: the inevitable need for communication, the incomplete knowledge about the global state of the system, the need to cope with failures, and asynchronous and thus non-deterministic behaviour. Adding to the increased complexity of distributed systems is the difficulty of testing such systems. While low level tests (i.e. unit tests) can be executed without actually deploying the system under test, some characteristics of the system can only be observed in an actual deployment on a targeted infrastructure. Although substantial efforts have been made to automate the development process to continuously integrate [19] and deliver [26] software, approaches to automating test execution are often very specialized and offer little support for tests that focus on “distributed” use cases of the system under test. This leaves the developer with the need for time-consuming manual testing and experimentation.

Another aspect is that the targeted infrastructures of a distributed system may not be available to the developer because it is technically to challenging or economically infeasible to make the target infrastructure available for testing purposes [23]. Often only a single infrastructure might be available for test and experimentation, which might not resemble the productive infrastructure closely enough or might have a scale that is well below (some) productive scenarios. Furthermore, parts of the infrastructure might be recycled for test executions and thus may acquire an undefined state. This limits the reproducibility of test results and the discovery of failures or malfunctions.

Adding to this is the difficulty of creating well-defined scenarios for experimentation in cases of failure. Reasons for potential failures can be numerous and must be dealt with appropriately in the presence of incomplete knowledge. Coping with failures is a required capability of a distributed system, yet investigating and (re)producing failure scenarios is often difficult on an infrastructure level. This might be because failures occur only in rare edge cases (e.g. high load of a central node, limited bandwidth) or very rarely on small test infrastructures.

1.2. Goals of this Work

Motivated by the challenges described above, the main goal of this thesis is to suggest a method that increases the degree of automation for the phase of testing and quality assurance during the software development process. Although distributed systems with loosely coupled, heterogeneous characteristics are targeted (e.g. grid computing applications, distributed databases), the general principles should be applicable to more tightly coupled, homogeneous systems as well (e.g. cluster computing systems). The developed methodology and framework are evaluated using two representative distributed systems: RCE as a distributed grid computing and integration environment for scientific computing and MongoDB as a distributed, document-oriented database that is designed for high availability.

In order to achieve this goal, a number of sub-goals must be addressed. First, how can distributed test cases be modelled taking the complexity of distributed systems into account? Particularly, the following aspects of distributed systems must be considered: the concurrency, the non-deterministic behaviour, the incomplete knowledge, and the need to cope with partial failures. Second, how can test cases be designed in a way that makes them automatically executable by a machine, while ensuring that domain experts, without programming knowledge, are still capable of expressing these test cases together with validation criteria? Third, how can a test automation method be implemented and evaluated using real world case studies? In this context, it is particularly interesting to include today's available infrastructure virtualization technologies into automated test execution. Instead of executing test logic on a single, static infrastructure, automation can be extended to the point where flexible, on-demand infrastructure provisioning is part of the test case.

Non-Goals

On the one hand, the method developed during the course of this work should be a sound and general foundation for automated testing of distributed systems. On the other hand, the implementation of the method is intended to be prototypical and its main focus is to serve as a foundation for the case study-based, empirical evaluation of the method. Although the method is generally agnostic to operating systems, the implementation is focused on UNIX-based operating systems in order to demonstrate feasibility. Finally, the method is focused on the automation of the execution of test cases, instead of (semi) automation for the generation of the test cases.

Deliverables

Appendix A contains detailed listings of some important parts of the source code. Any details that are not printed are available on the enclosed CD. It contains the entire source code that was developed and empirical data that was collected. Appendix B describes the contents of the attached data CD.

1.3. Thesis Structure

The thesis is structured into five chapters that cover the introduction (this chapter), foundations, related work, the method, and its evaluation. Chapter 2 introduces the foundations of software testing and test automation, the characteristics of distributed systems and the technological background of virtual infrastructure and Infrastructure as a Service (IaaS). Chapter 3 continues with an overview of related work in the field of test automation for distributed systems and discusses selected software testing approaches. Chapter 4 introduces the suggested method of test automation for distributed systems. It describes key requirements, introduces a model of testing together with a domain-specific language to express tests and finishes with a description of the implementation of the testing method. Finally, Chapter 5 evaluates the developed method by means of evaluation criteria that are derived from the key requirements of the method. A number of case studies are discussed that aim to demonstrate the additional value of the method.

2. Foundations

This chapter briefly introduces fundamental concepts and central definitions without details. It is structured into three sections. First, software testing is introduced and it is motivated why automation of tests is important far from trivial. Second, the focus of the thesis is narrowed down to test automation for distributed systems. Some important characteristics of distributed systems are introduced along with the discussion of why test automation for distributed systems is important but also challenging. Third, technologies for infrastructure virtualization are introduced and it is shown how these technologies enable a degree of automation that has previously not been feasible.

2.1. Automated Testing

The classic book on software testing, *"The Art of Software Testing"*, 1979 [34] offers a starting point as it provides a brief and precise definition of what software testing entails: "Testing is the process of executing a program with the intend of finding errors". The definition recognizes the fundamental fact that the absence of errors cannot be proved, even for very simple software systems. It follows directly from the definition that any method of software testing is good when it helps to find defects. The definition of testing is directly followed by a list of ten principles for software testing. Two of those principles emphasize that testing should neither be conducted by the programmer nor by the organization that develops the software. Together with the last principle that identifies testing as "an extremely creative and intellectually challenging task" the following guideline can be derived for a software testing method: any testing method must enable non-developers to express test cases while being flexible enough not to limit the creativity of the test author. It is challenging to propose a method that enables a broad audience to express tests without limiting the flexibility of the test author, as both goals might contradict each other to a certain degree.

2.1.1. The Software Testing Spectrum

Software testing is a diverse and unstructured field with a large body of published literature. A lot of disagreement exists about how to categorize types of test. Examples of test categories taken from [18] include functional requirements testing, server performance testing, unit testing, user interface testing, integration testing, program code coverage testing, system load performance testing, program module complexity analysis, security testing and memory leak testing. A more structured approach is suggested in the book *"Practical Software Testing: A Process-Oriented*

No.	Principles taken from Myers [34], 1979	No.	Principles taken from Burnstein [10], 2002
M1	A necessary part of a test case is a definition of the expected output or result.	B1	Testing is the process of exercising a software component using a selected set of test cases, with the intent of (i) revealing defects, and (ii) evaluating quality.
M2	A programmer should avoid attempting to test his or her own program.	B2	When the test objective is to detect defects, then a good test case is one that has a high probability of revealing a yet-undetected defect(s).
M3	A programming organization should not test its own programs.	B3	Tests should be inspected meticulously. (see M4)
M4	Thoroughly inspect the results of each test.	B4	A test case must contain the expected output or result. (see M1)
M5	Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.	B5	Test cases should be developed for both valid and invalid input conditions. (see M5)
M6	Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.	B6	The probability of the existence of additional defects in a software component is proportional to the number of defects already detected in that component. (see M9)
M7	Avoid throwaway test cases unless the program is truly a throwaway program.	B7	Testing should be carried out by a group that is independent of the development group. (see M2 and M3)
M8	Do not plan a testing effort under the tacit assumption that no errors will be found.	B8	Tests must be repeatable and reusable.
M9	The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.	B9	Testing should be planned.
M10	Testing is an extremely creative and intellectually challenging task.	B10	Testing activities should be integrated into the software life cycle.
—	—	B11	Testing is a creative and challenging task. (see M10)

Table 2.1.: Summary of principles on software testing that are suggested by Myers (M1-M10), 1979 and Burnstein (B1-B11), 2002.

Approach", 2002 [10]. Three dimensions are introduced to categorize software tests: test level, test strategy and test characteristics (see Figure 2.1). The test level organizes tests based on their degree of isolation. On the lowest level, small isolated units are subject to the test (i.e. unit test), while on the highest level the system as a whole is exercised by the test (i.e. acceptance tests). The complete list of test levels is: unit, module, integration, system, regression, alpha/beta, acceptance [10, p. 133ff.]. The test level also represents a distinction between tests that help to *build the system right* (i.e. avoid defects) and tests that help to *build the right system* (i.e. meeting the requirements). The test strategy specifies to what extent the internal workings of the system under test is of relevance for a test: white-box test, grey-box test and black-box test. And finally the test characteristics of a test describe the aspects of the software that are under test: portability, maintainability, efficiency, usability, reliability, and functionality.

It is useful to introduce four main classes of defects as it is done in [10, p. 43ff.]. Requirements or specification defects occur when the implementation does not meet its requirements. Design defects occur when components of the system under test or their interactions with each other or external systems are not correctly designed. Code defects are errors that stem from an incorrect implementation, and test defects are errors that are not located in the system under test but within the test cases.

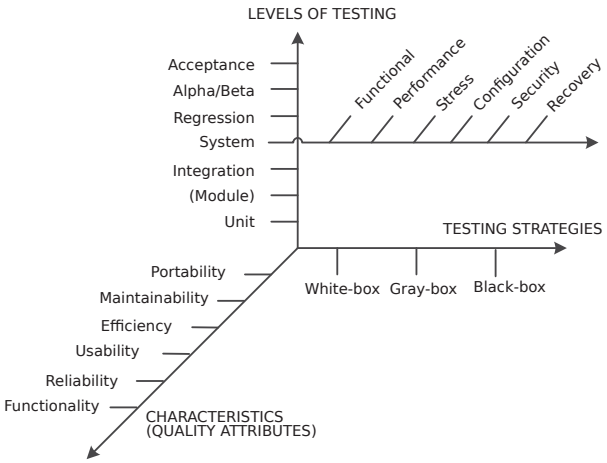


Figure 2.1.: Software testing can be characterized along three dimensions [5, 10]: test level, test strategy, test characteristics

2.1.2. Automated Acceptance Testing

The IEEE defines acceptance testing as "formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system". In the software testing spectrum, acceptance testing is located at the highest test level, which means that the whole product is subject to the test. Most methods for acceptance testing follow a black-box test strategy that provides input through a graphical user interface (GUI) or a command-line interface (CLI). Section 3.3 discusses existing methodologies and frameworks for acceptance testing in more detail.

An interesting aspect of tests on higher test levels is that the focus shifts towards ensuring that the system under test meets its requirements. In other words, specification defects can be targeted in the testing process. From that point of view, a test case or scenario might serve two purposes: first, it describes a test and as such is designed to find (specification) defects. Second, it represents an example of how the system is expected to behave and thus serves as a specification (by example). The idea that system behaviour can be specified using examples is also represented by *"Specification By Example"*, 2011 [3]. Therefore, testing methods on higher test levels, especially acceptance testing, should ideally facilitate the expression of test scenarios but also, to some extent, requirements. Combined with a high degree of automation, the evaluated acceptance criteria represent an "executable specification" (see [26, p. 195ff.]). However, based on theoretical considerations it is not decidable whether a program meets its specification. Thus no automation can exist that answers this question in general.

As emphasized above, testing should not be conducted by developers. This is even more important if test cases serve as specification because system requirements are typically expressed by experts of the domain. When the acceptance testing process is automated then the test case must be interpretable and executable by a machine. This means that test cases must be programs or scripts themselves. These considerations make it clear that it is not an easy task to find

a trade-off between flexibility for the test author, usability for domain experts (i.e. non-developers) and executability by a testing engine.

Fewster and Graham provide an excellent foundation on test automation for software systems in their book *"Software Test Automation"*, 1999 [20]. The authors suggest four quality attributes for test cases: *efficient*, *evolvable*, *effective* and *exemplary* [20, p. 5]. Efficiency describes how much effort is essential to perform a test and evolvability describes how much effort is required to maintain a test and adjust it to new situations. Both quality attributes, efficiency, and evolvability, influence how economic testing is. The effectiveness of a test describes how likely it is to find defects and a test is exemplary if it represents use cases and is understandable (see Figure 2.2).

The main motivation for test automation is that automated test cases can be more economic than manual testing. On the one hand, automation has the potential to increase efficiency given that test cases are executed frequently to compensate for the overhead that is associated with the automation. On the other hand, automating tests tends to render them less evolvable and maintainable compared to manually executed tests. The properties of effectiveness and exemplariness of test cases cannot be influenced by automation. Any method that automates test execution should therefore be evaluated based on how efficient and evolvable the automated test cases are.

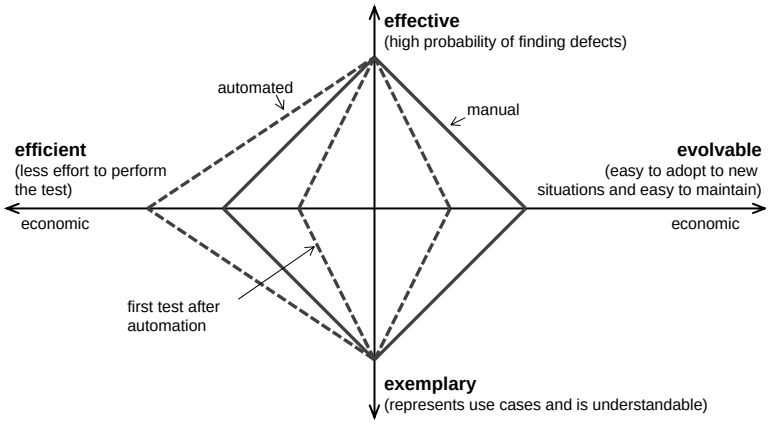


Figure 2.2.: The quality of a software test can be determined using four quality attributes. Thus a good software test is effective, exemplary, efficient, and evolvable. Manual testing (solid lines) is compared with automated testing (dashed lines). The larger the covered area the higher the testing quality. The figure is adopted from [20].

According to [20], other promises of test automation, apart from economic aspects, include: minimal effort to execute regression tests, running more tests more often, load and performance testing, better usage of testing resources (i.e. avoiding repetitive manual test execution), consistency and repeatability and increased confidence in the system under test.

According to [20, p. 17f.], the testing process can be broken down into five distinctive activities: *identify*, *design*, *build*, *execute*, *check*. The first two activities (i.e. *identify* and *design*) are considered to be "intellectual" as they govern the quality of tests and are repeated only rarely. In contrast, the two last activities (i.e. *execute* and *check*) are considered "clerical" as they are repeated many times. Clerical activities are those that can benefit most from automation, whereas the intellectual ones remain manual tasks in most cases. The *build* activity lies in-between the

intellectual and clerical activities.

The most obvious approach to test automation would be to record tests while they are conducted manually and then later on execute these recorded tests with a machine. Such capture-replay tests are tempting because they do not require any additional effort for automation, yet almost all sources advise avoiding this style of testing. Reasons include that a test might become dependent on details of the interaction with the system under test [26, p. 191f.] and that manual tests are very different from automated tests [3, p. 142f.]. Finally [20, p. 26ff.] dedicates an entire chapter to explaining why capture-replay test is neither an efficient nor an effective approach for test automation. The three main reasons are that ad hoc manual tests are often ineffective, that only input is automated (not the validation of the output), and that it is inefficient to re-record tests when the system under test changes. The authors summarized their observations in the brief and concise phrase: *automated testing is not the same as automating tests*.

Once testing is automated, it fits well into automated integration and delivery processes. A high degree of automation is a prerequisite for continuous integration [19] and continuous delivery [26]. Particularly when software is released often, it is not sufficient if only tests from the lower test level (i.e. unit and module tests) are conducted; acceptance criteria must also be checked [26, p. 123]. This leaves no other choice but to automate tests from higher test levels, in particular acceptance testing.

2.2. Distributed Systems

The method of test automation that is proposed in this thesis is focused on tests for distributed systems. A general definition of a distributed system is provided by the text book *"Distributed Systems. Principles and Paradigms"*, 2006 [42]: "A distributed system can be defined as a collection of computers that appears to its users as a single and coherent system". Based on that definition, examples of distributed systems include distributed *computing* systems (high performance systems, grid computing applications, cloud computing), distributed *information* systems (transaction-based systems, enterprise application integration) and distributed *pervasive* systems (ubiquitous computing systems, mobile computing systems, sensor and actuator networks). Architectural styles of distributed systems can be grouped into two main categories: centralized architecture (e.g. client-server model, n-tire), and decentralized architecture (e.g. peer-to-peer).

In [42] four common design goals of distributed systems are mentioned: *accessibility of resources*, *transparency*, *openness* and *scalability*. These design goals already give hints to the additional complexity that is introduced when distributing a system. For instance, if transparency is aimed for then the system should hide the fact that it is distributed, which results in a tendency of distributed systems to obfuscate the source of defects (e.g. a node that is temporarily unavailable). Openness also introduces a whole class of potential issues because components might be added or replaced at any time. And finally scalability requires a distributed system to perform well not only on a fixed scale but also in substantially larger or smaller deployments. The *"Fallacies of Distributed Computing"* [15] list some assumptions in the context of distributed systems that

one might be tempted to make but that all prove to be false in a long-term perspective. Table 2.2 summarizes these. These fallacies make it clear that the development of a distributed system introduces a whole array of additional challenges and pitfalls.

No.	Fallacy	No.	Fallacy
F1	The network is reliable.	F5	Latency is zero.
F2	The network is secure.	F6	Bandwidth is infinite.
F3	The network is homogeneous.	F7	Transport cost is zero.
F4	Topology does not change.	F8	There is one administrator.

Table 2.2.: The *"Fallacies of Distributed Computing"* [15] (F1-F8) summarize assumptions that turn out to be false in the long term.

The behaviour of a complex (distributed) software system sometimes appears to be unpredictable from its individual components. The interactions of components as well as interactions with the environment of the system might lead to "emergent" behaviour that is not explicitly specified. Likewise, unwanted behaviour (i.e. misbehaviour) might emerge (see [32]). An alternative definition of a distributed system stresses this point: "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."[1].

2.2.1. Testing in the Presence of Concurrency

In the most abstract conception every distributed system consists of two or more independent processes that exchange messages via communication channels. Therefore, a distributed system is also a concurrent system that uses messages for synchronization.

Thus a test automation method for distributed systems must account for the fact that multiple events might occur in parallel. A test author might have to express parallel execution explicitly or implicitly within test cases. For instance, one might want to examine a distributed database system that continues to successfully process writing operations, *while* one of the replication nodes is subject to a crash failure. How to program such parallel systems effectively is an on-going subject of research. An Advanced Grant of the European Research Council[2] was recently awarded to a project that addresses the "Popular Parallel Programming" challenge, which demonstrates the significance of the topic.

In some cases, concurrent processes might cause a (partially) non-deterministic execution of the distributed system. As a conclusion test cases for distributed systems must account for potentially non-deterministic test execution. In practice, this means that a test case might sometimes pass and sometimes fail, although the input remains identical. This raises the question of how results of test assertions can be reproducible when testing a non-deterministic system. Ideally, failure and success should always be reproducible; otherwise it becomes very difficult to draw any conclusions from a test result.

[1] Leslie Lamport, eMail from 1987, `http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt`, accessed May 30, 2013

[2] ERC-2010-AdG, PE6, "Domain-optimised parallelisation by polymorphic language embeddings and rewritings"

Another aspect inherent to distributed systems is that different, potentially contradictory, views exist within the processes of the system. For instance, in a distributed workflow engine some nodes might be aware that a workflow has terminated, whereas others are perceiving the workflow as being still executed. This is commonly known as the problem to reach agreement in a distributed system. Four main conditions influence the ability of a distributed system to reach agreement: 1) processes are synchronous or asynchronous, 2) messages are ordered or unordered, 3) communication time is bounded by a maximum or not, 4) transmission is point-to-point or multicast [48]. As soon as processes are asynchronous and messages are unordered it is impossible to guarantee that agreement will be reached under every circumstance. Most distributed systems must operate under such conditions. One way to make consensus possible in these cases is to introduce an ordered and reliable message delivery (e.g. TCP). Nevertheless, testing must account for different views within the system. This becomes particularly apparent when conditions must be checked to determine success or failure of a test. On lower test levels (i.e. unit tests) it might be sufficient to evaluate a single condition, but when testing distributed systems this condition might produce a different evaluation on every node. For instance, a test case for a distributed workflow engine should only pass if every node is of the opinion that the workflow was executed successfully. In the case of pervasive distributed systems it might even be necessary to express that a given threshold of agreement must be reached (e.g. 90% of the nodes must succeed).

A conclusion from these considerations on agreement is that timeouts cannot be avoided in an asynchronous system with unbound message transmission. For instance, it cannot be determined whether a node that does not respond to messages has crashed or is simply very busy and might respond if one just waits long enough. For a test case this means that failure or success cannot only be determined by a Boolean condition that is evaluated (potentially on multiple nodes) but also from a state where the condition has been unknown until a maximum amount of time has passed (i.e. timeout). For efficient test automation it is obvious that such timeouts should be avoided as much as possible.

2.2.2. Testing of Fault-Tolerant Systems

Many distributed systems are designed to be fault-tolerant to some extent, which implies that the system can continue to operate properly even when one or more of its components fail. A common failure model categories failures as follows [42]: 1) crash failures where a node terminates unexpectedly, 2) omission failures where messages get lost, 3) timing failures where a node might not respond before a given time threshold, 4) response failures where a node might send unexpected or wrong messages, and 5) arbitrary failures where a node might produce arbitrary messages. For instance, the Transport Control Protocol (TCP) is able to handle lost messages, which is an example of an omission failure. The subtle difference between a fault and a failure becomes clear from the following definitions: a *failure* occurs when the system is not able to perform its required function. A failure might be caused by a *fault*, which is an abnormal condition [42].

An interesting question for any distributed system is whether it provides the degree of fault

tolerance that is anticipated by its specification. In order to test tolerance to failures a distributed system could be deployed in a target infrastructure and the corresponding failure scenario would have to be established. This might require fine-grained control over the individual processes (nodes) and their communication among each other. After the system under test is subjected to the failure scenario an assessment must be made of whether the system has dealt with the failure according to its specification. If it is possible to express the degree of fault tolerance that is expected from a system as acceptance criteria then acceptance testing can be used for this evaluation. Another benefit would be that test cases might also serve as examples of conditions under which a distributed system is known to operate properly. It is obviously advantageous to have a high degree of automation in the execution of acceptance tests.

Once testing is automated on the acceptance testing level the technique of reverting the development and testing process ("test first" or test-driven development), which is commonly applied on lower test levels, becomes applicable to acceptance testing.

2.3. Virtual Infrastructure

Virtualization of computational resources such as CPU cycles, block storage, or IP-level connectivity has made great progress in recent years [35]. A number of software stacks are currently available that enable the operation and automation of virtual infrastructure. This is complemented by commercial providers that sell infrastructure on a "pay-as-you-go" basis (e.g. Infrastructure as a Service). In a Berkeley survey from 2009 [4] the authors go as far as to argue that the "long-held dream of utility computing" is currently becoming reality. Technologies and advances in the area of infrastructure, platform, and software as a services are summarized under the collective term "*cloud computing*". The NIST defines cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [31]. Very similar to this definition is that of Infrastructure as a Service (IaaS): "a standardized, highly automated offering, where computing resources, complemented by storage and networking capabilities, are owned by a service provider and offered to the customer on demand. The resources are scalable and elastic in near-real-time, and metered by use." [30].

Test automation for distributed systems can particularly benefit from the freedom provided by infrastructure virtualization, as it potentially allows full automation of infrastructure creation, test execution, and analysis of the results. Ideally, virtualization technologies can be leveraged to enable the test author to specify properties of the infrastructure as part of the test case. For instance, one might want to investigate the behaviour of a system in a scenario, where a node acts as a bottleneck that limits data throughput. Enabling such a testing environment would help to gain valuable insights into the properties of a distributed system as well as its behaviour under different conditions (e.g. crash failure). Instead of "carefully predicting the conditions in which a system will operate" [22], robustness could be ensured in diverse and heterogeneous environments through

direct testing and evaluation.

2.3.1. Elasticity of Computing Resources

A major benefit of virtualized infrastructure is the elasticity of computation, storage, and networking resources that results from the virtualization. Resources can be dynamically added and removed based on the current demands, making the infrastructure particularly appealing for systems where utilization varies greatly over time. This often occurs in automated test execution as part of the quality assurance in the software development process. During the test phase of a project, execution of test suites may require substantial computing resources while, in phases where no tests are conducted, the infrastructure is left unused. In [43] it is pointed out that, in practice, this usage pattern leads to repurposing of testing hardware to production hardware, making the infrastructure unavailable for future testing campaigns. Thus, the varying resource demands for testing can benefit from the elasticity that virtual infrastructure offered as a service can provide.

Table 2.3 gives an impression of the current pricing level for virtual computation resources (i.e. VMs). An on-demand instance of the type *micro* can be provisioned for 0.02 USD per hour when using Amazon Web Services (AWS). Resources can be provisioned and released at any time. The user is charged for every started computing hour but no further commitments or contracts are necessary. It makes no difference in price, if one instance is used for 100 hours or 100 instances are used for one hour.

Machine Type[3]	On-Demand (USD/hour)	Spot (USD/hour)	$1 - \frac{\text{Spot}}{\text{On-Demand}}$
Micro (smallest)	0.02	0.003	0.85
Small (standard)	0.06	0.007	0.88
Medium	0.12	0.013	0.89
Large	0.24	0.026	0.91
Extra Large (largest)	0.48	0.052	0.91

Table 2.3.: Table of prices of AWS standard instances (VMs) with Linux/UNIX as operating system. The table shows the difference in price between AWS on-demand and spot instances. The ratio between both illustrates how much cheaper spot instances are (prices as of February 25, 2013 10:50 AM).

On-demand instances are complemented by AWS *spot instances* which have a price that is calculated in real time based on supply and demand of computing resources. A spot instance can be requested by bidding a price per hour that one is willing to pay. Once the current price drops below the bid price, the request will be fulfilled and the instances are provisioned. If the current price exceeds the bid price then instances might be terminated without further ado. Spot instances are attractive because the price can be 85-91% lower than the on-demand price (see Table 2.3). If test execution were fully automated, supply/demand-based price models could be harnessed postponing test execution to a time window where prices are comparably low (e.g. at night).

2.3.2. Enabling Technology

Virtualization technologies such as virtual machines (VMs), virtual overlay networks, and software for virtual machine management have made great advances and gained wide popularity in recent years. Abstracting actual hardware components through a virtualization layer that emulates physical components provides flexibility that is especially useful in the context of cloud computing where it is a key objective to provide rapid and flexible provisioning of computational resources [31]. Disadvantages that are often associated with virtualization (e.g. generally decreased performance, additional overhead) are becoming less relevant as virtualization technology matures [35].

The central component that acts as an abstraction layer between physical resources and virtual machines is the hypervisor or virtual machine monitor (VMM). A hypervisor provides low level management for virtual machine provisioning such as starting and stopping individual VMs and assigning physical resources to VMs. Common hypervisor software includes Xen [6], KVM [24], and VMware ESXi. On top of these different hypervisors a great variety of proprietary solutions have been developed to manage virtual, shared computational resources together with networking and storage. In addition to proprietary solutions a number of open source frameworks are currently being developed. Examples include Eucalyptus [35], OpenStack [44], and OpenNebula [46].

Virtual infrastructure can be used to manage resources in private datacentres more flexibly (i.e. private cloud) or to make them publicly available in a “pay-as-you-go” model (i.e. public cloud). The latter is commonly referred to as Infrastructure as a Service, emphasising the ability to allocate and de-allocate infrastructure on demand. IaaS is the lowest layer in the often proposed three layered architecture of cloud computing: Infrastructure, Platform, and Software as a Service (IaaS, PaaS, and SaaS). Examples of commercial providers that offer IaaS in a public cloud are Amazon EC2 [1], Rackspace [47], and GoGrid [45] among many others. A shortcoming of most services is the lack of standardization and interoperability. Although IaaS frameworks face very similar problems centred around resource provisioning, APIs are often very different from each other. If applications are built on top of native APIs this effectively leads to a vendor lock-in. One approach to counter this issue is to use an abstraction layer that provides a common interface for multiple APIs. The jClouds [27] project tries to offer a “multi cloud” API that allows the migration of Java-based code between different software stacks and service providers without any changes. Currently the project allows the most common tasks of infrastructure management to be carried out (e.g. provisioning of machines, storage access) using a common API and provides vendor-specific extensions to access functionality that is unique to a service provider.

3. Related Work

This chapter introduces related work in the area of test automation for distributed systems. First, model based approaches are discussed that either model the system under test or its environment, which allows to predict the behaviour of the system under test. Second, the difference between remote and distributed testing is discussed and existing approaches are presented. Third, acceptance testing frameworks are discussed.

3.1. Simulation-Based Testing

An often proposed approach is to model a system prior to its implementation. The model can then be used to simulate the system in order to predict its behaviour at run time. In the context of distributed systems, modelling can either be applied for simulation of the system under test or the environment that the system under test interacts with.

3.1.1. Simulation of the System Under Test

An example of an approach that makes it possible to model component-based, distributed systems is the Palladio Component Model (PCM) [7, 8] which promises early insights into extra-functional properties like the performance (i.e. response time, throughput, resource utilisation) and reliability of a system (i.e. mean time to failure, probability of failure on demand). Individual components are modelled based on their specification in order to be simulated. Input values of components are modelled as probability distributions. Quality of Service (QoS) properties of the overall system are then predicted from the specifications of the individual components. This approach allows quick feedback cycles in early design phases as proposed in model driven development approaches. However, it has a number of limitations when predicting the behaviour of existing, distributed systems.

First, the method assumes that every component is fully specified. To specify a component, internal details such as branch conditions, number of loop iterations as a function of input size, and resource demands must be known. For instance if a file based input should be modelled then a designer must provide internal details such as the number of CPU operations per byte of input data [8, p. 12]. Second, despite the effort of specifying many details of every component, mathematical modelling requires a number of simplifying assumptions that often do not hold in real world scenarios. Noteworthy, in this context, is the assumption of stochastically independent resource demands. This might exclude sources of defects that are rooted in the completion of different system components for limited resources.

Apart from these two limitations the authors of [8] also mention the following: a) the architecture is static, meaning that components do not move to different hardware b) components are stateless, meaning that the execution solely depends on the input and that the runtime environment does not hold any state c) memory management of multiple components are not regarded and d) the support for concurrency is limited as the authors agree that predicting QoS properties in a concurrent system is difficult especially in multi-core architectures that become more and more common. Finally, PCM focuses on QoS attributes of a system, but other important extra-functional properties like fault-tolerance and resilience to disruptive influences are not targeted.

An extensive discussion of approaches that apply modelling and simulation of software evaluation (i.e. system execution modelling) is beyond scope of this work. However, strength and shortcomings of the PCM are exemplary for this type of system verification and validation.

3.1.2. Simulation of the Environment

Other model based approaches simulate the environment to which the system will eventually be deployed instead of simulating the system under test itself. An example of such an approach is the GridSim [12] toolkit that is a Java-based discrete-event Grid modelling and simulation framework. It allows to simulating applications in different grid computing environments. In order to be able to simulate a grid-like environment, heterogeneous types of resources must be modelled together with their non-functional capabilities and availability patterns. If resources, allocation policies and usage patterns are modelled correctly then the toolkit can be used to evaluate algorithms that operate in heterogeneous grid computing environments. A limiting factor of the project is the narrow focus on grid computing that represents only one type of distributed system, and the observation of the developers of the toolkit that modelling is challenging and requires experience with the framework.

Recent advances in virtualization technology make application architectures possible that take advantage of the on-demand availability of computation and storage resources. To model these kinds of characteristics the CloudSim [13] project pursues the activities of the GridSim project. The tookit extends GridSim with capabilities that allow to model cloud management environments more closely (e.g. on-demand virtualization enabled resources and application management). System developers can use the toolkit to test performance of provisioning and service delivery policies without an actual deployment to a virtualized infrastructure. The authors emphasize the possibility of a controllable and repeatable evaluation of a system that is free of charge because no actual resources are allocated. But this consideration does not take the effort for modelling and expert training into account.

The DIECAST project [23] is a methodology for evaluation complex distributed large-scale applications that use multiple different network services. The goal is to provide a testing environment that accurately reflects the – potentially very large – target system using as little resources as possible. Efficient usage of resources is achieved by multiplexing several components of the original infrastructure on fewer physical machines using virtualization. This allows to study large-scale scenarios even with little resource consumption while maintaining the relevant characteristics of

the involved resources. The authors report that a scaling factor of up to 10 can be achieved which would allow to simulate a scale of 1000 machines using only 100 physical machines. A key concept of the DIECAST methodology is the idea of trading time for other resources. The system components (e.g. VMs) are manipulated in order to perceive time slower than real-time (e.g. the operating system might be fooled to believe that only one second passes for every ten seconds in real time). Physical events are not influenced by this "time dilation" mechanism making them appear faster to the system components. An example of this is data that is arriving on a network interface that appears to have a rate of 100MBps although the physical rate is only 10MBps, given a time dilation factor of 10 is used.

Another approach to provide a software testing environment for distributed systems is D-Cloud [41]. The project also recognizes the need to test a distributed software system at the same scale as productive environments. Instead of simulating large-scale systems, the testing environment allows to inject faults that would otherwise only occur reasonably frequently in very large systems. The main concept of the framework is to express reproducible failure scenarios that can easily be authored using XML. Fault injections that can be triggered include corruption of specified hard drive sectors, 1bit and 2bit errors of network packages, and bit errors in the main memory. The implemented framework uses QEMU as virtualization software and the open-source cloud management software Eucalyptus.

Both, DIECAST and D-Cloud, assure that a distributed system is fault-tolerant and works reliable in different environments, which is indispensable for systems that are expected to scale well. The idea of injecting faults in order to assure fault-tolerance has also been suggested by many other authors [28, 29, 39].

3.2. Remote and Distributed Testing

It is useful to make a distinction between remote testing and distributed testing. Remote testing uses multiple remote machines to execute tests or to execute algorithms for software evaluation. The focus is on reducing the overall execution time. Frameworks that support remote testing might themselves be distributed systems, but the system under test is not necessarily a distributed system. Contrary, distributed testing focuses on testing distributed systems by deploying them to a possible target infrastructure. Then aspects of the system under test are exercised. For instance, a distributed database might be tested in order to evaluate its ability to maintain distributed consistency under a variety of interesting conditions.

3.2.1. Remote Testing

Remote testing can be used to scale a testing technique horizontally using multiple remote machines. One application where remote testing is attractive is the execution of test suites that contain a large number of test cases (e.g. unit tests). The individual test cases of a test suite are often entirely independent of each other by design. This makes it easy to parallelize the test execution process by scheduling test cases to remote resources. Another promising application

of remote testing is the field of automatic failure search. Techniques for automated failure search typically explore parts of the state space of an application in order to detect faults. The main challenge such techniques face is that the state space increases exponentially with the program size. In such cases it is particularly interesting to parallelize the algorithm and migrate the execution to multiple remote machines, because the extent to which faults can be detected is typically limited by the computational resources that are available.

Remote Parallel Execution of Tests

The key advantage of remote testing is the achieved speedup by executing independent tests on multiple machines in parallel. The total time required for execution of a test suite can be considerably reduced if the load is balanced across multiple machines. In order to allow parallel test execution all tests must be independent from each other (e.g. no state is carried from one test to the other). Tests from lower test levels (i.e. unit, module) are often well suited for parallel test execution as they are small and isolated by their nature. Additionally, the time required for execution is easier to predict for tests from lower test levels.

An example of a remote testing framework is distributed JUnit execution using Apache Hadoop. The Hadoop developers were confronted with the need to test their productive code quickly in order to get rapid feedback for developers. To meet this requirement a framework was developed that parellelizes unit test execution on the Hadoop platform itself [37]. The authors report that they were able to reduce the execution time by a factor of 30 using a 150-node cluster. The framework allows executing JUnit-based unit tests concurrently in order to reduce the overall testing time, but is not focused on testing aspects that are specific to distributed system (i.e. distributed testing).

Another example of remote testing is the GridUnit [17] project that extends the JUnit test framework in order to allow remote testing. The execution of JUnit-based tests is scheduled onto nodes in a computational grid in order to transparently distribute the execution. The authors report that they achieved good speedup values as expected for a problem that is easy to paralellize.

Remote Parallel Automated Failure Search

Automated failure search strategies rely on the exploration of the exponentially large state space of the system under test. A major advantage, compared to automated execution of test cases, is that the intellectual activity of designing and building test cases is not necessary. Many different algorithms are conceivable that use different search strategies in order to scan the state space. A technique that is frequently applied is to prune the search space based on policies that detect irrelevant or uninteresting execution paths. Obviously, parallelization of such algorithms is less trivial than the execution of independent test cases, but the potential benefits might justify the effort.

The York Extensible Testing Infrastructure (YETI) is an example of a framework that automates the failure search for Java or .NET applications. The failure search strategy is to execute the application by inserting random values as method parameters. Exceptions that are thrown by the system under test are assessed to determine their correspondence to a failure. In [36] it is

described how this failure search strategy can be parallelized and then be deployed on Amazon's Elastic Compute Cloud (EC2). The result is a reduced execution time through parallelization. However, the parallelization strategy assumes that classes can be tested in isolation in order to allow an easy scheduling of classes to computing resources. According to the authors, this assumption limits the number of defects that can be found.

Another research project that fits into the same category is Cloud 9 [14]. The employed algorithm searches the application using symbolic execution. Instead of using regular input for failure search, as done by many random failure search strategies, symbolic execution introduces a symbol or variable that is used as input. At the beginning, the content of the variable is unconstrained and will be only constrained if the program execution encounters a branching condition that depends on the value of that variable. The branching execution paths are then followed with a constrained range of possible values. If a defect is detected, the collected constrains can be resolved into specific input parameters for the system under test.

Like the YETI project Cloud 9 uses cloud computing resources in order to execute a parallelized version of the search algorithm. Corresponding to the authors, the main challenges for a scalable algorithm are the balanced partitioning of the search space, a search strategy that requires no coordination, and the prevention of redundant work and memory usage. The reported speedup that is achieved by a prototypical implementation of the parallel algorithm is on average 47 times faster using a 16-node cluster. The reason for the superlinear speedup (>16) is an increased probability of failure detection through parallelization.

The high degree of automation and the deployment of test execution onto virtual infrastructure establish the basis for software testing as a service. The pay-as-you-go price model of virtual infrastructure can be extended to the testing of software. Cloud 9 offers its failure search strategy using a service-based model. The interface to the service requires that the system under test is uploaded in binary form or as source code, that the testing goal is specified (e.g. coverage threshold), and that a resource provisioning policy is selected (e.g. number of nodes).

3.2.2. Distributed Testing

Methods that focus on testing characteristic properties of distributed systems using actual implementations of the system under test in realistic environments can be summarized under the term "distributed testing". In this context, the term "distributed unit testing" is used very inconsistently. The definition of unit testing limits the subject of the test to a small and isolated piece (unit) of code. Such a unit is typically a single class or method excluding its dependencies. In the context of distributed systems it is less clear how the term "unit" should be interpreted. The authors of "Defining unit testing of distributed systems" [5] introduce a categorization of different types of "distributed unit tests" based on the analysis of a number of frameworks that carry "unit" in their product name. Especially the introduced category called "End-to-end testing with integration tests" contradicts most of the common definitions of unit testing. One reason why the definition of distributed unit testing is blurred might be the observation that execution engines of unit testing frameworks (e.g. JUnit or NUnit) are often reused in frameworks that automate higher level tests

(e.g. the Cucumber acceptance test framework or the DisUnit framework).

A publication highlighting the need for distributed testing from an engineering perspective was published by the developers of the Chubby [11] software. Chubby is a fault-tolerant database system that provides a distributed locking mechanism using an implementation of the Paxos distributed consensus algorithm. The authors give valuable insights into the challenges they faced during the implementation. One observation is that fault-tolerant systems naturally mask problems and that no tools or methods are available that disclose defects or misconfiguration hidden by tolerance mechanisms. The authors claim that formal validation and proof of correctness of short algorithms do not scale to an implementation that is typically composed of thousands of lines of code. Furthermore, the authors argue that the failure models of fault-tolerant algorithms often do not represent the variety of failures that occur in real systems. An implementation of a distributed system must handle a wider array of failures. Finally, the authors emphasize that testing is a key ingredient for the development of distributed, fault-tolerant systems and claim that additional attention should be paid to research on testing methodologies.

Distributed Testing Frameworks

PNUnit[1] (parallel NUnit) is an extension of the Microsoft .NET unit testing framework NUnit that provides some basic functionality for testing distributed systems. It is based on agents that have to be installed on every remote node. The test configuration and infrastructure description is defined in an XML file. Test scenarios can be described using C# and a small API that provides simple synchronization mechanisms (e.g. barrier) in order to control the concurrent test execution flow.

BizUnit[2] is another "framework for automated testing of distributed systems" that is specialized on testing Microsoft's BizTalk Server. Test scenarios are described in XML files that contain a number of test steps. These steps are executed sequentially by an orchestrator and might communicate with remote services (e.g. Microsoft Message Queuing instance) but the concept does not provide test agents that are deployed into the environment. Tests are written against an existing productive infrastructure instead of a dedicated testing infrastructure.

DisUnit [40] is a tool for distributed testing that extends JUnit and permits writing test scenarios as Java code. The authors of the publication on DisUnit do not state whether the framework is general enough to test non-Java based distributed systems. Also support for concurrent execution of test scenarios is not mentioned.

An example of a project that focuses on non-functional testing of component-based distributed systems is UNITE [25]: Non-functional Intentions via Testing and Experimentation. Low level performance metrics that are derived from log files are gathered and related to high level non-functional requirements (overall latency, system reliability and security, or end-to-end response time). The intention of assessing non-functional requirements of distributed systems is very similar to the PCM that is discussed above, but UNITE uses the implementation of the system under test for quality assessment instead of a model.

[1] `http://www.nunit.org/index.php?p=pnunit&r=2.5`, accessed March 13, 2013
[2] `http://bizunit.codeplex.com/`, accessed March 13, 2013

A different approach to distributed testing and their behaviour under failure, is the PREFAIL programmable tool for multiple-failure injections [28] that recognizes the exponentially large space of failure combinations that arise in a distributed system and suggests a technique to guide the pruning of the failure space. It is designed to test large-scale distributed systems like the Hadoop Distributed File System (HDFS) or the Cassandra database system. The key idea is to use failure-injection in order to evaluate the system under test in multiple failure scenarios based on policies that the test author can specify. This is motivated by the observation that despite efforts to implement reliable and fault-tolerant systems many of the implemented mechanisms (e.g. recovery mechanisms) are buggy. Remarkably, the developers of the recovery mechanism of the Hadoop File System had to deal with 91 issues within four years.

Apart from the aforementioned frameworks, many other attempts for automated testing of distributed systems exist. Most of them have in common that they are very focused on a specific type of distributed system and that the documentation is very sparse or does not exist. Most work on test automation for distributed systems – including the frameworks investigated in [5] – focuses on higher test levels such as integration, system, and acceptance tests instead of unit and module tests. At the same time individual components tend to be treated as grey- or black-boxes rather than white-boxes (see Figure 2.1).

3.3. Acceptance Testing Frameworks

A number of frameworks exist that are centred around automated testing on the acceptance testing level (see test levels). The terminology in the field of acceptance testing is very inconsistent. The terms behaviour-driven development, automated acceptance testing, acceptance test driven development, executable specification or specification by example are often used to refer to the idea that domain experts express acceptance criteria as use case examples that can be automatically validated. Many solutions in this field aim to bridge the communication gap between domain experts and developers. The following sections briefly introduce some these frameworks.

3.3.1. Framework for Integrated Test (Fit)

The Framework for Integrated Test (Fit) is a popular acceptance testing framework [33]. Fit allows domain experts to specify their requirements using a natural, business understandable language. Requirements are expressed by the customers using examples that are complemented with input values and their corresponding expected output values. Input and output values are formatted as tables (called *decision tables*). Every example is translated into an automated black-box test for the system under test which must be implemented by a developer. The connection between decision tables and the system under test is realized as a Java class (called *fixture*). The FitNesse[3] framework uses Fit to provide a convenient environment for acceptance testing. FitNesse contains a web server and a pre-installed wiki that can be used to write requirements and decision

[3] `http://www.fitnesse.org`, accessed March 29, 2013

tables. The content of the wiki can be executed automatically at any time to assert that the system under test meets its requirements.

3.3.2. The Gherkin Specification Language

Similar to Fit, that uses tables to describe acceptance criteria, a number of tools use a simple specification language called Gherkin. The language helps to express requirements based on scenarios. Every scenario is composed of a number of steps that either describe preconditions (Given), actions/operations (When) or the desired output (Then) [50, p. 25ff.]. This simple specification language allows formulating requirements on the system under test as expected behaviour. Figure 3.1 shows a generic scenario that exemplifies how the Gherkin specification language can be used to express requirements.

```
Given [a certain situation]
When  [some interaction with the system under test is carried out]
Then  [some kind of output or observable behaviour is expected]
```

Figure 3.1.: An example how requirements to a system can be expressed as a scenario using a simple grammar (i.e. Gherkin) that is based on three parts: Given, When and Then.

Figure 3.2 gives a more complete example of the Cucumber tool [50]. The system under test is an imaginary calculator that is expected to be capable of adding two numbers, which is expressed as a feature. The feature is exemplified using a scenario that is composed of multiple, sequential steps. Every line of code corresponds to one step and must begin with one of the three possible keywords. The scenario accepts parameters that can be specified by the test author with concrete values. These example values are similar to the decision tables that are used in the Fit framework. The figure shows three sets of parameters that are used to execute the scenario three times in a row. The feature will be accepted if the test succeeds for all parameter sets. In that respect, scenarios together with parameter values can server as acceptance criteria. Cucumber allows to use multiple test scenario to exemplify one feature. The Gherkin specification language emphasizes the specification by example methodology and aims to provide an executable specification that is understandable by domain experts. Because the requirements to the system under test are expressed as expected behaviour the approach is also known as behaviour-driven development.

Next to the Cucumber tool a number of other frameworks exist for different platforms. Most noteworthy are RSpec[4] that provides an internal Ruby domain-specific language, JBehave[5] that is specialized on the Java platform and SpecFlow[6] that can be used in Microsoft .NET projects. All frameworks use a similar, behaviour-centred specification language and are focused on interaction with a graphical user interface (GUI). The GUI is used to provide input to the software and to assert the expected behaviour. Alternatively to the GUI a command-line interface (CLI) can be used to provide input.

[4] http://www.rspec.info/, accessed March 29, 2013
[5] http://www.jbehave.org/, accessed March 29, 2013
[6] http://www.specflow.org/, accessed March 29, 2013

```
Feature: Addition
    Add two numbers using the calculator

Scenario Outline: Add two numbers
    Given <input1> is entered into the calculator
    And <input2> is entered into the calculator
    When <button> is pressed
    Then the result should be <output> on the screen

Examples:
    | input1 | input2 | button | output |
    | 0      | 7      | add    | 7      |
    | 42     | 8      | add    | 50     |
    | 9      | 9      | add    | 18     |
```

Figure 3.2.: This example shows a feature that is expected to be implemented by the system under test. The feature is exercised using an example scenario that accepts parameters. If the scenario executes successfully using the specified parameters then this indicates that the feature is implemented properly.

3.4. Conclusion

This section discusses some model based approaches for software testing that rely on modelling the system under test or its environment. While useful for an early evaluation, model-based simulation suffers from some limitations in the context of distributed systems. On the one hand, accurate mathematical modelling of complex distributed systems can be very difficult and does not resemble real world scenarios close enough. On the other hand, lack of information or incomplete understanding of the system requires designers to make assumptions or guesses (e.g. interactions between loosely coupled system components). For those reasons, this work focuses on testing the actual implementation of a specified system and employs virtual on-demand infrastructures as environment for experimentation and testing.

As pointed out, virtual infrastructure can be harnessed for test automation in two fundamentally different ways: remote testing and distributed testing. Remote testing is concerned with efficient and fast execution of tests using virtual infrastructure, distributed testing uses virtual infrastructure to execute tests that require the system under test to be deployed on a target environment. This thesis focuses on the latter kind of testing in order to suggest a method of testing for distributed systems that has a high degree of automation. Some of the existing approaches to automated distributed testing are presented in this section. Those approaches are often very specialized on a certain platform or product and do not address conceptual questions to a sufficient extent. For instance, how the inherent concurrency in distributed systems can be represented using test scenarios.

In this thesis it is argued that the concepts of existing acceptance testing frameworks like FitNesse or Cucumber can serve as a starting point for the development of a test automation method for distributed systems. Especially simple specification languages similar to the Gherkin grammar can serve as a model for testing of distributed systems. But also the concept of scenario-based fault injection as applied the D-Cloud project is picked up in order to test under a variety of interesting conditions. Finally, the concept of low level non-functional performance metrics as foundation for the evaluation of high level requirements, as employed the UNITE project, is developed further to automate validation of acceptance criteria.

4. Acceptance Testing for Distributed Systems

This chapter introduces the suggested method that automates the testing process for distributed systems using virtual infrastructure. It is structured into four main sections. First conceptual foundations are introduced in Section 4.1 that describe general requirements together with considerations on main design decisions. Then, in Section 4.2, the suggested task-oriented model, that serves as the foundation for the test automation method, is described in detail. This is followed by Section 4.3 introducing a domain-specific language that allows to express test scenarios for distributed systems that are both automatically executable and understandable by domain experts. Finally, the fourth Section 4.4 continues with a discussion of the implementation details using a reference architecture for acceptance testing.

4.1. Conceptual Foundations

This section first discusses key requirements that were gathered for automation of acceptance tests for distributed system. Afterwards aspects that influence main design decisions are discussed.

4.1.1. Key Requirements

The introduction chapter discusses foundations on software testing, test automation, and distributed systems. These fundamentals are now used to derive requirements for a method that aims to automate software testing for distributed systems. The requirements are justified using findings of a number of book authors. Also the ten “vital principles for program testing” (see Table 2.1) as well as the “fallacies of distributed computing” (see Table 2.2) will be referenced. In the evaluation chapter Section 5.1 uses these requirements to derive evaluation criteria for the suggested method.

Quality of Tests

The method should facilitate the creation of automated tests that have high quality. To further define what a high quality means, the four introduced quality attributes of tests can be used: effective, efficient, evolvable, and exemplary (see Figure 2.2). First, tests should be *effective* which means that they should focus on finding defects instead of trying to show their absence

(principle M8 and B2). In order to be effective, tests should be adoptable to scan for additional defects using similar tests. This is promising because it is likely that additional defects are present in a component that is already known to contain defects (see principle M9 and B6). Second, for the purpose of being economic, automation must be *efficient* and *evolvable*. From efficiency follows that the execution of tests should be considerably faster compared to manual testing. From evolvability and maintainability follows that the negative impact of automation should be minimized compared to manual testing. Maintainability can be achieved through tests that are decomposable into smaller parts (modules) that are general enough to be reused in other tests and by facilitating the analysis and debugging of tests. Third, tests should be *exemplary* which means that they should be understandable by domain experts and should represent complete use case so that a test is an example of how the system should be used.

Reproducibility of Tests

Automation of test execution should have a positive effect on reproducibility of the test results. This is necessary in order to be able to derive useful conclusions from test success or failure. When tests are integrated in the software development life cycle (principle B10) and are executed using a continuous integration server then it is particularly important that test success and failure is reproducible.

A test is reproducible if the test result (success or failure) is determined only by the input parameters that the test accepts (e.g. test size, build/version of the system under test). Put differently, a test must either always succeed or always fail. If this definition is followed strictly, it turns out to be useless in practice, because no test can be 100% reproducible. The execution of a test relies on underlying hardware components that have failure rates (e.g. mean time between failures of hard drives) potentially causing a test that appears reproducible to fail in very rare cases. To simplify the discussion, tests will be characterized as reproducible if their result depends only on input parameters in the large majority of the executions.

The two catalogues of software testing principles (see Table 2.1) do not contain any explicit reference to reproducibility, although principle B8 mentions repeatability. Reproducibility of test execution results are particularly difficult to achieve when testing distributed systems indicating that acceptance testing for distributed systems is not within the focus of both catalogues.

One source for unstable test results in the context of distributed system is the environment that the system under test is deployed on. This environment must be well-defined and ideally stateless (e.g. no memory leakage, no temporary files) so that the execution of tests has no permanent influence on the environment. Side effects that are caused by repeated test execution must be avoided.

Another source for unstable test execution is that the execution of the system under test itself might be entirely non-deterministic with respect to the provided input values. The source for potential non-determinism is the concurrent communicating processes. Unpredictable latency or bandwidth in the network might cause the same test case to succeed or fail although the same input parameters were used. To counter this issue, the test author must be supported with

mechanisms to author tests that are sufficiently reproducible.

Support Distributed Systems

When testing distributed systems the focus should be on finding defects that are common in distributed systems. A promising starting point are the fallacies of distributed computing that summarize implicit assumptions that a developer of a distributed system might be tempted to make (e.g. homogeneous network, static topologies or transport cost is zero). Testing should help to reveal defects that stem from these false assumptions.

A second source of potential defects can be derived from properties that are characteristic for distributed systems. One on those is the ability of a distributed system to tolerate (partial) failure for the system to a certain degree which also referred to as graceful degradation. A well-known example of fault tolerance is the two phase commit protocol (2PC) that is implemented by many transaction-based database systems. It allows tolerating network partitioning while ensuring a strong consistency model. Fault tolerance is a requirement that is frequent in distributed systems and should thus be testable. Tests that exercise fault tolerance mechanisms promise to uncover defects that might otherwise remain unnoticed because they occur very rarely or are difficult to reproduce (failure transparency). When a test exemplifies a scenario in which the system under test handles a fault successfully then this might help to document or specify the degree of fault tolerance that a system offers. Another reason why a distributed system must be fault-tolerant is that it must cope with failures in the infrastructure that become increasingly frequent in larger scales. The more nodes or processes are involved in a distributed system the higher becomes the probability of a failure:

$$P(\text{a failure in the distributed system}) = 1 - P(\text{node not failing})^{\text{number of nodes}}$$

Mainly two choices exist when investigating the fault tolerance properties of a system under test. Either the test is conducted at a scale where these failures become frequent, or the probability of a failure is artificially increased to simulate larger scales. The latter is also known as fault injection and promises effective and efficient testing of distributed systems. An example of fault injection is causing a crash of one or more nodes (i.e. crash failure) or manipulating values in communication channels (i.e. response failure). Injecting failures also has the advantage that is reproducible. In order to avoid any side effects, testing itself should influence or alter the environment of the test as little as possible, unless this is explicitly stated within the test case. The same is true for the execution of the system under test itself. It should only be influenced by the method if this is stated in the test case.

In order to be generally applicable, no assumptions should be made on how the system under test works (e.g. runtime environment, programming language), nor on the characteristics of the infrastructure (e.g. protocols, operating systems). Finally, the testing concepts must scale at least as good as the software that it should test. More preferably, scaling up should not influence the test execution notably. If this is achieved, then issues that occur at larger scales can be directly

attributed to the system under test.

Enable Domain Experts

Testing should enable domain experts (non-developers) to create and perform tests. One reason for this is that developers should not test their own product (see principles M2, M3 and B7). Another reason is that domain experts are familiar with the requirements and might additionally detect specification defects. Thus, test automation must be simple enough to be understandable by non-programmers but should not limit the flexibility and facilitate the "intellectually challenging" activities of identifying, designing, and building tests (see Figure 4.1).

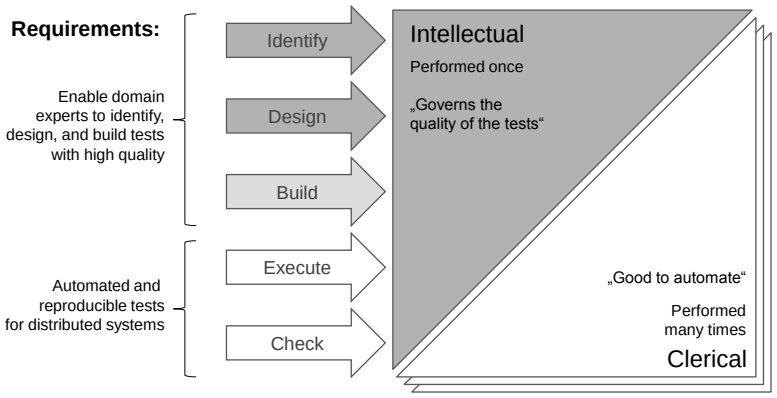


Figure 4.1.: The five testing activities identify, design, build, execute, and check. On the left hand side are the key requirements assigned to the five testing activities. Based on [20, p. 17f.].

It is not planned to automate these activities. If tests from higher testing levels are exemplary enough, then they might be assignable to features of the software and serve as an example of how to exercise the system. This might facilitate the communication between domain experts who know the requirements and system developers who implement the requirements.

4.1.2. Considerations on the Design

Some initial considerations are discussed in this section providing a framework for design decisions, before the task-oriented model of distributed testing is introduced in the next section.

Testability on Higher Test Levels

Automated tests for distributed systems are usually tests from higher levels of the software testing spectrum (see Section 2.1.1). On higher test levels internal details of a component or the entire system under test become less relevant. At the same time test cases are concerned with what the system should do (i.e. what the requirements are) rather than how the system does it (i.e. how the system is implemented). Apart from that, an approach that is general enough to support tests for arbitrary distributed systems is not allowed to make any assumptions on the internals of the system under test or its runtime environment.

These considerations make it clear that automated tests for distributed systems have to model the system under test as black- or grey-box. Thus, what can be tested is limited by the input that the system under test accepts and the output that it provides (Figure 4.2). For example, if the system under test does not provide output on whether an internal process has succeeded then a test cannot use this as a condition for test success or failure. The IEEE defines testability accordingly as “the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met”.

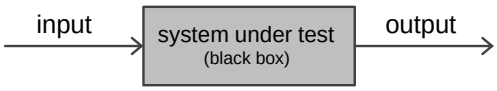


Figure 4.2.: Model of a black-box test: the system under test is executed given a well-defined input. After the test execution the output of the system under test can be used to evaluate the test.

On the acceptance testing level it is common to provide input to the system under test using graphical user interfaces (GUI). Selenium[1] is an example of an acceptance testing framework that uses the GUI – in this case websites – to interact with the system under test. As an alternative to the GUI, the command-line interface (CLI) can be used to provide input. A CLI might be provided by the system under test to either pass parameters when the application is invoked or to interact with the system under test at run time (interactive command-line interface). The standard output stream or log files might be used as the output of the system under test.

Test Orchestration versus Choreography

A framework that implements a method for distributed testing is likely to be a distributed system itself. Hence, it must be decided which high-level architectural style is appropriate for a distributed testing method. Architectural styles of distributed systems can be categorized into centralized or decentralized architectures. Centralized architectures include, for example, the common client-server model whereas examples of decentralized architectures include peer-to-peer systems. Another classification refers to the form of control of the execution of the distributed system. When the execution of a distributed system is controlled by a central entity the execution is classified as an orchestration. An example of an orchestration is the Business Process Execution Language (BPEL) that allows defining business processes within a distributed system. When no central entity controls the execution within a distributed system then this is referred to as a choreography. Most protocols are examples of choreographies. Orchestrated systems usually have a centralized architecture and systems that are executed as a choreography often have a decentralized architecture.

When the activity of testing distributed systems should be automated then a fundamental conceptual question is whether an orchestration or a choreography is more suitable for this task. Using a choreography would require every node or process in a distributed system to agree on a protocol that specifies valid behaviours. The behaviour of the overall system, which emerges

[1] `http://docs.seleniumhq.org/`, accessed May 28, 2013

of the choreography of multiple nodes, might be very hard to predict and a central description of the overall behaviour is not available. In contrast, an orchestrated system is controlled by one central node. In this case the overall process or workflow is well-defined from the perspective of the orchestrator and the orchestrating node knows which part of an overall process is currently being executed.

From this discussion it becomes obvious that it is favourable to use an orchestration for automated tests of distributed systems rather than a choreography. The main reason is that test cases should be directly expressible and understandable by humans. An orchestrated execution suggests a centralized architecture style for a framework that automates tests for distributed systems.

From Scenarios via Criteria to Features

Often tests are expressed as test cases or test scenarios. Although no sharp distinction between test cases and test scenarios exists, the notion of a *scenario* fits to tests on higher test levels better. These tests tend to be more verbose and also contain detailed descriptions of preconditions that must be fulfilled. Therefore, tests are often described as scenarios and the term test scenario will be used for the remaining text.

$\mathrm{F1}$	The software under test must be interoperable between different versions.
$\mathrm{AC1}$	A distributed computation workflow should **be** possible if one node uses version **1.9** and the other node uses version **2.0** of the system under test.
$\mathrm{AC2}$	A distributed computation workflow should **not be** possible if one node uses version **0.9** and the other node uses version **2.0** of the system under test.
$\mathrm{TS1}$	A distributed computation workflow should **[be|not be]** possible if one node uses version **[String]** and the other node uses version **[String]** of the system under test.

Table 4.1.: Feature $\mathrm{F1}$ is exemplified using two acceptance criteria ($\mathrm{AC1}$ and $\mathrm{AC2}$). Both acceptance criteria use Test Scenario $\mathrm{TS1}$. Additional acceptance criteria and test scenarios could be used to further exemplify and test the feature.

Test scenarios can be used to express acceptance criteria. A test scenario accepts a number of parameters that influence relevant parts of the test execution. Examples of such parameters are the version number of the system under test that should be used or the number of nodes that should be used for the test execution. A test scenario, together with concrete values as parameters, represents an acceptance criterion. One test scenario might be used to express multiple acceptance criteria which only differ in the parameters that are passed to the scenario. Table 4.1 shows the two Acceptance Criteria $\mathrm{AC1}$ and $\mathrm{AC2}$ that are both based on Test Scenario $\mathrm{TS1}$. In addition to acceptance criteria that describe what the system should be capable of, criteria can also be used to describe what the system should *not* be capable of. The testing principle M6 also suggests including negative examples when testing (see Table 2.1).

Acceptance criteria can be used to connect test scenarios with features. Features describe what a system should be capable of and represent, in that respect, requirements to the system. One or more acceptance criteria can be used as an example of how to exercise the feature. Table 4.1 uses the two Acceptance Criteria $\mathrm{AC1}$ and $\mathrm{AC2}$ to exemplify the Feature $\mathrm{F1}$. If all acceptance

criteria that are related to a feature are executed successfully then this feature can be assumed to be properly implemented. But because the correctness of the implementation cannot be proven, it has little value to express criteria with the intention of demonstrating the absence of defects. In the agile software development process, the number of features that have been tested by at least one test scenario, are used to measure progress in the development process. This metric is referred to as ”running tested features” (RTF).

4.2. Task-Oriented Model

A distributed system is commonly modelled as a number of concurrent processes that communicate with each other using communication channels. Strictly seen this model does not require the processes to be distributed across different machines (nodes) although in practice this is often the case. To simplify matters in the following sections it is assumed that every process of a distributed system is assigned to a separate node or, more formal, the function that maps processes to nodes is bijective. Thus the terms “process” and “node” are in most cases interchangeable.

4.2.1. Test Scenarios for Distributed Systems

Some approaches exist that use test scenarios as examples of features. When test scenarios are expressed as scripts using a programming language then they can be automatically executed and evaluated. Executable scenarios can help to achieve two important goals of test automation: first, automation becomes trivial if scenarios are directly executable and, second, scenarios can be expressed by domain experts that are typically non-developers. It is of little use if test execution would be fully automated but the test scenarios cannot be understood by non-developers. This is the reason why most approaches limit themselves to a very simple grammar and use verbose textual descriptions that translate into interactions with the system under test.

A major limitation of all scenario-based approaches discussed in Chapter 3 is that scenarios are sequential descriptions (e.g. steps) which are executed top to bottom. Therefore, concurrent execution cannot be expressed by the test author. For example, the following outline of a small scenario uses concurrency: while task A is executing in the background, continue with task B and C, but wait until A is finished, before continuing with E. If the system under test is a concurrent (distributed) system then test scenarios must allow the test author to express concurrency to a certain extent. Another limitation of existing methods is that generally only simple conditions are evaluated in order to determine if a scenario succeeded. These conditions evaluate either to true or false similar to assertions that are common in low level tests. When testing distributed systems then this approach is too simple, because every process might evaluate a test condition differently. Some processes might come to the conclusion that the test execution was successful, while others do not. Test scenarios for distributed systems should allow handling these “distributed” assertions explicitly by the test author.

As soon as concurrency is expressible by the test author, a whole array of potential issues arises that are difficult to debug (e.g. race conditions, dead locks, etc.). Therefore, the complexity

that is associated with concurrency must be hidden through abstraction, as much as possible, without limiting flexibility of the test author. In addition, the abstractions of concurrent execution must allow to reason about the execution of the test scenario (i.e. its run-time behaviour). Only if this is possible then tools can support the test author with hints to potential errors within the test scenario. An example of this are dead locks that should be detectable by a tool before the test is executed. Otherwise such pitfalls in a distributed system will be very hard to spot. From this discussion it becomes clear that it is not easy to develop a method that is flexible enough to express test scenarios for distributed systems and usable by domain experts.

Tasks as Abstraction of Concurrency

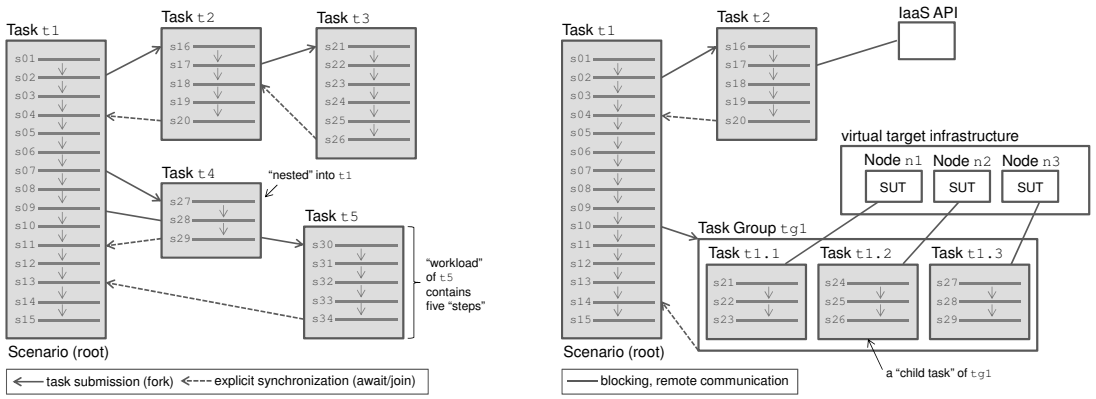
A possible approach to abstract concurrent execution is to model tests as sets of tasks that can be executed by an execution service or a scheduler component. To control the execution of tasks, it makes sense to express dependencies between tasks. A task can be only started if every task that it depends on is completed. If task A depends on task B and task B, in turn, depends on task C, then task A also depends on task C. Thus, dependencies are transitive relations. If two tasks do not depend on each other then they can potentially be executed in parallel.

This is a very fundamental task model that has its applications in many areas of computer science (e.g. resolving transitive module dependencies), but it does not allow to represent some situations required when expressing test scenarios. For instance, it might be necessary to express that, at some point, it should be guaranteed that a task was started. Also, stating dependencies only at the beginning of tasks might not be sufficient. Instead it should be possible to express dependencies during the course of a task. For instance, a task might proceed with some work before it awaits the termination of another task.

In order to achieve this flexibility the task-oriented approach can be combined with the join/fork model that is also used to create new processes from existing ones in Unix-like operating systems. Every task is composed of a number of steps that are executed sequentially and atomic. A test scenario can then be represented as one single root task as shown in Figure 4.3(a). Within that root task it is possible to submit (fork) new tasks at any point using a step. Every started task forks of a parallel execution thread. The execution thread that represents the task execution can be synchronized (or joined) at any point in time *after* it has been started. The figure shows how the root task $\mathrm{t1}$ starts a new task $\mathrm{t2}$ in step two $\mathrm{s02}$ and afterwards the newly spawned task $\mathrm{t2}$ starts another task $\mathrm{t3}$ in step $\mathrm{s17}$. A task that is started by another task stands in a child-parent relation to that task. A child task will be referred to as nested task from here on because its execution is nested into the execution of the parent task.

The first situation where a concurrent execution is started is in step $\mathrm{s02}$. This means that the following steps $\mathrm{s03}$ and $\mathrm{s16}$ can, but do not have to, run in parallel. The execution order is not defined. The next step in the root task, $\mathrm{s04}$, waits until $\mathrm{t2}$ is completed, therefore it is guaranteed that $\mathrm{s16}$ until $\mathrm{s20}$ are completed. Note that, on the one hand, in step $\mathrm{s10}$ it is guaranteed that both tasks $\mathrm{t4}$ and $\mathrm{t5}$ are already running, but, on the other hand, in step $\mathrm{s03}$ it is possible that task $\mathrm{t3}$ has *not* started because step $\mathrm{s17}$ has not been reached. From step $\mathrm{s05}$ on it is assured that $\mathrm{t3}$

has finished because t2 has finished.



(a) Example of tasks and task nesting.

(b) Example of remote communication and task grouping.

Figure 4.3.: A sketch of a task-oriented model. The test scenario is represented as task t1 in both figures. Starting and awaiting of tasks is represented as dashed arrows. Figure 4.3(a) demonstrates how a test scenario can start and await tasks. Figure 4.3(b) shows how tasks can be used to abstract remote communication (solid arrows) with third party systems and the target infrastructure for the test.

Such a task-oriented model of a test scenario can then be represented as a directed graph with steps as vertices and dependencies as edges. Based on the dependency graph the execution semantics can be analysed and can reveal, for instance, cyclic dependencies.

Tasks as Abstraction of Remote Communication

The concept of tasks provides a tool to encapsulate requests to third party systems. For example, provisioning or destroying nodes in the target infrastructure using the API of an IaaS provider. This is shown in Figure 4.3(b), where task t2 executes a blocking, remote call to an IaaS provider API in s17, while t1 can continue to execute. Another advantage of this task model is that it can be used to orchestrate the system under test on the remote infrastructure during the course of a test scenario. A task can encapsulate an operation on a remote node of the target infrastructure. For instance, the deployment of the system under test can be expressed as a task. If the system under test must be installed on multiple nodes, which is a likely scenario, then this can be expressed using one task per node. These tasks execute the same, or very similar, operations on different nodes. In this case, it would be tedious to create all these tasks by hand. A more productive approach is to group similar tasks together and execute the whole group of tasks in parallel, given that no dependencies between the tasks exist. Figure 4.3(b) depicts how the task group tg1 can be used to further abstract concurrency and remote communication that is needed to manage the system under test in the target infrastructure.

4.2.2. The Semantic Model

The concepts described in this section represent the semantic model that aims to further formalize the ideas of the previous sections. It introduces the semantics of tasks, nested tasks, node groups, task groups, and retryable assertions. In [21, chapter 11] such a semantic model is

characterized as a requirement for the development of a domain-specific language because it serves as a “domain model that is populated by a domain-specific language”.

Task Lifecycle

It is useful to distinguish four states in the lifecycle of a task as shown in Figure 4.4: *instantiated*, *submitted*, *started* and *finished*. Every task must have a definition of its workload that describes what the task should do. The definition is similar to a class in object oriented programming and can be used to create instances of that description. An instance can then be submitted to an execution service or scheduler component. From that point on, the task is scheduled for execution, but it might take some time until the task starts. Once the task is started, it is executing its defined workload until the last step of the workload has been completed. Then the task transitions into the state last state: finished. Every instance of a task can be only submitted once. In order to “repeat” a task, a new instance of the task description must be created. For simplicity, task definitions and task instances are not distinguished if the difference is not of relevance in the corresponding context.

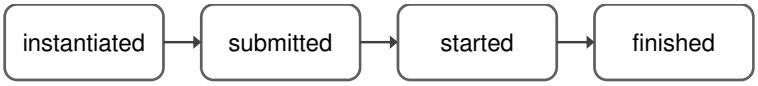


Figure 4.4.: The four states that a task can be in: instantiated, submitted, started, and finished. State transitions are only allowed in one direction. Once a task is finished, it stays in that state.

Tasks and Nested Tasks

A *task* is a unit of work (workload) that is executed (or started) asynchronously. The program flow continues directly after the task has been submitted to an executor service[2]. Thus parallel task execution is the default, whilst synchronization must be stated explicitly. An example where both, parallel execution and explicit synchronization, is needed would be the following scenario: one node has to be configured as a server (task A) and the other node as a client (task B), before the system under test is executed (task C) . Both configuration tasks, A and B, can be executed in parallel, because they are independent, but task C must wait until task A and B are finished.

Every task can return a result after its execution has finished. The type of the result is generic which is expressed by the type parameters in Figure 4.5. An example of a simple task result type would be a $\mathrm{Boolean}$ that indicates whether a condition is given on a remote node or not (e.g. existence of a certain file). Another example of a result type would be a String that contains, for instance, the standard output of a SSH session that was used for remote communication. The concept of statically typed results is very similar to the Java future[3] interface.

The *workload* of a task describes what has to be done in order to complete the task. It is specified as a single method that can contain arbitrary code and that is called when the task is started. The term *step*, that was previously used to describe the inner construction of a task,

[2] fully qualified name: `java.util.concurrent.ExecutorService`
[3] fully qualified name: `java.util.concurrent.Future`

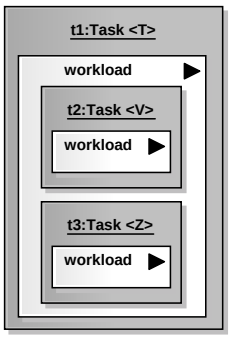


Figure 4.5.: An UML object diagram that shows two tasks ($\mathrm{t2}$ and $\mathrm{t3}$) that are nested into a parent task ($\mathrm{t1}$). The figure formalizes the ideas from Figure 4.3(a). The workload contains all steps that a task is composed of and every task has a return type: X, Y and Z.

can be interpreted as code statements in the workload. Within a workload, arbitrary additional *nested tasks* can be started that can potentially have different result types (X, Y and Z in Figure 4.5). The nesting level of tasks is not limited. Making use of nested tasks allows structuring and modularizing tasks. If side effects are avoided in task descriptions they can be reused in multiple test scenarios. It is, for instance, useful to encapsulate the initialization of a test scenario into a task that might internally coordinate multiple other tasks (e.g. “install software”, “install software”, “run software”). The nesting level and ordering of tasks does not reveal anything about the time when tasks are finished. Both nested tasks in Figure 4.5 ($\mathrm{t2}$, $\mathrm{t3}$) can potentially complete long after the outer task ($\mathrm{t1}$) finished its execution, because every task is executed asynchronously per default. The order in which tasks and nested tasks are completed solely depends on explicit synchronizations stated by the test author. How this is achieved is discussed in Section 4.3 which introduces a language that is translated into this task-oriented model. The interface of a task is described in Figure 4.7. It ensures that an implementation provides methods to submit the task ($\mathrm{submit}()$), to await its termination ($\mathrm{await}()$), to fetch the result of a task ($\mathrm{unwrap}()$), to test if a task is finished ($\mathrm{done}()$) and to return an identifier of the node or service that the task was used for ($\mathrm{node}()$).

Node and Task Groups

The concept of nodes is fundamental to distributed systems as every process is executed on a node. A reoccurring pattern in test scenarios descriptions is that tasks are applied to a group of nodes. Node groups might be defined by selected subsets of nodes based on their properties (e.g. operating system) or based on roles that stem from the domain of the system under test (e.g. “master”, “slave”, “client”, “server” or “peer”). For the purpose of be flexible when describing test scenarios it should be easy to perform common set operations on node groups like union or intersection. Additionally, convenience methods such as selecting a random member or splitting the group into subgroups based on a fixed ratio should be possible.

The concept of task groups and node groups can be used to manage similar tasks that must be applied to multiple nodes. Figure 4.3(b) visualizes this idea. Applying a task to a task group can be described more precisely as a one-to-one mapping (bijective function) of tasks (members

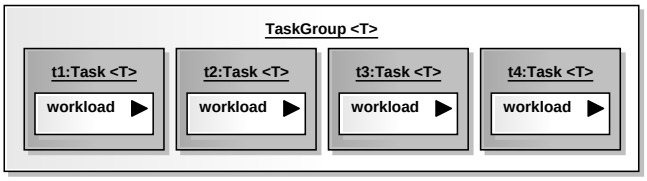


Figure 4.6.: An UML object diagram that shows a task group. The figure formalizes the ideas from Figure 4.3(b). The task group is composed of four child tasks ($\mathrm{t1}$, $\mathrm{t2}$, $\mathrm{t3}$ and $\mathrm{t4}$) that all have the same return type as the task group (T). The task group itself has no workload but adds high level functionality.

of a task group) to nodes (members of a node group). An example of this would be a scenario, where log files must be downloaded from every node. Creating individual tasks for every node would be tedious in such cases, but using a single task that iterates over the members of a node group would be inefficient (sequential). To counter the problem, *task groups* can be used as an abstraction that manages tasks (Figure 4.6).

Task groups are similar to tasks which is ensured by the $\mathrm{Awaitable}$ interface that both extend. It guarantees that both can be submitted and awaited and are in that respect indistinguishable (see diagram in Figure 4.7). In addition, task groups extend the interface $\mathrm{Iterable}$[4] to allow iteration over the child tasks, provide a method that allows to await termination of a subset of all child tasks ($\mathrm{await\,(int)}$), a method querying the child tasks that are already finished ($\mathrm{finished\,()}$), a method that returns a collection of all return values of the child tasks ($\mathrm{unwrapAll()}$) and a method that returns a list of node identifies that identify the remote node of the corresponding tasks ($\mathrm{nodes()}$).

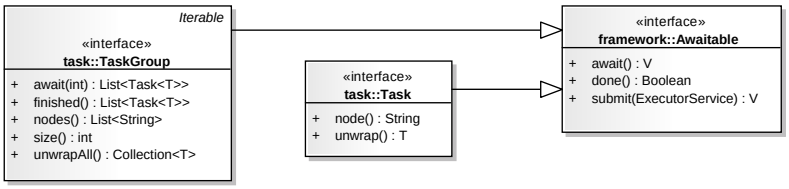


Figure 4.7.: UML class diagram that shows interfaces for task groups ($\mathrm{TaskGroup}$), tasks (Task). Both inherit from a common interface that generalizes the concept of starting (submit) and awaiting the execution ($\mathrm{Awaitable}$).

Task groups introduce a high level abstraction of the execution of their child task that makes test scenarios more expressive and leaves the details of the implementation to libraries on lower levels. This allows to express scenarios, where the test scenario only continues if, for instance, 10% of the child tasks in a task group have completed their workload. Task groups themselves have no workload and are finished once the last child task has finished. The concept of task groups is very similar to the “apply” method, often found in functional programming languages which allows to apply an operation to each element of an array. Noteworthy, all child tasks of a task group must have the same return type as the task group itself. In that respect task groups are similar to generic lists.

The main difference between nested tasks (Figure 4.5) and task groups (Figure 4.6) is that, in the case of nested tasks, the ordering combined with explicit synchronization allows to define the execution order of a test scenario while, in the case of task groups, the test author should not be

[4] fully qualified name: `java.lang.Iterable`

concerned with how the execution is managed.

4.2.3. Assertions in Distributed Systems

A common approach in software testing is verification by comparison. Given a well-defined input for the system under test, the expected output is compared to the actual output. In [20, p. 52f.] comparisons are categorized into dynamic comparators that are evaluated during the runtime of a test and post-execution comparators that are evaluated after the test execution finished. In this context the term *comparator* is very similar to the term *assertion* (or test assertion) which is today more commonly used in literature on software testing. Thus, it will be used in the course of this work as well. In a distributed, concurrent system it is especially useful to distinguish *dynamic assertions* and *post-execution assertions*.

In contrast to a centralized system, where a comparison is either true or false, in a distributed system every node (or process) has its own (potentially contradicting) view on a condition. An example of this is a database system that replicates its content across multiple, distributed nodes. If one node is subject to a crash failure it might take a considerable amount of time before every node is aware of the crash. In the meantime different views on the global state may exist on different nodes. This implies that the question whether all database replicas are currently available, might be answered differently by some nodes. In the worst case nodes might not agree at all on one global state considering that reaching agreement in a distributed system is far from trivial (see Section 2.2.1).

Utilizing Task Groups

The concepts of tasks and task groups, which were introduced above, can be used to provide dynamic assertions for distributed systems. A test assertion can be realized as a task group that has a Boolean return type. It contains one child task for every node that is part of the distributed assertion. The workload of each child task is then designed to test a condition on the node. Such a condition could be, for example, that a certain file should exist or should contain (or not contain) a given substring. When the task group finishes its execution (i.e. all child tasks are done) then an array of Boolean values is returned that represent the individual assertion. The overall success or failure of the distributed assertion must be derived from this list by reducing the multiple Boolean values to a single Boolean value. The most obvious way to reduce an assertion is using a logical and conjunction when all individual nodes are expected to pass the assertion. Other possibilities include a minimum threshold for the number of individual assertions that must pass (e.g. at least 80% of all individual assertions should pass).

Retryable Assertions

When authoring test scenarios for distributed systems the need for synchronization arises naturally. For example one might want to execute some test logic only after the system under test finished starting up on all nodes. A rather unsuitable approach would be, to wait an experimentally

determined period of time and then proceed. This would be very unreliable and would render the overall time required for test execution a useless measurement. A better approach is to use a dynamic assertion and retry the assertion after a short period of time until the dynamic assertion passes or a maximum number of retries is reached. If the assertion does not evaluate to true, after the maximum number of retries, this is interpreted as failure. From a theoretical point of view, the usage of timeouts cannot be entirely avoided (see properties of distributed systems in Section 2.2.1) indicating that this approach is a valid compromise.

Listing 4.1: A distributed, retryable test assertion on three nodes that succeeds after five retries.

```
1  Retryable assertion: 'One MASTER; rest SLAVE' (retry 1/10, passed: 0/3, nodes: 3)
2  Retryable assertion: 'One MASTER; rest SLAVE' (retry 2/10, passed: 1/3, nodes: 3)
3  Retryable assertion: 'One MASTER; rest SLAVE' (retry 3/10, passed: 1/3, nodes: 3)
4  Retryable assertion: 'One MASTER; rest SLAVE' (retry 4/10, passed: 2/3, nodes: 3)
5  Retryable assertion: 'One MASTER; rest SLAVE' (retry 5/10, passed: 3/3, nodes: 3)
6  SUCCESSFUL (finished pulling on 3 node(s) for 5 (interval: 3000) times)
```

The log file shown in Listing 4.1 illustrates how a distributed retryable assertion on three nodes works. In this example, the three nodes are part of a master/slave database replication set. This global state can be expressed using a distributed assertion which asserts that each node is aware of one master and $n - 1$ slaves. The log file shows that it takes five retries until agreement on the global state has been reached. Once the retryable assertion succeeds the test scenario might continue.

4.3. Task-Oriented Language

The previous section introduced a task-oriented domain model that is now applied to provide a domain-specific language. The language allows expressing task-oriented test scenarios that are represented using the concepts of the task-oriented model. Using a language instead of a framework or library has several advantages: a language is a clean abstraction that adds notation to the conceptual model, it limits the potential sources of semantic errors by introducing additional constrains compared to a general-purpose language and offers the opportunity for static analysis of the program that is expressed in the language [49, p. 30f.].

4.3.1. A Domain-Specific Approach

Benefits of using a domain-specific language include productivity, quality, validation, and verification, data longevity, domain expert involvement, productive tooling, platform independent-dence/isolation [49, p. 40ff.]. Many of these advantages match the key requirements that were collected for the suggested method in Section 4.1.1. On the one hand, the quality of tests, in particular the quality attributes efficiency, exemplary, and evolvablility, can profit from the promised benefits of a DSL. On the other hand, domain experts can be enabled to author test scenarios and use the language as a tool for communication and thinking.

Choosing a Host Language

Domain-specific languages can be categorized into *internal* (or *embedded*) and *external* DSLs. An internal DSL is embedded into a host language that is usually a GPL. The distinction between internal DSL and an API is not always clear. Internal DSLs do usually not require a grammar, hence, constrains or properties of programs cannot be checked based on a grammar. The host languages of internal DSLs are often statically typed but the embedded languages are dynamically typed to offer more flexibility. An advantage of internal DSLs is that less initial effort is required for the development. This facilitates to improve and refine the language continuously over many iterations with minimal effort. Because of these considerations, using an internal DSLs is favourable. The development of an external DSL could be subject of future work that is out of scope of this thesis.

The host language should integrate seamlessly into the Java platform and should be executable on the JVM. Other important aspects include the capability to run new or changed test scenarios without the need to recompile code, mature IDE support (e.g. Eclipse and Maven), and support for functional programming.

From the JVM languages that exist today Groovy fits best to these requirements. Groovy is an object-oriented, dynamic language for the JVM that has been inspired by other languages like Python, Ruby or Smalltalk. Plugins exist in stable versions for Eclipse (Groovy-Eclipse[5]) and Maven making the language ready to use in a Java environment. Groovy code is compiled to Java bytecode which can be directly executed on the JVM. Most Java programs are also syntactically correct Groovy programs and existing Java classes and interfaces can be directly accessed using Groovy. Furthermore, Groovy can be used as dynamic scripting language which allows loading and executing code at run time. Functional programming style is supported using the $\mathrm{Closure}$[6] interface that extends the two Java interfaces $\mathrm{Callable}$[7] and $\mathrm{Runnable}$[8].

The type system of the Groovy language implements strong and dynamic typing. Thus type-based constraints are checked at run time. On the one hand this increases the flexibility for the programmer but, on the other hand, it can be a disadvantage when invalid programs should be discarded early. Current versions (2.0 or higher) of the language offer optional static typing based on annotations (i.e. $\mathrm{TypeChecked}$[9]) that can be used to activate static type checking on method or class level.

A Brief Introduction to the Groovy Syntax

The syntax of the Groovy language is different to the Java syntax in some aspects. However, most Java syntax is also valid Groovy syntax easing the migration from a Java to a Groovy code base. Most obvious differences are that Groovy allows omitting semicolons and brackets where they can be inferred. Another difference is that anonymous functions can be defined using only curly

[5] `marketplace.eclipse.org/content/groovy-eclipse-juno`, accessed April 4, 2013, version 2.7.1
[6] fully qualified name: `groovy.lang.Closure`
[7] fully qualified name: `java.util.concurrent.Callable`
[8] fully qualified name: `java.lang.Runnable`
[9] fully qualified name: `groovy.transform.TypeChecked`

braces syntax. These differences make Groovy more expressive and less verbose than Java.

Listing 4.2: Java code example (also valid Groovy syntax)

```
1 myMethod("hello world", new Callable<Boolean>() {
2     public Boolean call() { return true; }
3 });
```

The Java code in Listing 4.2 shows a method that accepts one parameter of the type String and another one of the type $\mathrm{Callable}<\mathrm{Boolean}>$.

Listing 4.3: Groovy code example

```
1 myMethod 'hello world', { true }
```

Listing 4.3 is equivalent to the above Java example but omits brackets and semicolons and uses a closure that returns a $\mathrm{Boolean}$ (the last expression is interpreted as implicit return statement).

4.3.2. Execution Semantics

Section 4.2 introduced the fundamental concept of tasks and task groups. This section will discuss how tasks can be defined and executed. The definition of tasks describes what a task should do. It is the sequence of statements that a task is composed of. The definition itself does not execute the task. This happens when a defined task is submitted to an execution service. From that point on the task is scheduled for execution but execution must not necessarily start directly. The task finishes when the last statement in the task definition has been executed (see task lifecycle in Figure 4.4).

Asynchronous Tasks

The default execution semantic of a task is asynchronous and non-blocking which will be referred to as *asynchronous task*. In order to execute an asynchronous task it must be submitted to an execution service[10] that can be realized using a thread pool.

On the language level the async statement submits an instance of an asynchronous task definition. More generally the async method can be used to submit anything that implements the $\mathrm{Awaitable}$ interface (see Figure 4.7) which includes also task groups. Listing 4.4 is an example of an asynchronous tasks instantiation and submission. The curly braces syntax is used in the Groovy language to define an anonymous function or more precisely a closure. The closure is evaluated when the test scenario reaches this line of code causing a task instantiation. This is achieved by converting the closure into a task with the closure as workload. Tasks that are derived from closures have no return type. The print statement is a method call that simply prints a string on the screen.

Listing 4.4: Submitting an asynchronous task without label

```
1 async { print 'Hello World! I am an anonymous task.' }
```

[10] fully qualified name: `java.util.concurrent.ExecutorService`

Listing 4.5 also defines an asynchronous task but adds a label (i.e. 't1') that can be used to reference the task later on. A label must be unique within a test scenario and must not be reused. This simplifies static analysis which is discussed in the next section. Once an asynchronous task is submitted the scenario execution continues instantaneously without any guarantee that the task execution has already started. At some point it might be necessary to ensure that a task is completed before the scenario continues. This can be achieved by explicitly synchronizing the execution. Listing 4.5 uses an await statement for synchronization which blocks the scenario execution until a given task is finished.

Listing 4.5: An asynchronous task execution followed by a synchronizing statement

```
1 async 't1', { print 'This is a task labeld as "t1".' }
2 await 't1'
```

Using a task-based approach to abstract parallel programming was also recently introduced to C# and VisualBasic[11]. Both languages provide first-class language support for asynchronous task execution and synchronization with similar execution semantics.

Synchronous Tasks

A frequent pattern that can be observed in distributed test scenarios is an asynchronous task that is directly followed by a blocking statement that awaits the termination of this task. Such a situation occurs in the previous Listing 4.5. The execution semantics of these two statements together are equal to the notion of a *synchronous task* that blocks execution until the task is done. Therefore, it makes sense to provide direct support for synchronous tasks within the language. Listing 4.6 shows a synchronous task that has the same execution semantics as both statements in Listing 4.5. Internally, a synchronous task is translated into an asynchronous task that is directly followed by a blocking API call.

Listing 4.6: A synchronous task execution

```
1 sync 't1', { print 'This is a task labeld as "t1".' }
```

The task label of synchronous tasks – in this example 't1' – can be omitted in most cases, because no further synchronization and thus referencing is required.

Task Libraries

The previous sections give examples of how asynchronous and synchronous tasks can be defined within a test scenario. Alternatively task definitions can be migrated to a task library. A task library is an instance of a class that serves as a collection of methods which return instances of tasks or task groups. Task libraries are handy when the number of written test scenarios for a system under test increases because it helps to reduce duplicated code. Listing 4.7 submits an asynchronous task that is loaded (i.e. instantiated) from a library named common that contains

[11] http://msdn.microsoft.com/library/vstudio/hh191443, accessed May 7, 2013

task definitions that are general enough to be reused for many systems (e.g. uploading a file to a node). Thereafter, the second statement submits another task instance that is loaded from a library called domain containing tasks that are specific to the domain of the system under test (e.g. configuring nodes as clients and servers). The library methods that return task instances can accept parameters in order to influence the task description. This is not shown here but will be used in later examples.

Listing 4.7: Task can be loaded from a library

```
1 async common.mockTask()  /* Create general task (Task<T>) */
2 async domain.mockTask()  /* Create domain specific task (Task<T>) */
```

In the following examples task libraries that provide general tasks will we denoted as common and task libraries that contain domain-specific tasks will be denoted as domain although libraries can generally have arbitrary names. Note that tasks from libraries can be executed synchronously or asynchronously.

Controlling Execution Order

Synchronous and asynchronous tasks can be used to gain fine-grained control on the execution of a test scenario. A test scenario can be compared to a workflow in order to describe the execution semantics. The paper [2] summarizes general workflow patterns that are referenced in the following discussion in order to clarify the concepts.

In the simplest test scenario it might be sufficient to have a purely sequential execution of a number of tasks (workflow pattern 1: sequence). Listing 4.8 illustrates such a sequential scenario where three tasks are executed after another (first $\mathrm{t1}$ then $\mathrm{t2}$ and finally $\mathrm{t3}$).

Listing 4.8: Three tasks are sequentially executed

```
1 sync 't1', { /* ... */ }
2 sync 't2', { /* ... */ }
3 sync 't3', { /* ... */ }
```

However, distributed systems are composed of processes that are executed in parallel. Therefore, test scenarios for distributed systems have to enable the test author to express concurrency. Listing 4.9 uses a combination of synchronous and asynchronous task execution together with explicit synchronization. It could be translated as: $\mathrm{t1}$ and $\mathrm{t2}$ should be executed in parallel (workflow pattern 2: parallel split; occurs once for every asynchronous task submission) while $\mathrm{t3}$ should only be executed when $\mathrm{t1}$ and $\mathrm{t2}$ are finished (workflow pattern 3: synchronization; occurs once for every await statement).

Listing 4.9: A partial order is defined using explicit synchronization

```
1 async 't1', { /* ... */ }
2 async 't2', { /* ... */ }
3 await 't1'; await 't2'
4 sync  't3', { /* ... */ }
```

In contrast to Listing 4.8 that only uses synchronous tasks, Listing 4.10 only uses asynchronous tasks. In such a case all tasks are potentially executed in parallel as no execution order is defined. It is only guaranteed that the tasks are submitted in the order in which they are stated (e.g. $\mathrm{t1}$ will be submitted before $\mathrm{t2}$) because task submission is atomic.

Listing 4.10: No synchronization; any task could complete first

```
1 async 't1', { /* ... */ }
2 async 't2', { /* ... */ }
3 async 't3', { /* ... */ }
```

Task Nesting as Function Nesting

Section 4.2.2 introduces the concept of nested tasks as an instrument to structure and modularize test scenarios. Figure 4.5 illustrates the concept using a task that contained two nested tasks within its workload. The nesting must be easily expressible within test scenarios. One way to achieve this is to translate nesting of functions into nesting of tasks. Because Groovy is a functional programming language it provides first-class support for functions. This allows to reference functions through variables and to pass functions as arguments to other functions (i.e. function nesting). Expressing task nesting using the Java language would be much more verbose because anonymous inner classes would have to be used to implement single-abstract-method interfaces (e.g. $\mathrm{Callable}$ or $\mathrm{Runnable}$) on many occasions. This would substantially decrease the readability of test scenarios.

Listing 4.11 shows how nesting of anonymous functions in the Groovy language can be applied to express nesting of synchronous tasks (i.e. $\mathrm{t3}$ is nested into $\mathrm{t2}$ and $\mathrm{t2}$ is nested into $\mathrm{t1}$). The submission order is the same as in Listing 4.8, but the finishing order is reversed compared to Listing 4.8 which means that $\mathrm{t1}$ can only finish after $\mathrm{t2}$ is finished which in turn can only finish after $\mathrm{t3}$ is finished. Nested synchronous task submissions can always be rewritten as purely sequential scenarios without nesting. Therefore, the code example corresponds to the sequence workflow pattern.

Listing 4.11: Three nested synchronous tasks (termination order: t3, t2, t1)

```
1 sync 't1', {                          /* t1 finishes only when t2 and t3 are done */
2     sync 't2', {                      /* t2 finishes only when t3 is done */
3         sync 't3', { /* ... */ }}}    /* t3 starts after t2 is submitted */
```

Nesting of tasks can help to structure and modularize test scenarios hierarchically. Analogous to nested synchronous tasks asynchronous tasks, can also be nested. Listing 4.12 illustrates three nested tasks that are executed asynchronously. As in the example above tasks are reliably submitted in the order of declaration ($\mathrm{t1}$, $\mathrm{t2}$, $\mathrm{t3}$) but the interleaving of start and finishing state transitions of all tasks is arbitrary because every task submission forks of one parallel execution thread.

Listing 4.12: Three nested asynchronous tasks (arbitrary termination)

```
1 async 't1', {                          /* t1 might finish even before t2 or t3 */
2     async 't2', {                      /* t2 is submitted after t1 but before t3 */
3         async 't3', { /* ... */ }}} /* t3 is submitted last; might finish first */
```

It is likely that deeper nesting levels of tasks might result in complex test scenarios with an execution semantic that is difficult to follow or understand. Especially in the case of asynchronous tasks, test scenarios should be designed in a way that deep nesting levels are only used where they cannot be avoided.

Defining Node Groups

Section 4.2.2 emphasizes the need for node groups and the need to apply tasks to groups of nodes which led to the concept of task groups. Before task groups are discussed in the following section it is first described how node groups can be defined. The first line in Listing 4.13 defines a group named all which references five nodes that might be used to address all nodes (i.e. machines) in a distributed system. Noteworthy, groups only contain references to actual nodes. The method $\mathrm{mockNodes(int)}$ creates mocked nodes for testing purposes and is used only for demonstration in this case. In a real test scenario the list of nodes would be queried from an IaaS provider. Every node can be included in multiple groups.

Listing 4.13: Defining groups of nodes: all, server, client

```
1 group 'all', mockNodes(5)                        /* Create some mock nodes */
2 group 'server', ids(group('all')).any()          /* Select an arbitrary node */
3 group 'client', group('all') - group('server')   /* A group for clients */
```

The second line of the listing selects an arbitrary node from all nodes of the system and adds it to the newly defined group server. Afterwards, the group client is created containing all nodes excluding those that are part of the server.

Using Task Groups

The previous sections introduced statements for task submission using sync and async, explicit synchronization using await, task libraries and node groups. These concepts can be combined to define task groups. Listing 4.14 shows how a task group can be submitted asynchronously. It applies one task instance to every member of the node group all respectively.

Listing 4.14: A task group with at least three nodes

```
1 async 'task group', common.mockTaskGroup(group('all'))  /* Create a task group */
2 await 'task group', 3      /* Three of the five child tasks must be done */
3 await 'task group'         /* All child tasks must be done */
```

The second statement is an explicit synchronization that blocks execution until three of the tasks in the submitted task group are completed. It is assumed in this case that the task group contains three or more nodes. The third statement finally blocks until the task group is finished meaning

that every task within the task group is finished. In this case, the statement does only make sense if the task group contains more than three tasks.

Advanced Flow Control

The described examples mainly correspond to the three fundamental workflow patterns: sequence, parallel split, and synchronization. More advanced flow control that corresponds to other workflow patterns is possible using language elements of the Groovy host language. It offers loops with a variable number of iterations (while) and a fixed number of iterations (for) and conditional execution (if ... then ... else or switch ... case). These elements can be used to implement scenario execution patterns that correspond to patterns in [2]: exclusive choice (4) or multiple choice (6) using if ... then ... else, synchronizing merge (7) using multiple await statements, or arbitrary loops (10) using for or while. These advanced flow control elements are excluded from the following discussion about semantic analysis because they make the analysis more complex. It might even be impossible to reason about test scenarios that can contain loops with a dynamic iteration count, because of the turing-completeness of WHILE programs.

4.3.3. Semantic Analysis

The previous sections describe how distributed test scenarios can be modelled and expressed using an internal Groovy DSL. Because the DSL is embedded into the Groovy language, every test scenario is also a valid Groovy program. This allows using the Groovy compiler to detect Syntactical errors at compile time. Examples of incorrect syntax are missing or not matching brackets or a missing semicolon. Additionally, Groovy provides optional support for static type checking based on annotations. Constraints that are introduced by types can therefore also be checked statically. Issues that are typically detected by static typing include invocation of non-existing methods or passing parameters that have an incompatible type. This allows discarding incorrect test scenarios prior to their execution helping to avoid test defects. In the context of distributed testing it is of great importance to detect test defects in scenarios because they might take several minutes to execute and allocate testing resources.

Static syntax and type checking is provided by the Groovy host language but it is limited to check constraints that are applicable to any program. Constraints that are domain specific must be created individually for every domain. For the domain of distributed test scenarios some additional constraints are useful. For instances, it makes sense to discard test scenarios that contain a dead lock.

Listing 4.15: A typing error

```
1 async 'my asynchronous task', common.mockTask()
2 await 'my asynchronuos task'  /* contains a typing error */
```

Another useful constraint would be, that no await statement should exist awaiting a task that will never be started. This might simply happen because the task label is not spelled correctly. Listing 4.15 shows a test scenario that is syntactically correct but it is semantically incorrect because

the test author made a typing error. Static analysis can find such issues. In order to further reason about the execution semantics of test scenarios the program code can be translated into a dependency graph.

Deriving a Dependency Graph

To allow detailed static analysis of the execution of tasks within a test scenario, a directed dependency graph $G_{dep} = (V', E_{dep})$ can be used that expresses in which order the statements – and therefore tasks – are executed. The set of vertices V' of the dependency graph is the set of all statements in a test scenario. Thus, a task is represented as a sequence of statements. Every dependency is represented by a directed edge. Edges are described by the transitive relation $E_{dep} \subseteq V' \times V'$ where $(a, b) \in E_{dep}$ models that a must be finished before b can start. Based on the dependency graph, reasoning about the run-time behaviour of the test scenario becomes possible. Dependency graphs are also used by compilers to detect code sections that can be potentially executed in parallel because they are independent.

Groovy provides an AST builder[12] that allows creating an abstract syntax tree (AST) from an arbitrary Groovy script. The abstract syntax tree G_{syn} represents the script of the test scenario as a tree data structure based on its syntax. Every vertex V in the syntax tree is a statement of the test scenario. Therefore, the set of vertices in the dependency graph (V') is only a copy of the vertices in the syntax tree (V). The edges in the syntax tree E_{syn} represent parent-child relations of statements in the test scenario.

This tree representation can be translated into a dependency graph that represents the execution semantics. The dependency graph visualizes tasks and the parallel execution threads that are caused by submission of the tasks.

Algorithm 1 provides a pseudo-code representation of a recursive procedure that realises a translation from an abstract syntax tree of a test scenario $G_{syn} = (V, E_{syn})$ into a dependency graph of a test scenario $G_{dep} = (V', E_{dep})$. Appendix A.6 contains a detailed listing of a Java implementation. The algorithm iterates recursively through the syntax tree using a depth first strategy (see line 12). Listing 4.16 shows a small test scenario that can be used to demonstrate the translation process.

Listing 4.16: Example scenario to demonstrate the algorithm

```
1 sync  't1', { print '1'; print '2' }   /* a sync task with two statements */
2 async 't2', { print '3'; print '4' }   /* an async task with two statements */
3 print '5'; await 't2'
```

The test scenario contains a task that is submitted synchronously and another task that is submitted asynchronously. It uses eight statements in total that represent vertices in the syntax tree but are also the vertices of the derived dependency graph. Figure 4.8 shows the (simplified) abstract syntax tree for the scenario. Solid edges belong to the abstract syntax tree and dashed edges represent the dependency graph. The dependency graph shows how the synchronous task $\mathrm{t1}$ is translated into a sequential execution: $\mathrm{s1} \rightarrow \mathrm{s2} \rightarrow \mathrm{s3}$. Statement $\mathrm{s4}$ submits the asynchronous

[12] fully qualified name: `org.codehaus.groovy.ast.builder.AstBuilder`

task $\mathrm{t2}$ which results in two parallel execution threads: $\mathrm{s4} \rightarrow \mathrm{s5} \rightarrow \mathrm{s6}$ and $\mathrm{s4} \rightarrow \mathrm{s7}$. Finally, the await statement $\mathrm{s8}$ joins both execution threads.

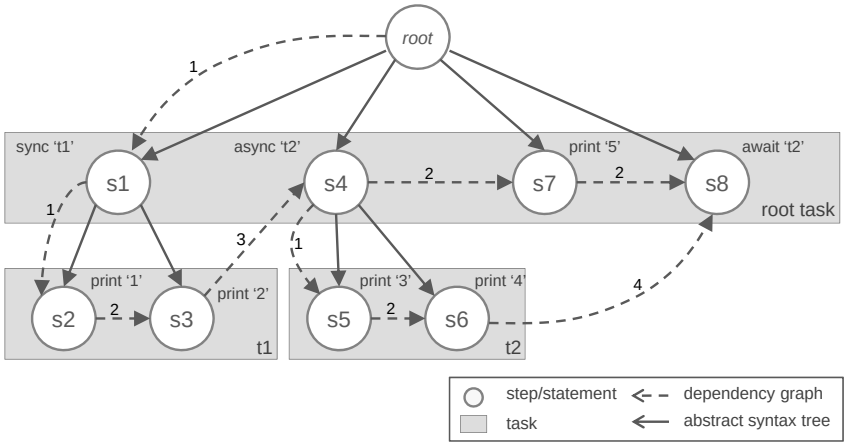


Figure 4.8.: Example of how algorithm 1 translates the syntax tree into a dependency graph. The graph figure is based on Listing 4.16. The numbers within the vertices ($\mathrm{s1}$ to $\mathrm{s8}$) represent the order in which the algorithm visits the statements and numbers on the dashed edges refer to the situations when the algorithm adds the edge to the graph (i.e. case 1, 2, 3 or 4). Vertices: $V' = V$, edges: E_{dep} (dashed), E_{syn} (solid)

At first, the set of dependencies is empty ($E_{dep} = \emptyset$). Then the algorithm iterates over every statement in the order indicated by the numbers within the vertices in the figure ($\mathrm{s1}$ to $\mathrm{s8}$). The algorithm distinguishes four cases in which a dependency is added to the set of edges E_{dep}. Each of the four cases is also represented in the example scenario and is indicated by a number next to the dashed edges in Figure 4.8. Case 1, line 6: if the current statement is the first child statement in the AST, then a dependency is added from its parent statement (e.g. $\mathrm{s1} \rightarrow \mathrm{s2}$). Case 2, line 9: if the current statement is not the first child statement in the AST and is *not* a sync statement (those are handled by case 3), then a dependency is added from that statement (e.g. $\mathrm{s2} \rightarrow \mathrm{s3}$). Case 3, line 15: if the current statement is not the first child statement in the AST and *is* a sync statement, then a dependency is added from the last child statement of the previous statement (e.g. $\mathrm{s3} \rightarrow \mathrm{s4}$). If the previous sync statement has no child statements, then a dependency is added from the statement itself (not shown). This is expressed by the $\mathrm{lastChildStmOrSelf}()$ function in the algorithm. Case 4, line 23: the dependencies that are introduced by await synchronization are added to the graph within a loop with a worst case complexity of $\Theta(n^2)$ steps. The algorithm might be improved in order to perform with a lower complexity.

In order to create the dependency graph some simplifying assumptions are made. First, statements are filtered by a white list before they are added to the graph. Only statements that are method calls to the methods sync, async, await and print are allowed currently. Second, no loop expressions (e.g. for or while) are allowed although loops that have a constant number of iterations could be included in future versions of the algorithm. Third, conditional branching using if or case statements is not considered although they might be useful to express more advanced test scenarios. Fourth, task groups are treated as a single vertex although they are internally composed of multiple tasks. The reason for this is that task groups cannot be analysed statically because their size might be dynamically determined and thus only known at run time.

Algorithm 1 Conversion of an abstract syntax tree into a dependency graph

Require: $G_{syn} = (V, E_{syn})$ represents the abstract syntax tree of test scenario.
Ensure: $G_{dep} = (V', E_{dep})$ represents the dependency graph of test scenario.

```
 1: function CONSTRUCTDEPENDENCYGRAPH( parentStm, G_dep, depth)
 2:     prevStm ← null
 3:     for each expression e in parentStm.getClosure() do                    ▷ O(n)
 4:         V' ← stm ← newVertex(e, prevStm, depth)
 5:         if !stm.hasPrevStm() then
 6:             E_dep ← (parentStm, stm)                                      ▷ case 1
 7:         end if
 8:         if stm.hasPrevStm() && !stm.getPrevStm().isSync() then
 9:             E_dep ← (stm.getPrevStm(), stm)                               ▷ case 2
10:         end if
11:         if stm.containsClosure() then
12:             constructDependencyGraph(stm, G_dep, depth + 1)               ▷ recursion
13:         end if
14:         if stm.hasPrevStm() && stm.getPrevStm().isSync() then
15:             E_dep ← (stm.getPrevStm().lastChildStmOrSelf(), stm)          ▷ case 3
16:         end if
17:         prevStm ← stm
18:     end for
19:
20:     for each awaitStm in V' do
21:         for each asyncStm in V' do                                        ▷ Θ(n^2)
22:             if awaitStm.getLabel() == asyncStm.getLabel() then
23:                 E_dep ← (asyncStm.lastChildStmOrSelf(), awaitStm)         ▷ case 4
24:             end if
25:         end for
26:     end for
27: end function
28:
29: constructDependencyGraph(startStm(G_syn), G_dep, 0)                       ▷ start conversion
```

Test Scenario Visualization

The algorithm described above can be used to translate a test scenario into a dependency graph representation. The dependency graph can be stored using a sparse directed graph data structure[13] from the Java Universal Network/Graph Framework (JUNG) framework. The framework also provides basic graph layout and visualization capabilities. The implementation of Meyer's self-organizing map layout algorithm ISOM[14] is used for visualization of the dependency graph in all following figures. Figure 4.9 shows the result of the graph generation that was applied to the example scenario from the previous section (Listing 4.16). The graph is equivalent to the dependency graph in Figure 4.8 because it is based on the same listing. The START and END vertices are added to the graph in order to make it more readable. Vertices that represent task submission with sync or async are depicted as squares; all other statements are depicted as elliptic shape. The graph can be generated and displayed while the test author works on the test scenario permitting direct feedback.

Reasoning about Execution Semantics

The dependency graph of a test scenario not only allows to visualize the execution paths of a test scenario but also serves as a foundation for static reasoning about the run-time behaviour of scenarios. The following paragraphs discuss some examples of static analysis that are based on

[13] fully qualified name: `edu.uci.ics.jung.graph.DirectedSparseGraph`
[14] fully qualified name: `edu.uci.ics.jung.algorithms.layout.ISOMLayout`

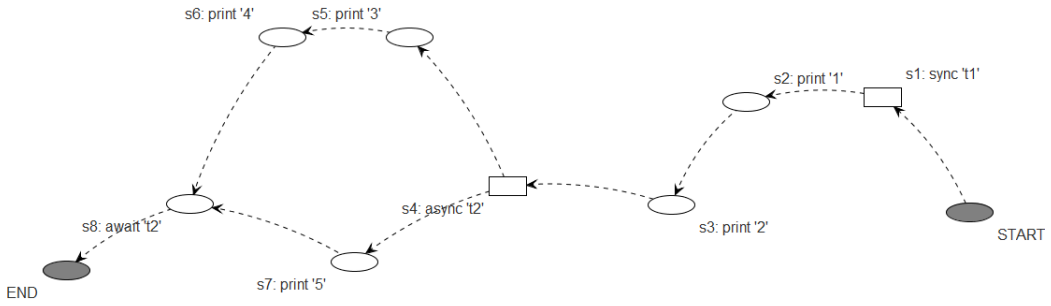


Figure 4.9.: Visualization of the dependency graph that was derived from the test scenario in Listing 4.16. The graph shows the execution semantics of synchronous task submission in statement $\mathrm{s1}$ and asynchronous task submission in statement $\mathrm{s4}$.

the dependency graph representation of test scenarios.

Dead Lock Detection A useful constraint is surely to discard scenarios that contain a dead lock. While dead locks are hard to spot in the textual representation of a test scenario, they be can easily recognized in the dependency graph representation.

Listing 4.17: A scenario that would not terminate

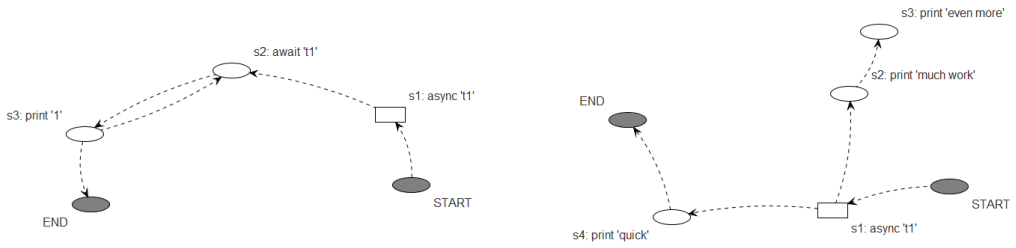
```
1 async 't1', { await 't1'; print '1' }  // 't1' depends on itself
```

Listing 4.17 contains a dead lock situation that is caused by the await statement. It is located within the task $\mathrm{t1}$ meaning that the execution of the task depends on its own termination. Figure 4.10(a) shows how this scenario translates into a dependency graph. The print statement $\mathrm{s3}$ depends on the await statement $\mathrm{s2}$ because of the ordering in which the statements are defined (case 2 in the algorithm 1). At the same time, the await statement $\mathrm{s2}$ depends on the last statement in the task definition of $\mathrm{t1}$ (case 4 in algorithm 1), which is the print statement $\mathrm{s3}$. This results in a cyclic dependency which is a sufficient condition for a dead lock. Fully automated detection of scenarios that contain a dead lock is valuable because it helps to avoid test defects that would otherwise require extensive debugging at run time and unnecessary resource allocation.

Unsynchronized Execution Detection Another information that can be extracted from the dependency graph are asynchronous tasks that are not synchronized (using await) at any point. Tasks that are never synchronized do not necessarily cause problems in the scenario execution but bear the potential for unexpected behaviour.

Listing 4.18: The task 't1' is never synchronized

```
1 async 't1', { print 'much work'; print 'even more' }
2 print 'quick'
```



(a) A deadlock scenario (see Listing 4.17) (b) An unsynchronized task (see Listing 4.18)

Figure 4.10.: Two small test scenarios represented as dependency graph. Figure 4.10(a) shows how a cyclic dependency in the graph indicates a dead lock situation. Figure 4.10(b) shows an asynchronous task ($\mathrm{t1}$) that has no corresponding await statement which might indicate a test defect.

Listing 4.18 and the corresponding graph in Figure 4.10(b) show a scenario that submits an asynchronous task $\mathrm{t1}$ that has a long running workload (i.e. statements $\mathrm{s2}$ and $\mathrm{s3}$). After task $\mathrm{t1}$ is submitted, the scenario continues with statement $\mathrm{s4}$ and reaches the END vertex, which causes the scenario to be terminated, although task $\mathrm{t1}$ is still busy executing its workload. Therefore, unsynchronized tasks can be considered an anti-pattern that can be extracted from the dependency graph. A vertex with an outgoing edge degree of less than one, is a sufficient condition to detect execution threads that are not synchronized/awaited, if the END vertex is excluded. Statement $\mathrm{s3}$ in Figure 4.10(b) is an example of such a vertex.

Unnecessary Synchronization Detection The semantic analysis can detect await statements that are unnecessary. For instance, it does not make sense to wait for a task to finish twice in a scenario if a dependency between these await statements exists. Listing 4.19 exemplifies this case. Task $\mathrm{t2}$ and $\mathrm{t3}$ both wait until $\mathrm{t1}$ is finished. Meanwhile it is waited for task $\mathrm{t2}$ and afterwards for $\mathrm{t1}$. To wait for $\mathrm{t1}$ does not do any harm but is unnecessary because it is already guaranteed that $\mathrm{t1}$ has finished, because termination of $\mathrm{t2}$ is awaited in the previous statement and $\mathrm{t2}$ awaits the termination of $\mathrm{t1}$ in its workload. If the await statement in line 4 would be missing then all three await statements that await the completion of $\mathrm{t1}$ would be reasonable and no unnecessary synchronization would be detectable.

Listing 4.19: Example of an unnecessary await statement

```
1 async 't1', { print 'some workload' }
2 async 't2', { await 't1' }  // both 't2' and 't3' wait for 't1' in parallel
3 async 't3', { await 't1' }  // thus both awaits are reasonable
4 await 't2'  // This guarantees that 't1' is finished
5 await 't1'  // Not necessary because of prev. await
```

Unnecessary await statements can be detected in the dependency graph by finding all await statements that wait for the same task and then determine if a path exists between any two of the statements. If a path exists, then the last statement in the path is an unnecessary await statement because the dependency graph guarantees that such an await statement has already

been passed. This analysis is comparable to the detection of assignments in general purpose languages that have no effect (e.g. $\mathrm{a}{=}\mathrm{a}$). It relies on the restriction that a task can be only submitted once during the course of a test scenario.

Tool Support for Test Authors

IDE and other tool support is one main reason to develop a language that is specific to a domain [49, p. 40ff.]. The development of internal Groovy domain-specific languages is well supported by Eclipse and the Groovy-Eclipse plugin. Key features include syntax highlighting, auto-complete, refactoring, type inference, and formatting. The most basic support for the test author is to report errors that result from invalid syntax (e.g. missing brackets) and from invalid semantics (e.g. using variables before their declaration, type-based constraints). Both is supported by recent Groovy versions although Groovy is a dynamic language. Static type checking can be activated on method or class level using annotations. If the type of a variable is known or can be inferred statically, then an IDE can offer code completion as shown in Figure 4.11.

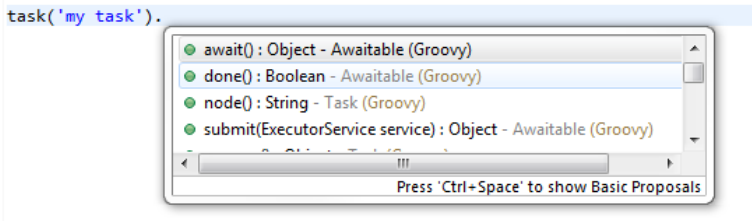
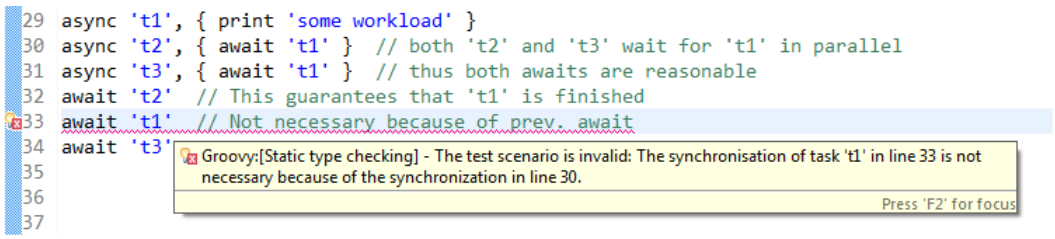


Figure 4.11.: Static typing allows specific code completion while typing. In this case, code completion is provided for the type Task (see interface definition in Figure 4.7). A reasonable suggestion is to call the $\mathrm{await}()$ method that would block execution until task $\mathrm{my\ task}$ is finished.

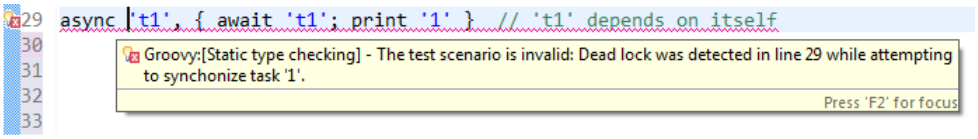
The section above describes how a test scenario can be translated into a dependency graph that represents the execution semantics of the test scenario. This graph permits further analysis of the run-time behaviour of the test scenario and is the foundation to detect potential test defects. The static type checking mechanism of the Groovy compiler can be extended with such domain-specific semantic validations that are not described with the grammar of the language. Groovy offers an event based API that allows hooking custom validations into the compilation process. The process can be extended at different events, for instance, before or after the compiler visits a method.

Using a type checking extension is somewhat misleading because the checked constraints are not based on the type system, but the mechanism is the easiest way to display issues in the Eclipse code editor. Test defects are added as compilation errors[15] and displayed in the Eclipse code editor. Figure 4.12 shows three examples of test defects that are based on the analysis of the dependency graph of the test scenario. Figure 4.12(a) shows how expressive the domain-specific error messages are as it is gives detailed information on the exact line numbers of the redundant synchronization. Another useful feature is shown in Figure 4.12(c) where an await statement is detected that has no corresponding task submission in the test scenario because

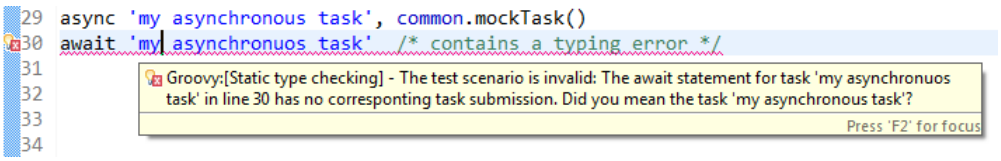
[15]fully qualified name: `org.codehaus.groovy.transform.stc.addStaticTypeError`



(a) The Groovy static type checking mechanism is extended to detect unnecessary synchronization (`await 't1'`). The previous `await` statement already guaranteed that `t1` is finished. Details are described in Section 4.3.3



(b) A dead lock has been detected while authoring a test scenario. The termination of task `t1` depends on itself. Details are described in Section 4.3.3



(c) A test scenario that contains a test defect because the test author made a typing error (see also Listing 4.15). A correction can be suggested based on the static analysis.

Figure 4.12.: A Groovy plugin is available for the Eclipse IDE platform that allows annotating the source code with potential issues (curved underline). Problems are indicated at compile time while the test author is typing.

of a typing error. The double Metaphone algorithm[16] that was published by Lawrence Philips is used to suggest other labels of tasks in the test scenario that sound similar to the wrong label. The example shows that the static analysis can suggest the correct task label: "`my asynchronous task`". The double Metaphone algorithm is an improved version of the Metaphone algorithm that can be used for indexing words based on their pronunciation. It works not only for English words but also for other languages.

4.4. Implementation

Acceptance testing can be organized into a three layered high-level reference architecture [26]: acceptance criteria layer, test implementation layer and application driver layer (Figure 4.13). This architecture can be used to structure the discussion of the implementation of the concepts that are described in the previous sections.

During the course of this work the three layer architecture has been extended by an infrastructure driver layer that automates the management of the target infrastructure. This layer is very similar to the application driver layer in the sense that it abstracts details of different infrastructure providers in order to enable a provider change. The application and infrastructure driver layer are both implementing the task-oriented model that is described in Section 4.2 for operation.

[16] fully qualified name: `org.apache.commons.codec.language.DoubleMetaphone`

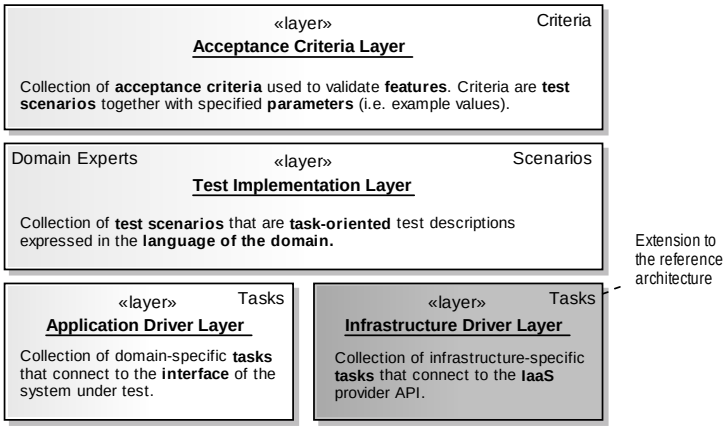


Figure 4.13.: Three layers of acceptance testing as suggested by [26]: acceptance criteria layer, test implementation layer and application driver layer. The architecture is extended by the infrastructure driver layer.

4.4.1. Acceptance Criteria Layer

The acceptance criteria layer connects features of the system under test with test scenarios using acceptance criteria. An acceptance criterion is a reference to a test scenario together with concrete values for the parameters that the test scenario accepts. An example of this is given in Section 4.1.2. In order to automate the execution of acceptance criteria the test runner of the JUnit framework version 4 is used, because it is one of the most mature test execution frameworks that is available. Using JUnit, every acceptance criterion can be encapsulated into a method that instantiates a test scenario together with concrete values of the parameters. The test class can be used to combine all acceptance criteria that relate to a feature. This makes it possible to evaluate a feature by executing all test methods in a test class using the test runner of JUnit. If all tests succeed then the feature is assumed to be properly implemented. Test classes can be written using Groovy or Java. Listing 4.20 is an example of a Groovy "unit test" used to instantiate and execute Test Scenario $\mathrm{TS}2.1$ with parameters specified in Acceptance Criterion $\mathrm{AC}2.1.1$. The test scenario is discussed in the MongoDB case study in Section 5.3.2.

Listing 4.20: A method in a unit test class that is used to instantiate a test scenario

```
1  @Test public void runTestScenario21() {
2    // open a session to a IaaS provider (e.g AWS)
3    final TestSession testSession = TestSession.getInstance("test", "d:/thesis",
4      CredentialFactory.create().getContext("tmp-aws.context.properties"));
5    // create an instance of a test scenario from a groovy script file
6    final Scenario scenario = AbstractScenario.loadScenario(
7      new File("domain/mongodb/TestScenario21.groovy").getAbsolutePath(),
8      Executors.newCachedThreadPool(), testSession,
9      ["replicaSetSize": "3", "failingCount": "1", "toleranceExpected": "true"]);
10   // execute the test scenario
11   scenario.executeBlocking();
12   // let the "unit test" fail if the scenario fails and vice versa
13   assertTrue(scenario.getSuccess());
14 }
```

The JUnit test runner is integrated into many IDEs including Eclipse. The test author can therefore use the same tool for test authoring and test execution. Figure 4.14 shows a) the test class in the Eclipse package explorer b) three acceptance criteria that are contained in the test class c) the current status of the running test, and d) the logging output that is generated during the test execution. The feature that is evaluated in this case is the high write availability of the MongoDB database system during failure of nodes in a replica set, which is discussed in detail in the evaluation chapter in Section 5.3.3.

The execution of acceptance criteria can also be triggered without a tool environment using the standalone binary version of the developed framework. The binary version is a JAR file that accepts command line parameters which specify credentials for the virtual infrastructure provider (e.g. AWS), the location of the test scenario on the hard drive, and an output directory that is used to store test results. The command-line interface of the framework is only prototypically implemented because a sophisticated implementation is beyond the stated goals of this thesis.

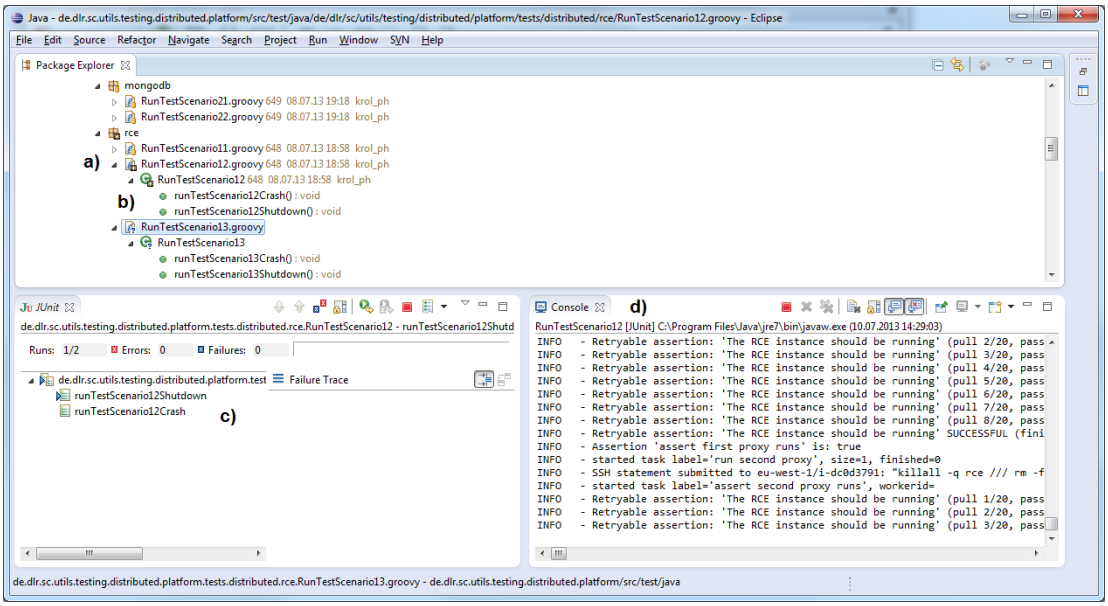


Figure 4.14.: Overview of the Eclipse environment that can be used for test authoring. The Eclipse perspective shows a) a Groovy test class that contains acceptance criteria as tests, b) three test methods that represent one acceptance criteria each, c) the current JUnit test runner status, and d) the console output of the test execution. The source code view has been hidden in order to provide a better overview.

4.4.2. Test Implementation Layer

The test implementation layer is concerned with the development and maintenance of test scenarios that are specific to the system under test. Test scenarios are expressed in the domain language of the system under test and are written by domain experts but not by developers. Every test scenario is represented as a Groovy script that defines tasks and can be executed and validated fully automatic. The language that is used to describe test scenarios is an internal domain-specific language that is hosted in the Groovy programming language. It uses a common vocabulary to express circumstances that are needed in every distributed task-oriented test sce-

nario. Examples of these general language elements are task submission with async or sync and synchronization of tasks with await. This vocabulary is extended with an array of tasks that allow operating the system under test and the target infrastructure. The application and infrastructure driver layer provide these tasks as task libraries. Most of the listings in the previous section and also the extended listings from different case studies in Appendix A are located in the test implementation layer. Changes in that layer do not require compilation, because the test scenarios are loaded dynamically from the source files.

4.4.3. Application Driver Layer

The application driver layer abstracts the connection to the interface of the system under test. It decouples the concrete interaction with the system under test from the high-level domain language that is used in the test implementation layer to describe test scenarios. One advantage of decoupling test descriptions from the interface of the system under test is that changes in the interface can be handled without changes in the test scenarios. However, if the domain logic changes then changes in the test implementation layer cannot be avoided.

The application driver layer is realized as collections of tasks that are grouped into task libraries (see Section 4.3.2). The test author might choose tasks from the libraries to compose test scenarios. Ideally, a few parametrized tasks can be combined into a large variety of test scenarios. Most test scenarios that were developed in the course of the evaluation make use of two task libraries (i.e. common and domain). The common library contains tasks that are not specific to the system under test and can therefore be widely reused. Examples of such tasks are file up and download, testing files for conditions (e.g. containing a given string) or probing metrics about the resource consumption of a node (e.g. CPU or memory consumption) . The domain library contains tasks that are specific to the domain of the system under test. Examples of such tasks are typically concerned with the installation, configuration, and execution of the system under test in a given virtual infrastructure.

An initial effort is required for every new system that should be tested in order to create a pool of tasks. This can be accomplished in collaboration with domain experts and system developers. The application driver layer should be continuously adopted and improved in order to evolve towards a language that expresses test scenarios, use cases, and eventually requirements to the system under test as good as possible. It might also be used as a tool to bridge the communication gap between domain experts and system developers. Task libraries can be implemented using Java or Groovy.

Agent-based Execution

A task as described in the task-oriented model (see Section 4.2) contains a workload that describes what must be done to complete the task. Typically the workload performs requests to remote systems which might take a considerable amount of time to complete. One example is a task that connects to an IaaS provider to request new computing nodes (VMs). Often it takes at least 45 seconds until such a request is fulfilled. Once the infrastructure is provisioned, a set

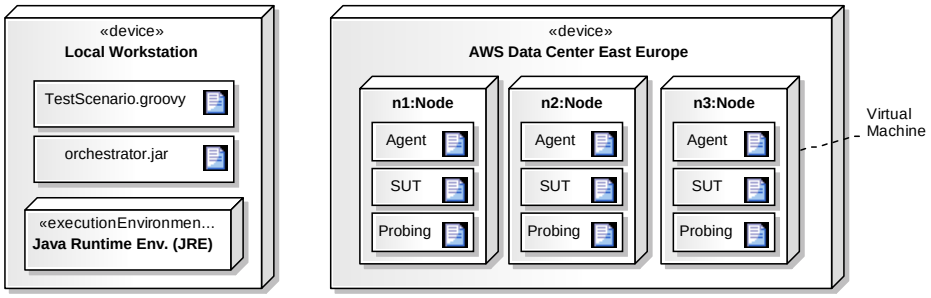


Figure 4.15.: UML deployment diagram as physical view on the deployment during test execution. Any local workstation can orchestrate the test execution, given a scenario script and the binary of the orchestrator. The JVM must be installed on the orchestrating workstation. The virtual test infrastructure can be hosted in any datacentre that is accessible as a service. In this case, an AWS datacentre in eastern Europe.

of blank nodes is available. From that point on most tasks contain a workload that executes behaviour on a particular node or a group of nodes (i.e. node group). For instance a task might initialize a node by creating a dedicated directory and install some basic software. To make this possible, every node must be capable of receiving arbitrary instructions.

A conceptual approach that solves this problem is to deploy agents on every node in the test infrastructure (see Figure 4.15). After the agents are installed, they listen for commands and execute those on the node. Once the execution is finished, the agent returns an agent response that contains information about the execution of the command on the node. A design goal is to decouple any agent implementation from the rest of the implementation. Dependencies towards any agent functionality should be expressed using interfaces instead of concrete implementations in order to allow exchanging different implementations of the agent concept. Another benefit is that the implementation becomes easier to test, because a mock implementation of the agent concept can be used to execute tests without any actual remote communication.

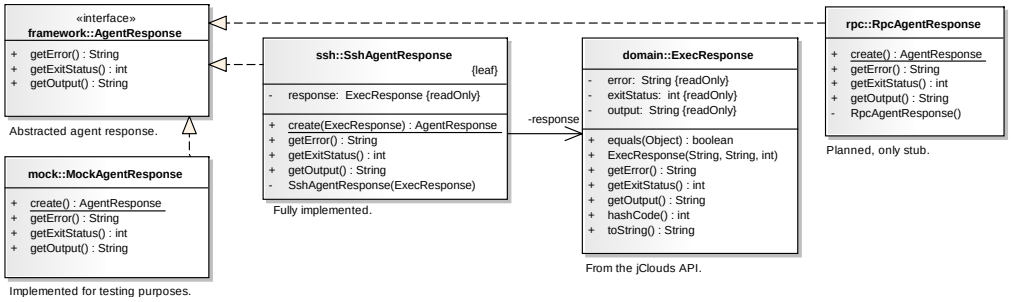


Figure 4.16.: UML class diagram that shows the $\mathrm{AgentResponse}$ interface that is implemented by the SSH and the mock agent. The $\mathrm{SshAgentResponse}$ wraps the $\mathrm{ExecResponse}$ that is provided by the jClouds API.

The $\mathrm{AgentResponse}$ interface that is shown in Figure 4.16 describes the contract that any agent implementation must conform to. It also shows three implementations: an SSH based agent, a mock agent for testing and a remote procedure call (RPC) implementation that has not been implemented in the scope of this thesis. The following paragraph discusses how the generic agent concept can be implemented using SSH for remote communication.

SSH Agent One implementation of an agent-based architecture is SSH itself. An SSH connection is initialized by a client and established towards a demon (or server) that represents the agent. In the case of SSH the commands that are submitted are shell commands. Using SSH brings some advantages. First, mature implementations exist for many UNIX distributions and also for Windows. Second, SSH gives access to the shell of an operating system which ensures flexibility. Third, the communication is encrypted which enables a secure communication with nodes of an infrastructure. This is especially interesting if infrastructures are provisioned within a public cloud.

Listing 4.21: A method that constructs a task group (application driver layer)

```
1 public TaskGroup<AgentResponse> pingWikipedia(final Set<NodeMetadata> nodes) {
2   return sshAgent.taskGroup(new Function<NodeMetadata, Statement>() {
3       public Statement apply(NodeMetadata node) {
4           return Statements.exec("{ping} www.wikipedia.com");
5         }
6     }, nodes);
7 }
```

Listing 4.21 exemplifies how a task group ($\mathrm{TaskGroup}$) is defined within a task library. The task group executes a simple shell command via SSH that causes every node in the group to issue a "ping" to the Wikipedia website. The SSH agent receives a function that translates node metadata to shell statements in its apply method. Every task within the task group has the return type $\mathrm{AgentResponse}$. In this case, the agent response would contain the familiar output that the ping tool produces. The types $\mathrm{NodeMetadata}$ and $\mathrm{Statement}$ are provided by jClouds. The method $\mathrm{pingWikipedia}$ only constructs the task group but does not submit it. This is done within test scenarios in the test implementation layer as exemplified by an asynchronous task submission in Listing 4.22.

Listing 4.22: Submitting a task group from within a scenario (test implementation layer)

```
1 async 'ping wikipedia', common.pingWikipedia(group('all'))
```

Figure 4.17 uses an UML sequence diagram to summarize the events that are triggered when a task group is submitted. Solid arrowheads represent synchronous execution, otherwise execution is asynchronous. Thread 0 executes the task group asynchronously so that the scenario continues after submission. The code in Listing 4.22 would trigger this behaviour. The three remaining threads 1, 2 and 3 execute the individual tasks of the task group. In general $n + 1$ threads are required to execute a task group of size n.

The task group coordinates its child tasks and hides the access to the necessary threads. The workload of the individual child tasks contain the establishment of the SSH connection to a remote node followed by the execution of a list of shell statements. The remote communication and statement execution is synchronous so that the respective thread blocks until the shell finished executing all statements. Therefore, the actual work is done on the node and the thread is mostly idle until the work is done. This means that the threads only act as proxies to the processes that are executed on the nodes. The SSH demon crates a new process for every established

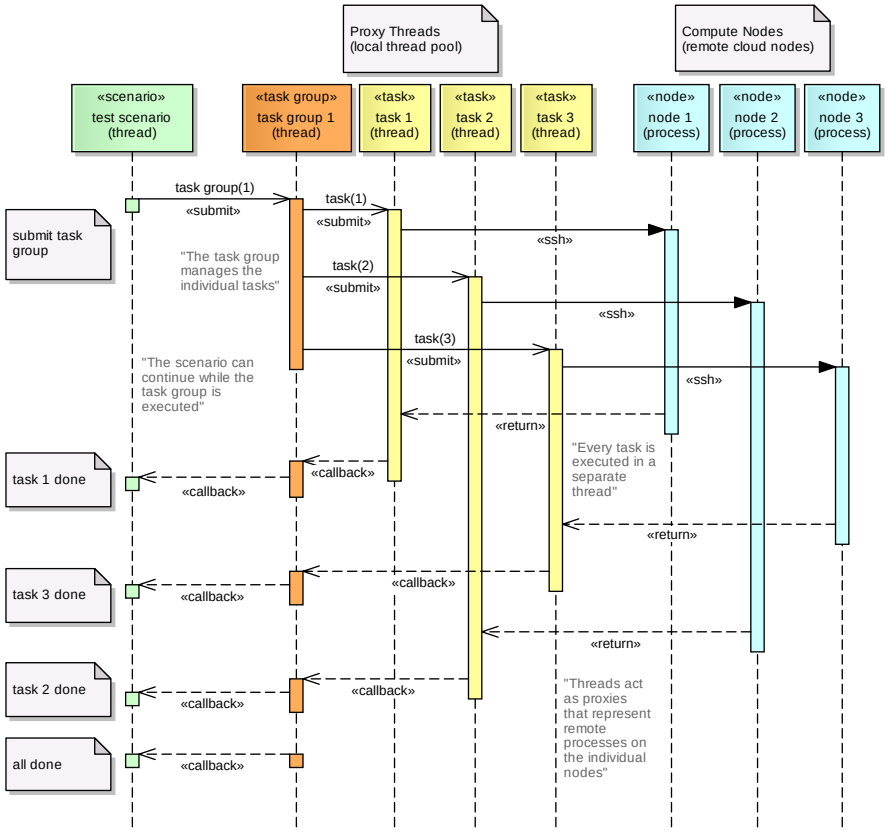


Figure 4.17.: An UML sequence diagram that shows an example of a task group that has three tasks. The diagram shows how each task and task group is executed by a local thread. The thread acts as proxy that initiates a blocking remote communication (in this case SSH) that is used to invoke the agent to execute the task on its corresponding node.

connection so that multiple parallel SSH connections to the same node do not cause any side effects by default. A disadvantage of the SSH system is that shell scripts must be designed very carefully to be platform independent. Some statements might not even be portable between different UNIX distributions. Examples are the differences in package manager interfaces that are used by Red Hat distributions: Yellowdog Updater Modified (YUM) and Debian distributions: Advanced Packaging Tool (APT). Section 5.1.3 discusses a mechanism that enables expressing portable statements in a platform independent representation that can be rendered to different platform specific commands by the agent.

Mock Agent As visible in Figure 4.17, every submitted task group requires $n + 1$ threads to be created for the execution. If task groups are large and the test scenario submits many tasks in parallel then a considerable amount of concurrently acting threads are created. For the purpose of efficient testing of the task implementation itself that does not require actual nodes to be allocated a mock agent was implemented. The mock agent only produces logging output instead of establishing actual remote connections. The SSH agent can easily be replaced by the mock agent because the remaining system is not coupled to a specific implementation. Figure 4.16 also includes the mock implementation of the agent response interface.

4.4.4. Infrastructure Driver Layer

The infrastructure driver layer offers functionality that is concerned with the infrastructure that the test is executed on. Similar to the application driver layer, this layer is also composed of a collection of tasks. The difference is that infrastructure tasks do not communicate with the system under test via agents as described above, but perform API calls to web services of infrastructure providers. Typically these API calls aim to fulfil infrastructure management tasks like creating, destroying or restarting machines. An important function of this layer is to abstract details of the virtual infrastructure provider in order to make the provider exchangeable. A project that abstracts many different APIs is the jClouds "cloud abstraction API". It provides a common Java interface that is translated to provider-specific web service calls. The advantage of extending the reference architecture with an infrastructure driver layer (see Figure 4.13) is that infrastructure management is made available to the test author within the test scenario as tasks.

Cloud Abstraction API

The developed code base uses the jClouds API as abstraction layer between provider-specific APIs and the client code. jClouds aims to provide a common interface to services that are offered by many IaaS providers. The most common services are centred around computing, storage, and networking. In the context of this work the compute services are used mostly. Such services allow to request or release nodes (i.e. VMs) and to query the current state of these nodes to get information about the properties of the nodes (e.g. public and private IP address). Listing 4.23 shows a minimal example that uses jClouds to provision nodes from an infrastructure provider without using vendor specific API calls. The $\mathrm{ComputeService}$[17] can be used to access information about the current state of all computing nodes.

Listing 4.23: API agnostic node provisioning and destruction

```
1  // Build a compute context for AWS and fetch the compute service
2  ComputeService service = ContextBuilder.newBuilder("aws-ec2").
3      credentials("aws id", "aws api key").
4      buildView(ComputeServiceContext.class).getComputeService();
5  // Create five new nodes (i.e. VMs)
6  service.createNodesInGroup("my-group", 5);
7  // There should be five nodes now (assuming there
8  // were no nodes in the context before)
9  assertTrue(service.listNodes().size() == 5);
10 // Destroy all five nodes to free resources
11 service.destroyNodesMatching(inGroup("my-group"));
12 // All nodes should be gone by now
13 assertTrue(service.listNodes().size() == 0);
```

The method call $\mathrm{listNodes}()$ in Listing 4.23 fetches a list of available nodes. The method returns a list of $\mathrm{ComputeMetadata}$[18] entries. Every compute metadata entry contains very general information about the properties of a node. The most important property of a node is its id that allows identifying the node reliably. The type $\mathrm{ComputeMetadata}$ only contains information that is

[17] fully qualified name: `org.jclouds.compute.ComputeService`
[18] fully qualified name: `org.jclouds.compute.domain.ComputeMetadata`

common in all APIs. Richer metadata information is available through the NodeMetadata[19] type but the completeness of this information might vary between IaaS providers. ComputeMetadata is a super interface of NodeMetadata.

Listing 4.24: Establishing a SSH towards a node

```
1 // Select an arbitrary node and cast to NodeMetadata
2 NodeMetadata myNode = (NodeMetadata) service.listNodes().iterator().next();
3 // Fetch a SSH client for "myNode" based on its NodeMetadata
4 SshClient client = service.getContext().utils().sshForNode().apply(myNode);
5 // Initialize SSH connection, execute shell statement and release connection
6 client.connect(); client.exec("uptime"); client.disconnect();
```

The node metadata of a node serves as starting point for further interaction with the node. A very common task after node provisioning is to connect to the node via SSH and execute shell commands. The metadata of a node contains all necessary information (i.e. host name, user, public key) to create a SSH client[20]. Listing 4.24 shows how an arbitrary node is selected from all nodes and then a SSH client is created based on its metadata. Then the SSH client is used to submit a shell command to the node. In this case the UNIX tool uptime is called returning the time that has passed since the node was started. The execution of a SSH command returns the type ExecResponse[21] that allows to access the standard output of the execution as well as the exit status code.

[19] fully qualified name: `org.jclouds.compute.domain.NodeMetadata`
[20] fully qualified name: `org.jclouds.ssh.SshClient`
[21] fully qualified name: `org.jclouds.compute.domain.ExecResponse`

5. Experimental Evaluation of the Methodology

This chapter aims to evaluate the suggested method of automated acceptance testing for distributed systems. In Section 5.1 criteria for the evaluation are deduced from the key requirements that have been elaborated. Section 5.2 shows how the developed concepts can be applied to a distributed, workflow-driven integration environment for scientific computing (i.e. RCE), and Section 5.3 demonstrates how a distributed database system (i.e. MongoDB) can be tested.

5.1. Evaluation Criteria

Section 4.1.1 introduces the key requirements for test automation of distributed systems. These requirements are used in this section to derive evaluation criteria and discusses to which extent these criteria have been met.

5.1.1. Quality of Tests

One approach to assess the quality of a software testing method is to decompose the quality of testing into four distinctive quality attributes: efficiency, evolvability, exemplarity, and effectiveness (see Figure 2.2). These attributes are used to discuss the level of quality that can be reached with the described testing method.

Efficiency of Testing

The elasticity of virtual infrastructure can be harnessed to achieve an efficient test automation for distributed systems. The promise of virtual infrastructure is that it can be provisioned and released on-demand. Figure 5.1(a) gives an impression of how long it takes until a request for an on-demand infrastructure is fulfilled. On average it takes about 60 seconds to receive a running virtual machine when AWS is used. Rackspace and GoGrid are significantly slower in terms of provisioning time. Releasing the infrastructure after the test execution is typically done within a few seconds (data not shown). Another interesting property of the virtual infrastructure providers is that the provisioning of multiple nodes is not slower than provisioning only few nodes. Figure 5.1(b) summarizes provisioning times measured during the execution of multiple test scenarios and grouped by the number of nodes that were provisioned using AWS. These figures demonstrate

that on-demand provisioning of virtual infrastructure can be used efficiently for the execution of tests. It is viable to include infrastructure provisioning into the automated execution of tests.

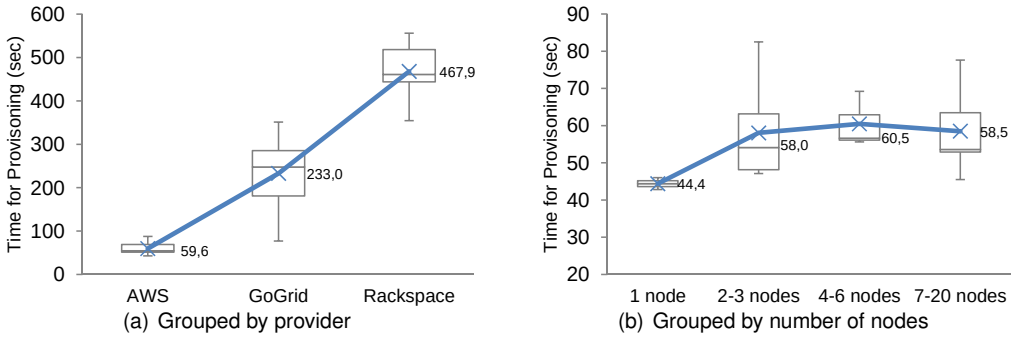


Figure 5.1.: Summary on provisioning times for VMs. Figure 5.1(a) compares different providers and Figure 5.1(b) compares provisioning time for different infrastructure sizes (AWS only).

Another advantage of virtual infrastructure is the a priori knowledge about testing costs that can be simply calculated from the price per machine per hour:

$$\text{testing cost} = \text{machine price/hour} \times \text{number of machines}$$

This calculation assumes that the test execution does not exceed one hour of time and that the infrastructure is only used for one single test. The following calculation demonstrates how costs associated with one test execution that uses ten machines can be determined using the current price of the smallest AWS instance type (micro):

$$\text{testing cost} = 0.02\ \text{USD} \times 10 = 0.20\ \text{USD}$$

A price of two cents per compute hour appears very low but continuous use of one machine over an entire month or year sums up to a price that is higher than that of many long-term contracts:

$$0.02\ \text{USD/hour} = 14.40\ \text{USD/month} = 175.20\ \text{USD/year}$$

This demonstrates that varying demands of infrastructure resources make the usage of on-demand virtual infrastructure particularly interesting.

It is valuable to be able to determine the exact price of a test execution. The costs associated with the execution of an entire test suite are simply calculated as the sums of the costs of the individual tests. It is far more difficult to assign an exact price to a test suite execution if the infrastructure is rented on a monthly basis in combination with a long-term contract.

Another advantage of virtual infrastructure is that tests do not have to be executed sequentially in order to reuse a limited amount of resources, but could be executed in parallel. The entire test suite could be executed in parallel reducing the overall testing time to the time that is required to execute the longest running individual test. Parallel execution is not associated with any additional costs but promises very fast feedback on the system under test. However, it is assumed that a provisioned machine is only used for a single test.

In practice, test execution often takes considerably less time than one hour leaving plenty of paid resources unused. The remaining minutes of every started compute hour can be used to execute additional tests on the same infrastructure. This allows to reduce the cost of an individual test by the number of tests executed per hour:

$$\text{testing cost (reused)} = \frac{\text{machine price/hour} \times \text{number of machines}}{\text{tests/hour}}$$

Such a pricing model is interesting because it reduces testing cost, but also the time required for resource provisioning can be saved in subsequent test executions. However, potential side effects between test executions should be avoided in order to preserve test reproducibility and tests must be efficiently scheduled onto the available resources. If four tests can be executed per hour, which is realistic as shown in the following case studies, the price reduces to five cents per test:

$$\text{testing cost (reused)} = \frac{0.02 \text{ USD} \times 10}{4} = 0.05 \text{ USD}$$

Figure 5.2 demonstrates how the two pricing models translate into infrastructure usage. On the left hand side, only one test is executed per provisioned compute hour. On the right hand side, spare computing resources are used by additional tests. The scheduling of tests onto computing resources would depend on the predicted time that a test needs for execution and the number of nodes that it requires. This is still assuming that all nodes have the same characteristics (e.g. OS, CPU or memory). A discussion of an efficient node pooling and scheduling algorithm is beyond the scope of this work.

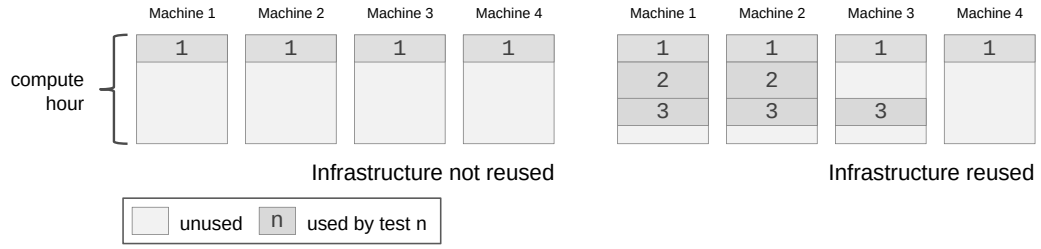


Figure 5.2.: Two models of infrastructure usage. Left: one test per virtual infrastructure. Right: virtual infrastructure is reused for multiple test executions.

Evolvability of Testing

Automating the testing process is expected to have a negative influence on the evolvability of tests compared to manual testing. Manual, ad hoc testing offers the highest degree of flexibility because the tester can react to changes instantly, whereas automated testing requires additional efforts for maintenance (e.g. adoption to the changing interface of the system under test). Therefore the goal should be to minimize the negative influence of test automation on evolvability and maintainability. One design measure to achieve this is the application driver layer that decouples the system under test from the test scenarios. The interface to the system under test is centralized minimizing the

effort necessary to adopt tests to interface changes.

Moreover, the usage of executable test descriptions expressed as an internal Groovy DSL helps to detect many potential test defects. Test defects that stem from syntax errors, from semantic errors on the level of a general-purpose language (e.g. type constraints), and from semantic errors that are domain-specific (e.g. dead lock detection) can be checked by the compiler during the authoring of tests. These checks represent a safety net that allows to constantly restructure and optimize tests without losing confidence in their correctness.

Evolvability is not only an important concern when testing software, but also in the entire software development process itself. Small iterative evolution towards a better product is the foundation of agile software development methodologies. Hence, numerous tools are available for the management of source code. Most of these tools can be applied to test scenarios as well. For instance, tools for the management of source code versioning (e.g. SVN, Git) can be used to version test scenarios, and tools for source code refactoring (e.g. method renaming, reference updating) common in all IDEs can be used to maintain test scenarios, because test scenarios are also valid Groovy programs.

The task-oriented model provides task libraries as a reusable and configurable pool of task descriptions that can be instantiated across multiple test scenarios. This helps to avoid code duplication and facilitates maintenance of the code base similar to source code libraries for software development. The concept of nested tasks complements task libraries. The nesting allows test authors to structure test scenarios into visual and logical blocks that help to understand the scenario better and to encapsulate functionality that might later be migrated into a task library.

Exemplarity of Testing

The economic quality attributes (efficiency and evolvability) of tests are those that can be influenced by automation. Nevertheless, test scenarios should exemplify how the system under test is used. The need for exemplarity was taken into account by relating test scenarios to acceptance criteria and to features. Acceptance criteria are used as examples of specific features of the system under test similar to the specification by example approach [3]. The clear structuring of features, acceptance criteria and test scenarios is also demonstrated in the two case studies in the following Section 5.2 and 5.3. This facilitates the organization of tests in a way that exemplifies how the system under test should be used, but also how it should not be used.

Effectiveness of Testing

The effectiveness of a test is its ability to find defects. As such, it is a property that is difficult to measure on the level of test scenarios because a scenario either finds a defect or it does not. The case studies presented in the following sections aim to demonstrate how little adjustments in the scenario can be used to exercise different aspects of the system under test. This supports the creativity of the test author when designing test scenarios. A test scenario that does not find a defect can easily be modified to scan for defects that could be attributed to the corresponding feature.

After a defect has been fixed, the test scenario that detected the defect should succeed. It is oversimplified to assume that the scenario is ineffective from that point on, because it continues to succeed in every subsequent execution. It is not uncommon that succeeding tests might break as the system is developed further and new features are implemented (i.e. regression defects). In such cases, it is impossible to know if a succeeding test scenario might detect regression defects at some point in the future. This makes it even more difficult to judge about the effectiveness of a test. It might be useful to discard some of the test scenarios that correspond to fixed defects, whereas it might be reasonable to keep others in order to detect potential regression defects. Judgement about test effectiveness remains an experience-based decision by the testers or domain experts. This is also emphasized by the two lists of testing principles introduced in Section 2.1 which characterize testing as “an (extremely) creative and challenging activity”.

5.1.2. Reproducibility of Testing

The described testing method uses new infrastructure components for every test execution, given the simplifying assumption is made that the provisioned infrastructure is not reused for test execution. In this case, nothing exists before test execution and everything will be destroyed after test execution. This is an ideal condition for reproducible test because side effects between multiple tests cannot occur. However, services that the test scenario depends on might substantially influence the test execution. For instance, it was observed that the time required for downloading the system under test from Sourceforge varies greatly. Such influences should be avoided if possible. If the provisioned infrastructure is reused for multiple tests then the application driver layer must provide functionality to avoid side effects between test executions (e.g. deleting all generated files). But even if the infrastructure is reused for a limited number of tests (i.e. the number of tests that can be executed within one hour) this is still a significant difference to a static, physical infrastructure that might use operating systems that were not reinstalled for several weeks or months.

The presented case studies in the following sections demonstrate test scenarios that are designed to install the system under test for every test execution using well-defined configurations for every test execution. Therefore, the test scenarios must contain all information that is necessary to make the system run properly.

Moreover, it is noteworthy that a node should always be dedicated to a single test execution meaning that only one test at a time is executed on a node. From a technical perspective it would be simple to reuse a node for multiple concurrent tests as the usage of dedicated processes for every acting agent provides an isolated environment. However, every node has limited resources which influences the reproducibility of a test. A succeeding test execution that uses small machines exclusively might fail, if multiple tests are executed on the node. One example of this is a system that runs out of heap space if the machines are shared.

A side effect of a high degree of automation is that the deployment of the system under test becomes streamlined over time because it is repeated with every test execution. Issues that arise in the deployment, configuration, and set up process are uncovered early and can be addressed.

From that perspective a test scenario is similar to executable deployment description.

5.1.3. Support for Distributed Systems

The suggested test automation method can be evaluated based on how well it is suited to automate tests for distributed systems. The main idea behind the testing method is to deploy the system under test to a variety of different environments and then exercising the software as described in the test scenarios. Testing using infrastructure that resembles real world scenarios has the potential to quickly uncover implicit assumptions (e.g. zero latency, infinite bandwidth) potentially made during the development process, because these assumptions do not hold any more. Other assumptions can be tested explicitly within test scenarios. Section 5.2.3 discusses a case study which describes a test scenario that executes a distributed workflow while the topology changes (see Section 5.2.3). The assumption of a static topology is one of the eight fallacies of distributed systems development.

Operating System Abstraction

The execution of operations on nodes is handled by agents (see Section 4.4.3). An agent receives commands from the test orchestrator, translates them into operations and executes the operations. In order to be flexible enough and to support as many operating systems as possible, different types of agents can be implemented. During the course of this work an agent was implemented that is based on SSH. The SSH client is used to connect to an SSH demon that acts as an agent. The connection is then used to submit shell commands to the node.

One advantage of using SSH is that it is often pre-installed on UNIX-based VM images and can also be installed on Windows machines. The main disadvantage of this approach is that the shell statements are not portable between operating systems. Therefore, statements must be translated into an appropriate representation before being submitted to the SSH demon. Such an abstraction mechanism[1] is implemented in the jClouds API that renders platform independent statements into platform specific statements. The set of available statements can be extended but the rendering must be defined for every target platform. The following rendering shows how a portable delete statement using the token $\{\mathrm{rm}\}$ is rendered into a UNIX and a Windows version:

$$\text{"}\{\mathrm{rm}\}\ \mathrm{readme.txt}\text{"} \xrightarrow{\text{unix rendering}} \text{"}\mathrm{rm\ readme.txt}\text{"}$$

$$\text{"}\{\mathrm{rm}\}\ \mathrm{readme.txt}\text{"} \xrightarrow{\text{windows rendering}} \text{"}\mathrm{del\ readme.txt}\text{"}$$

An alternative way to achieve platform independency is to implement a Java based agent. The abstraction of the operating system is then handled by the JVM in most parts. The communication with the individual agents could be realized – in the simplest case – as remote method invocation (RMI).

[1] fully qualified name: `org.jclouds.scriptbuilder.domain.Statement`

Testable Distributed Systems

A system that accepts input via command-line interface (CLI) and provides output as log files or on the standard output (stdout) stream is testable in principle. Systems that necessarily require input via a GUI are not considered. Fortunately, in the case of distributed systems this is rarely the case because such systems cannot assume that a GUI is always available on every machine. Additionally, distributed systems are often required to provide an (non-GUI) interface for remote communication in order to enable communication among processes.

The internal implementation and the programming language are irrelevant to the presented testing methodology because the system under test is modelled as black-box test. However, the kinds of tests that can be conducted with a given system are limited by the testability of the software (see Section 4.1.2). For instance, testability can be limited if the CLI does not provide a command to trigger the aspect that should be tested or if the output does not indicate whether the test was successful or not.

5.1.4. Enable Domain Experts

The three intellectual testing activities identify, design, and build (see Figure 4.1) remain a manual task even if test execution is fully automated. Ideally, these tasks can be conducted by domain experts that only have a basic understanding of programming concepts. The three layered reference architecture for acceptance testing described in 4.4 permits a clear distinction between responsibilities in order to fulfil this requirement. The acceptance criteria layer allows phrasing requirements in form of acceptance criteria by specifying parameters for existing test scenarios. No programming is required for this activity. The test implementation layer allows to author tests in form of scenarios using a domain-specific language. On the one hand, the DSL can smooth the learning curve compared to a general-purpose language and uses a task-oriented model to abstract the complexity associated with programming concurrent systems. On the other hand, it opens the possibility to support test authors with additional tools such as domain-specific static analysis to detect test defects automatically.

5.2. Case Study 1: Scientific Computing

As introduced in Section 2.2 distributed systems can be categorized into, a) distributed computing systems, b) distributed information systems, and c) distributed pervasive systems [42]. Distributed computing systems are mainly of two distinct categories. First, high performance computing (HPC) systems that have a clear focus on fast and efficient computation. HPC systems are often characterized by high-end components that form a closely coupled homogeneous system composed of similar hardware. A HPC system is often owned by one organization. Second, grid computing systems that are loosely coupled, heterogeneous systems where nodes are geographically dispersed and potentially owned by multiple different organizations. Grid computing focuses on dynamic sharing of resources across institutions (i.e. virtual organizations).

5.2.1. Software Under Test: Remote Component Environment (RCE)

The RCE[2] framework (Remote Component Environment) [16] is an open source distributed, workflow-driven integration environment that is mainly applied in aerospace and traffic research. It provides a workflow engine for distributed computation workflows that enables multiple research groups from different disciplines to collaborate. In that respect, it fits best in the category of grid computing. RCE is designed as a loosely coupled, decentralized, and modular system that is based on the Eclipse Rich Client Platform and, therefore, the Eclipse OSGi implementation Equinox. It is currently being developed at the German Aerospace Center (DLR).

5.2.2. Feature: F1.1, Hierarchical Overlay Topologies

Multiple RCE instances can be connected with each other to form a homogeneous distributed system. The configured connections between instances create a logical overlay network that contains all established communication channels. RCE is designed to be very flexible in respect of the possible topologies of the overlay network. Point-to-point connections are possible as well as hierarchical topologies. Investigating if RCE provides support for different kinds of topologies is not an easy task as a) a great variety of possible scenarios arises from the provided flexibility and b) tests that model real world scenarios close enough are very time-consuming to set up and typically contain manual steps. The following discussion shows how one type of topology can be tested at different scales. Likewise, additional scenarios could be created in the same manner to test other types of topologies (e.g. peer-to-peer networks).

F1.1	RCE should support distributed workflow execution within hierarchical overlay topologies.
AC1.1.1	A distributed workflow must succeed within a hierarchical overlay topology that has three layers using **5** nodes with version **2.4.1** of the system under test. ➤ TS1.1
AC1.1.2	A distributed workflow must succeed within a hierarchical overlay topology that has three layers using **35** nodes with version **2.4.1** of the system under test. ➤ TS1.1
TS1.1	A distributed workflow must succeed within a hierarchical overlay topology that has three layers using **[Integer]** nodes with version **[String]** of the system under test.

Table 5.1.: Decision table for Feature F1.1 based on Test Scenario TS1.1.

Table 5.1 summarizes two acceptance criteria that make use of a test scenario that represents a three layered topology. The following section discusses how this scenario can be implemented using a test scenario.

Test Scenario: TS1.1, Three Layered Topology

The test scenario should evaluate the capability of RCE to form hierarchical overlay networks and to support distributed workflow execution of multiple workflows within that topology. The complete scenario is listed in Appendix A. The following section discusses some of the interesting parts of the scenario. At first, the system under test is installed on every node using a single line of code.

[2] `http://www.rcenvironment.de`, accessed March 3, 2013

Listing 5.1: Install RCE on all nodes

```
1 sync  'install rce', domain.installRce(group('all'), workingDir, downloadUrl)
```

This is shown in Listing 5.1 where a task group (see Section 4.2.2) is used to apply the task of installing RCE to nodes that are members in the node group all. This group contains all nodes that have been provisioned for the test scenario. The method $\mathrm{instalRce()}$ of the task library domain creates an instance of that task group. The parameter $\mathrm{downloadUrl}$ specifies where RCE can be downloaded which is in this case SourceForge[3]. The scenario does not continue until the installation is completed on every node, because the task group is submitted synchronously using the sync statement.

Once RCE is installed on every node, the topology of the overlay network must be configured. Depending on the number of nodes (n) used to execute the scenario, a topology will be generated that resembles a three layered hierarchy. The top layer consists of one single node (server). The second layer contains one third of the nodes (i.e. $(n-1)/3$) that can communicate with the server, but not with each other (proxies), and the remaining two thirds (i.e. $(n-1)/3*2$) of the nodes (clients) will be randomly connected to one of the proxies. Figure 5.3 illustrates the topology configuration of the RCE overlay network.

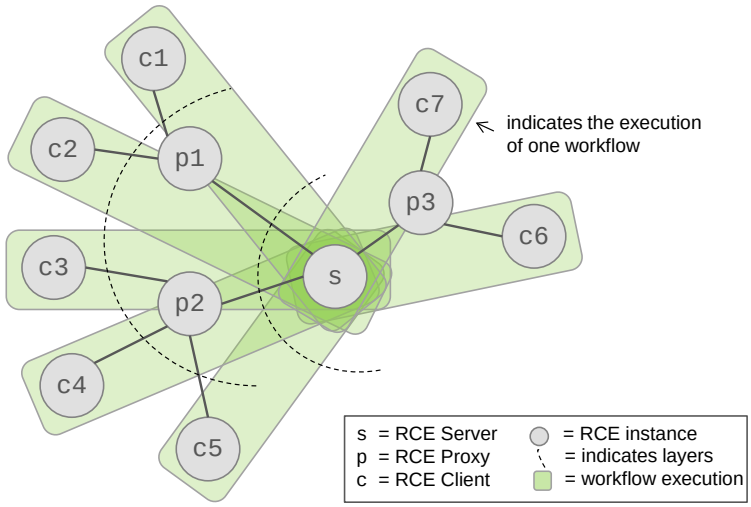


Figure 5.3.: A hierarchical topology of RCE nodes. The first layer contains only the server node s, the second layer contains three proxies $\mathrm{p1}$, $\mathrm{p2}$ and $\mathrm{p3}$ and the third layer contains seven client nodes $\mathrm{c1}$ to $\mathrm{c7}$ that are connected to a random proxy. Every client executes a remote workflow with the server.

For the purpose of configuring the topology, three new node groups are defined: server, client and proxy. The configuration of the three node groups are entirely independent of each other and can thus be executed in parallel. This is realized using three asynchronous tasks. Listing 5.2 shows one such asynchronous task definition illustrating how the server is configured.

[3] `http://sourceforge.net/projects/rcenvironment/`, accessed June 18, 2013

Listing 5.2: Configure the RCE server

```
1 async 'configure server', {
2   // write meta information to execution log
3   sync common.execLog(group('server'), 'server', workingDir)
4   // configure node to provide a network contact point
5   sync domain.configServer(rceDir, group('server'))
6   // make general RCE configurations
7   sync domain.configureRceGeneral(rceDir, group('server'))
8 }
```

Three nested synchronous tasks are defined within the task "server configuration" that decompose the overall task of configuring the server into three smaller tasks. The first one writes information into a log file, the second one applies the topology configuration, and the third one carries out general RCE configurations. All three tasks are anonymous because they do not have a label and translate into writing configuration files on the corresponding nodes. Clients and proxies are configured similarly, but the listings are not shown here for brevity (see Appendix A). The three execution threads created by the three asynchronous tasks are visible in the complete dependency graph of the scenario shown in Figure 5.5.

Once the configuration of the individual node groups is completed, the system under test can be executed. Listing 5.3 contains three statements that orchestrate the software execution on the server group.

Listing 5.3: Run RCE on server node

```
1 await 'configure server'
2 // run RCE on the server node
3 async 'run server', domain.runRce(rceDir, 0, group('server'))
4 // block until RCE runs on server node
5 sync  'assert server runs', domain.assertRceRuns(rceDir,group('server'))
6 // continue only after proxy configuration
```

The first statement awaits the completion of the configuration task described above. This makes sense, because the configuration of RCE should be completed before RCE is started. The second statement executes RCE asynchronously (i.e. "in the background"). This is necessary because the execution of the system under test should happen in parallel to the continuing scenario. The third statement is a retryable distributed assertion (see Section 4.2.3) assuring that RCE actually finished starting up.

As described in the semantic model, retryable assertions are task groups themselves. Internally, the assertion checks a log file for an expected substring to detect, if the system under test started up successfully. The scenario blocks until RCE is ready due to the synchronous execution of the retryable assertion. Then the scenario continues running the system under test on the proxy and client nodes likewise. Starting the system under test in the defined order (first server then proxies and finally clients) is a requirement of the domain. The scenario would not succeed if the ordering was different.

Listing 5.4: Run RCE on proxy and client nodes

```
// continue only after proxy configuration
await 'configrue proxy'
// run RCE on the proxy nodes
async 'run proxy', domain.runRce(rceDir, 2000, group('proxy'))
// block until RCE runs on all proxy nodes
sync  'assert proxy runs', domain.assertRceRuns(rceDir, group('proxy'))
// continue only after client configuration
await 'configrue client'
// run RCE on all client executing a workflow
async 'run rce client', domain.runRce(rceDir, wfFile, 2000, group('client'))
// ensure that every workflow succeeds
sync  'assert wf success', domain.assertWfSuccess(rceDir, group('client'))
// scenario should fail if workflow execution failed
assertThat 'assert wf success'
```

Listing 5.4 shows how the system under test is invoked on proxy and client nodes. The approach is very similar to the execution of RCE on the server node. All four tasks executed in this example are realized as task groups, because they must be applied to multiple nodes. The library method $\mathrm{runRce()}$ accepts a parameter that specifies a time offset in milliseconds. This offset is used as a break between the execution of the individual child tasks of the task group and avoids that the workflow execution is started at the same time on every client. This allows expressing more realistic scenarios that do usually not contain such a precise timing of events.

The task "$\mathrm{run\ rce\ client}$" executes RCE using an additional parameter (i.e. wfFile) that specifies a path to a workflow file that should be loaded and executed after start-up. This causes the RCE clients, unlike the server and proxy nodes, to run a workflow after start-up, which is also indicated in Figure 5.3.

Finally, the last statement in the listing (i.e. "$\mathrm{assert\ wf\ success}$") asserts that the execution of the workflow finished successfully. The RCE workflow uses a compute component on the executing client and another compute component on the remote server. Every client executes its own instance of the workflow. Therefore, the server instance must be capable of managing multiple concurrent workflow executions. Additionally all proxy nodes must properly forward the communication between clients and servers because they are not directly connected. Only if none of these mechanisms fails or malfunctions, the execution of the workflow will be successful on every client.

Criteria Validation Table 5.1 demonstrates how the described test scenario together with specified parameter values can be used to express acceptance criteria. The scenario accepts the number of nodes and the version of the system under test as parameters. This allows expressing a number of different acceptance criteria. For instance, it might be the case that older versions of the system under test only work properly with smaller topologies than newer versions. These kinds of acceptance criteria can be expressed using the described test scenario.

Table 5.2 summarizes five test executions that were conducted with RCE. All five tests were successful. The first and the last row in the table represent the acceptance criteria that were used to evaluate Feature $\mathrm{F1.1}$ in Table 5.1.

criterion	nodes	version	time (sec.)	test result	costs (USD)	data reference
AC1.1.1	5	2.4.1	220.4	success	0.10	☞ 13-05-12-13-07-01
–	10	2.4.1	260.4	success	0.20	☞ 13-05-12-13-27-12
–	20	2.4.1	402.4	success	0.40	☞ 13-05-12-14-41-29
–	30	2.4.1	531.0	success	0.60	☞ 13-05-12-15-03-01
AC1.1.2	35	2.4.1	751.0	success	0.70	☞ 13-05-12-15-19-04

Costs are calculated assuming a price of 0.02 USD per machine hour and a test scheduling model that does not reuse the infrastructure (see details in 5.1.1). References to experimental data see Appendix B.

Table 5.2.: Summary of execution results of Test Scenario TS1.1.

Figure 5.4 demonstrates measured probing information about CPU usage from individual nodes during scenario execution. The test scenarios used a topology with 35 nodes. Three arbitrary client (green) and proxy (orange) nodes where chosen in the graph to represent the respective node groups as well as the server (blue). It is clearly visible that the server is started first, followed by the proxy nodes and finally the client nodes that execute the workflow. The figure demonstrates that the server uses an average of about 40% of its CPU to execute 35 workflows concurrently. This data could be used to formulate additional acceptance criteria that are evaluated after the test scenario has been executed. For instance, a maximum threshold for the average CPU usage of the server node could be defined as acceptance criteria.

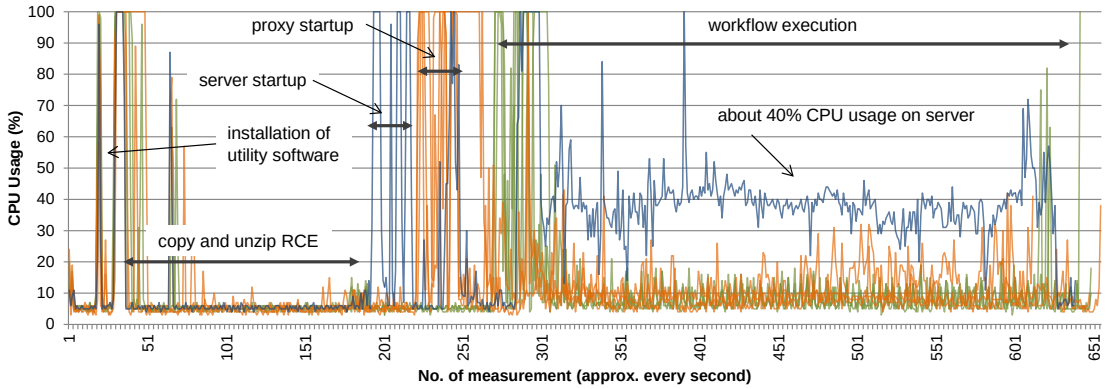


Figure 5.4.: CPU usage profiles of individual nodes while Test Scenario TS1.1 is executing. 35 nodes are used and data of three clients (green), three proxies (orange), and the server (blue) are shown.

The complete dependency graph for this test scenario is shown in Figure 5.5. It represents the execution semantics. The configuration of the RCE server node discussed above (Listing 5.2) is visible as the first forking task execution in the scenario (statement s7). The following statements are s11 that submits the asynchronous task for configuration of the clients, statement s15 that submits the task for configuration of the proxy instances, and statement s18 that is the first synchronization within the scenario. It waits until the configuration of the server has completed before it starts RCE on the server node. After the proxies are configured (statement s21) they are executed and after the clients are configured (statement s24) they are also started. Finally, in the tear down section of the scenario profiling data and log files are gathered and downloaded. Section 4.3.3 describes how a dependency graph is created from a test scenario script.

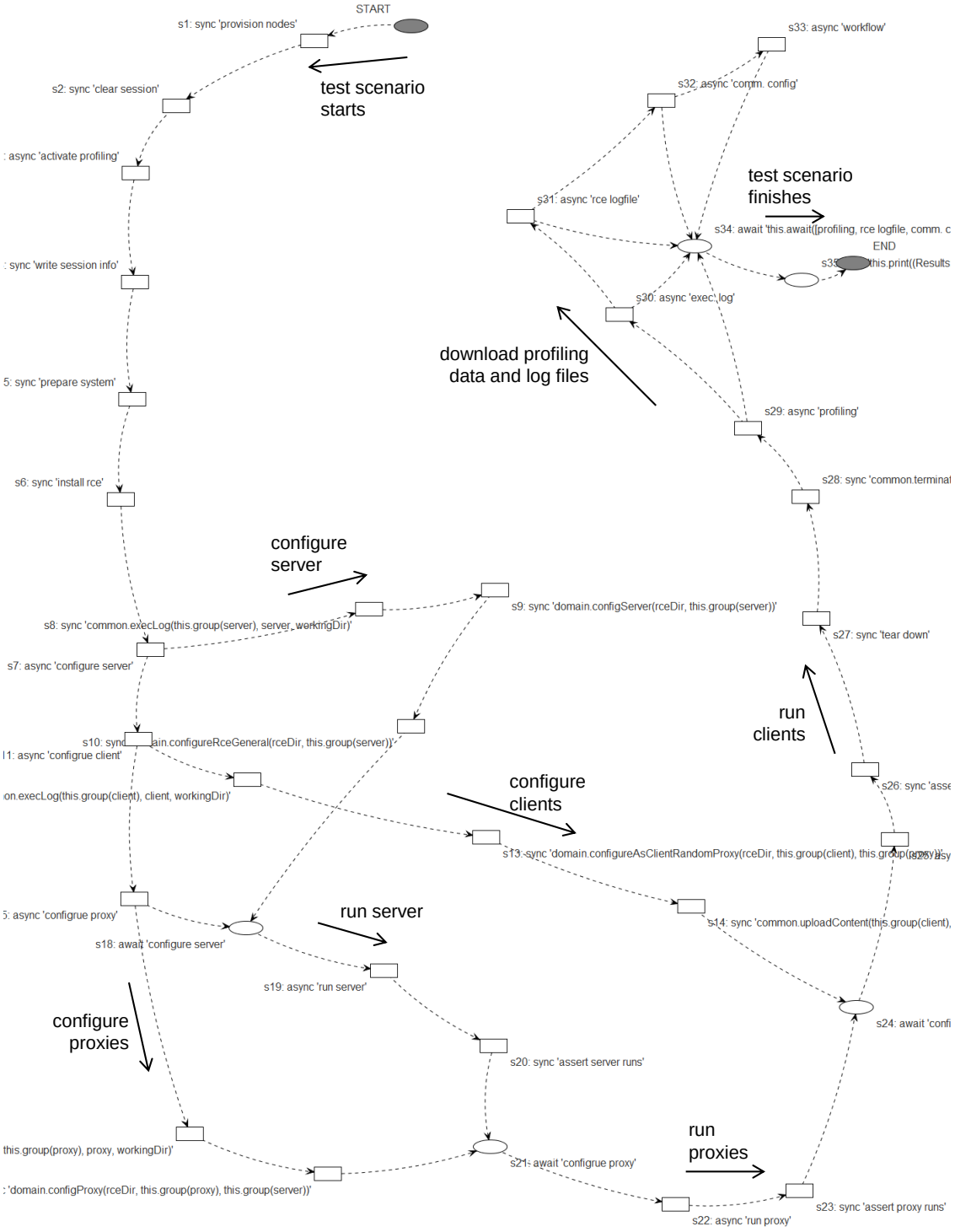


Figure 5.5.: Complete dependency graph representing the execution semantics of Test Scenario TS1.1. The complete source code is available in Appendix A. (Vertices represent statements of the test scenario and edges represent execution dependencies of the test scenario. Rectangular vertices visualize `async` and `sync` statements (i.e. task submission) and round vertices visualize all other statements.)

5.2.3. Feature: F1.2, Reliable Workflow Execution

Challenging to a distributed workflow engine are changes within the overlay network topology that occur while a workflow is executed. A great variety of scenarios is possible influencing workflows that are currently being executed or will be executed shortly after topology changes. For instance, nodes that are part of a workflow might become unavailable or unresponsive during workflow execution or nodes along a communication route might fail. A distributed workflow engine should tolerate node failure or network partitioning during workflow execution to a certain degree. For instance, a long running workflow should not depend on all participating nodes to be permanently available, if some nodes are only necessary in small parts of the workflow. Similar to Feature $\mathrm{F2.1}$ that evaluates the ability of RCE to form different overlay topologies, this feature is difficult to evaluate and it is not easy to specify what degree of fault tolerance is provided or is not provided by the system under test.

$\mathrm{F1.2}$	RCE should be able to execute distributed workflows in a dynamic, changing overlay network topology.
$\mathrm{AC1.2.1}$	A distributed workflow must **succeed**, if a connecting RCE node **shuts down** shortly before workflow execution. ➤ $\mathrm{TS1.2}$
$\mathrm{AC1.2.2}$	A distributed workflow must **succeed**, if a connecting RCE node **crashes** shortly before workflow execution. ➤ $\mathrm{TS1.2}$
$\mathrm{AC1.2.3}$	A distributed workflow must **succeed**, if a connecting RCE node **shuts down** during workflow execution. ➤ $\mathrm{TS1.3}$
$\mathrm{AC1.2.4}$	A distributed workflow must **not succeed**, if a connecting RCE node **crashes** during workflow execution. ➤ $\mathrm{TS1.3}$
$\mathrm{TS1.2}$	A distributed workflow must **[not succeed|succeed]**, if a connecting RCE node **[shuts down|crashes]** shortly before workflow execution.
$\mathrm{TS1.3}$	A distributed workflow must **[not succeed|succeed]**, if a connecting RCE node **[shuts down|crashes]** during workflow execution.

Table 5.3.: Decision table for Feature $\mathrm{F1.2}$ based on Test Scenario $\mathrm{TS1.2}$ and $\mathrm{TS1.3}$.

Table 5.3 shows Feature $\mathrm{F1.2}$ that describes the requirement of RCE to be able to handle topology changes before or during distributed workflow executions. Four acceptance criteria are used to evaluate the feature. The first two use a test scenario that investigates the workflow execution shortly after topology changes occurred ($\mathrm{TS1.2}$), the third and fourth use a similar test scenario that evaluates workflow execution *while* the topology changes ($\mathrm{TS1.3}$). Both scenarios are discussed in the following sections.

In order to develop a test scenario that evaluates the reliability of the workflow execution during and shortly after topology changes, a small distributed arrangement with four RCE nodes can be used as shown in Figure 5.6. One client RCE instance (c) initiates the distributed workflow that uses a component provided on the server RCE instance (s). Client and server are not directly connected but use the intermediate RCE nodes $\mathrm{p1}$ or $\mathrm{p2}$ for communication. If both of the proxies are working properly then client and server have two independent routes available to communicate with each other. This diamond shaped arrangement can be used to evaluate the four acceptance criteria from Table 5.3.

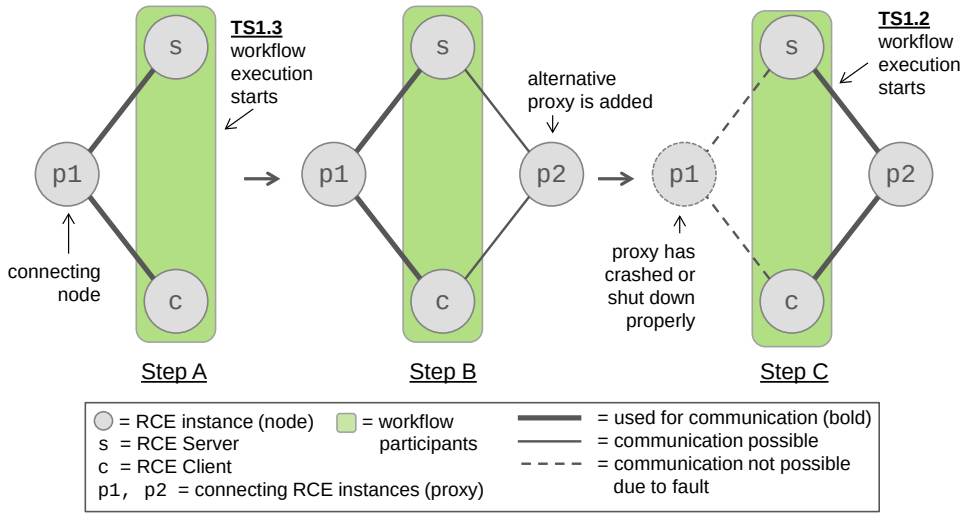


Figure 5.6.: A three step overview of Test Scenario $\mathrm{TS1.2}$ and $\mathrm{TS1.3}$. Step A: client c and server s are connected via proxy $\mathrm{p1}$. Step B: a second proxy $\mathrm{p2}$ joins the network. Step C: the first proxy $\mathrm{p1}$ crashes or shuts down leaving $\mathrm{p2}$ as an alternative communication route.

Test Scenario: TS1.2, Topology Changes Shortly Before Workflow Execution

Figure 5.6 shows how $\mathrm{AC1.2.1}$ and $\mathrm{AC1.2.2}$ can be validated using three steps. First, all nodes are initialized, the system under test is installed, and the topology is configured. This is not shown as a separate listing, but is available in Appendix A. Then, in step A, the server is started followed by proxy $\mathrm{p1}$ and the client. Afterwards, in step B, the second RCE proxy instance joins the overlay network offering an alternative communication channel for client and server.

Listing 5.5: Conditional task submission

```
1 sync 'cause fault', {
2   // skip if scenario has failed already
3   if (failure) return
4   if (shutdownInsteadOfCrash)  sync 'shut down first proxy', domain.shutdownRce(group('first'))
5   if (!shutdownInsteadOfCrash) sync 'crash first proxy', domain.killRce(group('first'))
6 }
```

Finally, in step C, the first proxy instance is shut down gracefully using the stop command that the interactive RCE command-line provides ($\mathrm{AC1.2.1}$) or it is crashed by killing the RCE process ($\mathrm{AC1.2.2}$). This is also shown in Listing 5.5 where two synchronous task submissions are executed conditionally based on the acceptance criterion. Right after the first proxy ($\mathrm{p1}$) has terminated, the client starts executing a workflow. The test scenario is successful if the workflow was successful. The complete test scenario is shown as a dependency graph in Figure C.1.

Criteria Validation The scenario can only succeed if RCE can cope with changes in the overlay topology. The message routing mechanism must be aware of the unavailable first proxy early enough in order to avoid communication attempts towards that instance. Also, it must be aware of the alternative route that has been added by the second proxy only shortly before the workflow starts.

criterion	workflow succeeds	shut down/ crash	time (sec.)	test result	costs (USD)	data reference
$\mathrm{AC1.2.1}$	yes	shut down	322.3	success	0.08	☞ 13-07-11-16-31-46
$\mathrm{AC1.2.1}$	yes	shut down	324.9	success	0.08	☞ 13-07-11-16-53-03
$\mathrm{AC1.2.2}$	yes	crash	333.5	success	0.08	☞ 13-07-11-16-39-03
$\mathrm{AC1.2.2}$	yes	crash	324.9	success	0.08	☞ 13-07-11-16-53-03

Costs are calculated assuming a price of 0.02 USD per machine hour and a test scheduling model that does not reuse the infrastructure (see details in 5.1.1). References to experimental data see Appendix B.

Table 5.4.: Summary of execution results of Test Scenario $\mathrm{TS1.2}$.

The two acceptance criteria evaluate two different severities of topology changes. $\mathrm{AC1.2.1}$ allows the first proxy to gracefully shut down which causes the RCE instance to inform all other instances about its soon shut down. Contrary, $\mathrm{AC1.2.2}$ terminates the RCE process which represents a crash of the system under test. In this case the other instances must detect that the first proxy is unavailable by themselves, which is an entirely different mechanism. Table 5.4 shows that both acceptance criteria succeed.

Most of the acceptance criteria presented here use AWS Micro Instances ($\mathrm{t1.micro}$) that are the smallest machines available. They have one vCPU (virtual CPU), 0.615 GiB main memory, a "very low" network performance, and a 64-bit processor architecture. The Acceptance Criterion $\mathrm{AC1.2.1}$ was used to investigate how stronger computing nodes influence the overall execution time of a test scenario. It was executed four times using the micro instances and four times using the AWS general-purpose M1 Instances ($\mathrm{m1.large}$) that have two vCPUs, 7.5 GiB main memory, a "moderate" network performance, and a 64-bit processor architecture[4]. The average execution time for $\mathrm{AC1.2.1}$ on micro instances was 341.3 seconds, whereas a scenario took on average 256.7 seconds using the significantly larger instances. This represents a 33% faster scenario execution on the large nodes but the price per hour is twelve times higher compared to a micro instance (micro: 0.02 USD, large: 0.24 USD).

Test Scenario: TS1.3, Topology Changes During Workflow Execution

Test Scenario $\mathrm{TS1.2}$ described above can be slightly altered to investigate how RCE handles topology changes *during* workflow execution. The same basic configuration as shown in Figure 5.6 can be used. The only difference is the point in time at which the workflow is started. It is not started in step C but already in step A before the topology changes are caused. Then the scenario continues like $\mathrm{TS1.2}$ and it succeeds if the workflow succeeds.

The test scenario is designed so that the workflow starts before the second proxy is available. This guarantees that the first proxy is used for communication. Otherwise, RCE might choose any of the two available routes for communication which would cause the test to succeed always if $\mathrm{p2}$ had been chosen and to potentially fail if $\mathrm{p1}$ is chosen. Such a scenario would be unsuitable because the test result is not reproducible. Another source for irreproducibility is the time required for the execution of the workflow. If the workflow runs too quickly then it might finish before step

[4] `http://aws.amazon.com/ec2/instance-types/`, accessed July 12, 2013

C is reached which causes an unintended success. In order to avoid unintended success of tests (false positives) an additional assertion is inserted into the scenario assuring that the workflow was still executing by the time the fault is caused on $\mathrm{p1}$. This is also shown in Listing 5.6. The complete test scenario is shown as a dependency graph in Figure C.2.

Listing 5.6: An assertion making sure that the workflow still executes

```
1  // assert that the client is still executing the workflow
2  sync  'wf still running', domain.assertWfStillRunning(rceDir, group('client'))
3  // fail if workflow to quickly finished
4  assertThat 'wf still running'
```

Criteria Validation Obviously this test scenario represents a more challenging requirement to RCE. In fact, the execution of Acceptance Criterion $\mathrm{AC1.2.3}$ fails as illustrated in Table 5.5 although a graceful shut down is used. It is known that current versions of RCE (e.g. 2.5.2) cannot reliably handle topology changes along the communication route of a running workflow, which is expressed by the test failure. Nevertheless, the test is very useful as it is likely that such a degree of resilience to topology changes will be required in the near future. Therefore, the criterion can be used as a "test first" example that will allow to approve future implementations.

criterion	workflow succeeds	shut down/ crash	time (sec.)	test result	costs (USD)	data reference
$\mathrm{AC1.2.3}$	yes	shut down	363.0	failure	0.08	☞ 13-07-11-17-25-07
$\mathrm{AC1.2.3}$	yes	shut down	422.6	failure	0.08	☞ 13-07-11-17-34-39
$\mathrm{AC1.2.4}$	no	crash	419.3	failure	0.08	☞ 13-07-11-17-34-39
$\mathrm{AC1.2.4}$	no	crash	402.0	failure	0.08	☞ 13-07-11-17-54-02

Costs are calculated assuming a price of 0.02 USD per machine hour and a test scheduling model that does not reuse the infrastructure (see details in 5.1.1). References to experimental data see Appendix B.

Table 5.5.: Summary of execution results of Test Scenario $\mathrm{TS1.3}$.

The last Acceptance Criterion $\mathrm{AC1.2.4}$ uses a fault of the connecting proxy instead of a shut down. This further aggravates the situation, because the failing node has no chance to announce its shut down. The communicating nodes would have to become aware of the unavailability of the proxy which is typically only possible by a maximum threshold for the response time. It is known that RCE does not provide such a degree or resilience and it is also not a requirement to implement mechanisms that attempt to reach this. This is also expressed by $\mathrm{AC1.2.4}$ because the workflow is expected to *not* succeed. This is a way to demonstrate what a system should not be capable of as required by testing principle M6.

It is noteworthy that this criterion only succeeds in cases of a "successful" workflow failure. The workflow execution can fail due to many different causes that all lead to a failure of the workflow, but should not cause a failure of the entire system (i.e. RCE instance). Thus, if the RCE client continues to operate properly after a workflow failed the test should succeed. As shown in Table 5.5 the tests fails. The is because RCE does not provide output that would allow to judge about whether the workflow failure was handled properly or not, which demonstrates a situation where the testability is limited because of insufficient output (see Section 4.1.2).

5.3. Case Study 2: Distributed Databases

Another type of distributed systems is distributed databases that can be classified as distributed information systems. It is known that a distributed database system can only have two of the three properties: strong consistency, high availability and partition tolerance (CAP theorem). Most distributed database systems either focus more on a strong consistency model or on high availability. Systems that provide strong distributed consistency are often realized using transactions that are atomic, consistent, durable, and isolated (ACID), whereas systems that offer high availability use less constrained consistency models (BASE). In this case study it is demonstrated how two features of the distributed database system MongoDB, that implements a loosened consistency model, can be evaluated: distributed consistency in case of node failure ($\mathrm{F2.1}$), and high write availability in case of node failure ($\mathrm{F2.2}$).

5.3.1. Software Under Test: MongoDB

MongoDB (from “humongous database”) is a schema-free, document-oriented database system that supports search by field, by regular expression and by range queries. It can be configured as a distributed system where nodes are communicating MongoDB instances grouped into a replica set. The replica set performs a master-slave replication of the data. The whole system appears as one data storage to the user. Write commands are issued to the primary node and read commands can be issued to any node of the replica set. Replication of data to the secondary nodes is asynchronous. This design offers a highly available system that accepts write input even in the presence of network partitioning or node failure. However, this is achieved by sacrificing a strong consistency model. The database might have an inconsistent state in between the moment, when data has been written to the primary node and the moment, when the data is replicated to all secondary nodes. Therefore, this distributed consistency model is also referred to as eventual consistency.

5.3.2. Feature: F2.1, Eventual Distributed Consistency

MongoDB implements a fault tolerance mechanism that allows the primary (master) instance to continuously accept write commands while parts of the system fail. The tolerance of failures is achieved using redundancy. Instances of MongoDB services can be grouped together into replica sets. A replica set of size k is composed of one primary (master) node and $k-1$ secondary (slave) nodes. The primary node accepts write operations to the database and causes the written data to be replicated to all secondary nodes. Read access is possible from any node in the replica set, given an according configuration (i.e. $\mathrm{setSlaveOk}()$).

If a node becomes unavailable then the primary node continues to accept write input asserting the availability of the system. If the primary node itself becomes unavailable then a new MongoDB primary instance is elected. The election is achieved by a majority vote of all nodes of the replica set. Therefore, the overall system tolerates failure of arbitrary nodes (primary or secondary), as long as the majority (more than $\lceil k/2 \rceil$) of nodes did not fail. Put differently, the system tolerates up

F2.1	A MongoDB replica set must eventually achieve distributed consistency, even if up to $\lceil k/2 \rceil - 1$ instances temporarily fail. ($k = \text{replica set size}$)
AC2.1.1	A replica set with **3** members should **achieve** distributed consistency even if **1** node(s) fail(s) temporarily. ➤ TS2.1
AC2.1.2	A replica set with **6** members should **achieve** distributed consistency even if **2** node(s) fail(s) temporarily. ➤ TS2.1
AC2.1.3	A replica set with **3** members should **not achieve** distributed consistency even if **2** node(s) fail(s) temporarily. ➤ TS2.1
TS2.1	A replica set with **[Integer]** members should **[achieve|not achieve]** distributed consistency even if **[Integer]** node(s) fail(s) temporarily.

Table 5.6.: Decision table for Feature F2.1 based on Test Scenario TS2.1.

to $\lceil k/2 \rceil - 1$ crash failures at any given time. This feature is expressed in the Table 5.6 using three acceptance criteria. The Acceptance Criteria AC2.1.1 and AC2.1.2 are examples of scenarios that should be supported by MongoDB, because the number of failing nodes is less than $\lceil k/2 \rceil$. On the contrary, Acceptance Criterion AC2.1.3 represents a degree of fault tolerance that is not expected from a MongoDB replica set. All acceptance criteria are founded on Test Scenario TSTS2.1 that is described in the following section.

Test Scenario: TS2.1, Consistency in Case of Failure

In order to evaluate the fault tolerance of MongoDB, a test scenario can be used that causes crash failures on purpose and issues write commands while nodes fail. Figure 5.7 gives a four step, high level description of such a scenario. Step A shows how a replica set with five members is set up. The size of the replica set is one of three parameters that the scenario accepts. MongoDB supports replica sets that contain up to twelve nodes. In this case, the second node has been elected as primary node and all others are secondary nodes.

Step B causes a crash failure of a given number of nodes by terminating the mongod service on the corresponding nodes using the UNIX kill command. Listing 5.7 shows how this step is encoded in the test scenario. First, two groups are defined to reference failing and healthy nodes. Then MongoDB is terminated on the corresponding group and a distributed assertion is used to ensure that the election of a new primary node was successful from the perspective of every node. The assertion fails if the replica set is unable to elect a new primary node. This is typically the case when more than $\lceil k/2 \rceil - 1$ nodes fail. If a successful primary election could not be confirmed then the entire test scenario should fail. This is achieved with the assertThat command that sets the scenario state to failure, depending on the result of the distributed assertion. In the depicted example this assertion succeeds, because after node 1 and 2 failed, node 3 could be elected as the new primary node (see step B).

In step C the new primary node is determined, data is written to that primary node, and the data is asynchronously replicated to the remaining, healthy nodes (4 and 5). Afterwards in step D the failed nodes are restarted and the written data is replicated to those nodes eventually achieving distributed consistency in respect of the written data. To assert success of the test scenario, the data is read from every node in the replica set and checked for consistency. If this was successful

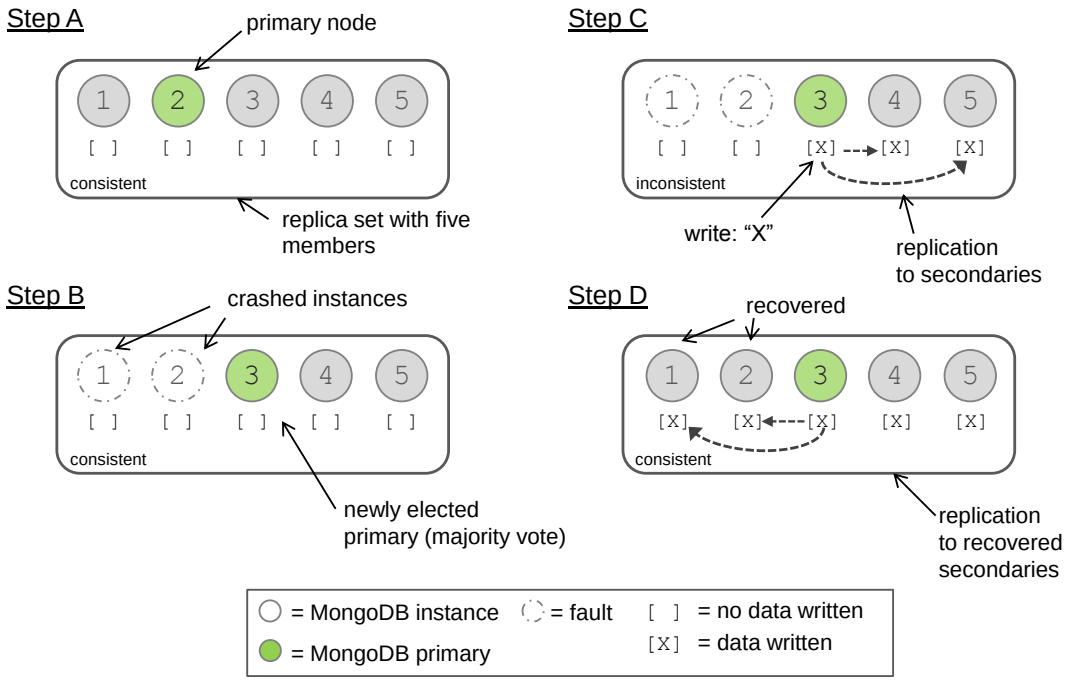


Figure 5.7.: A four step overview of Test Scenario TS2.1. A MongoDB replica set with five nodes exemplifies how Feature F2.1 can be evaluated.

then the overall scenario succeeds. The complete scenario is listed in Appendix A.

Listing 5.7: Grouping of nodes and reproducible partial failure of a distributed system

```
1  // create group with arbitrary nodes as failing nodes
2  group 'failing', group('all').any(failingCount)
3  // create group that contains healthy nodes
4  group 'healthy', group('all') - group('failing')
5  // kill all services on nodes in group 'failing'
6  sync  domain.killService(group('failing'))
7
8  if (toleranceExpected) { // tolerance expected
9    // make sure a primary was elected
10   sync  'assert primary election', domain.assertOnePrimary(group('healthy'))
11   // scenario should fail if no primary can be found
12   assertThat 'assert primary election'
13 }
```

Listing 5.7 shows how the concept of node groups can be used to express the desired test scenario. As shown, the group of failing nodes is created by a random selection of nodes in the group of all nodes. The scenario could be slightly adjusted to select the failing nodes based on another group. For instance, it would be easy to guarantee that the primary node is not a member of the failing group, or that it is always a member of the failing group by small adjustments in the node groups. Another idea how the given scenario could be altered is to let only the primary node fail. Once a new primary node is elected this node could then be crashed, which causes a re-election again. This could be repeated until no more than $\lceil k/2 \rceil + 1$ MongoDB instances are running in order to exercise the primary election mechanism. In summary, this demonstrates how testing of a variety of different scenarios is possible with only small modifications of the source

code. Tests that have revealed defects can therefore be altered to “harvest” for additional defects (see principles M9 and B9).

Criteria Validation Table 5.7 summarizes observations from some test executions. Among them are the three acceptance criteria expressed in Table 5.6 in the first three rows. All three criteria succeed.

criterion	replica set size	failing nodes	achieve consist.	time (sec.)	test result	costs (USD)	data reference
$\mathrm{AC}2.1.1$	3	1	yes	231.8	success	0.06	☞ 13-06-24-15-44-09
$\mathrm{AC}2.1.2$	6	2	yes	435.6	success	0.12	☞ 13-06-24-16-04-23
$\mathrm{AC}2.1.3$	3	2	no	139.6	success	0.06	☞ 13-06-24-16-12-58
–	12	5	yes	358.6	failure	0.24	☞ 13-06-24-16-28-36
–	7	3	yes	177.1	success	0.14	☞ 13-06-24-16-36-19

Costs are calculated assuming a price of 0.02 USD per machine hour and a test scheduling model that does not reuse the infrastructure (see details in 5.1.1). References to experimental data see Appendix B.

Table 5.7.: Summary of execution results of Test Scenario $\mathrm{TS}2.1$.

The fourth row is the result of a test execution that uses a replica set with twelve members that should tolerate failure of five ($\lceil 12/2 \rceil - 1 = 5$) instances at the same time. The scenario fails, although the number of failing nodes is less than half of the replica set size. This is due to the maximum of allowed voting members in a MongoDB replica set, which is seven. Thus, no more than three (voting) nodes are allowed to fail even when a replica set has more than seven members. This finding indicates how a test scenario might not only facilitate the discovery of code defects or design defects but also specification defects. The described feature should contain additional information about the maximum value of k, that is not twelve but seven.

The last row in the table represents a situation where the maximum number of concurrent faults that MongoDB can tolerate (i.e. $\lceil 7/2 \rceil - 1 = 3$) is tested successfully.

5.3.3. Feature: F2.2, High Write Availability

A promise of MongoDB, but also of many other non-relational databases, is that a distributed installation of the system is highly available for write operations. This property is maintained even if the network is temporarily partitioned. This is achieved by loosening the consistency constraints compared to transaction-based systems with ACID characteristics. A write to a distributed transaction-based system can typically only succeed when the network is not partitioned and all nodes are available. Locking mechanisms assert that consistency is guaranteed but effectively limit availability in larger or less reliable systems.

Because MongoDB sacrifices the strong consistency model, it should not suffer from a reduced availability in an unreliable environment. Nodes that are not available during a write operation must be updated at a later point in time when they are available again. It is not worth anything if the updating mechanism slows down the overall system and reduces the availability to a similar degree as locking mechanisms in transaction-based systems do. Feature $\mathrm{F}2.2$ in Table 5.8 ex-

F2.2	The write availability of a MongoDB replica set should not be influenced, if members of the replica set become temporarily unavailable.
AC2.2.1	A replica set with **3** members should not be more than **10**% slower when receiving continuous writes, if **1** member(s) of the replica set is/are temporarily unavailable (using **1000** write operations at **100** bytes). ➤ TS2.2
AC2.2.2	A replica set with **5** members should not be more than **10**% slower when receiving continuous writes, if **2** member(s) of the replica set is/are temporarily unavailable (using **1000** write operations at **100** bytes). ➤ TS2.2
AC2.2.3	A replica set with **7** members should not be more than **10**% slower when receiving continuous writes, if **3** member(s) of the replica set is/are temporarily unavailable (using **1000** write operations at **100** bytes). ➤ TS2.2
TS2.2	A replica set with **[Integer]** members should not be more than **[Integer]**% slower when receiving continuous writes, if **[Integer]** member(s) of the replica set is/are temporarily unavailable (using **[Integer]** write operations at **[Integer]** bytes).

Table 5.8.: Decision table for Feature F2.2 based on Test Scenario TS2.2.

presses this requirement by stating that writing speed should not be reduced if the infrastructure is less reliable.

Test Scenario: TS2.2, High Availability in Case of Failure

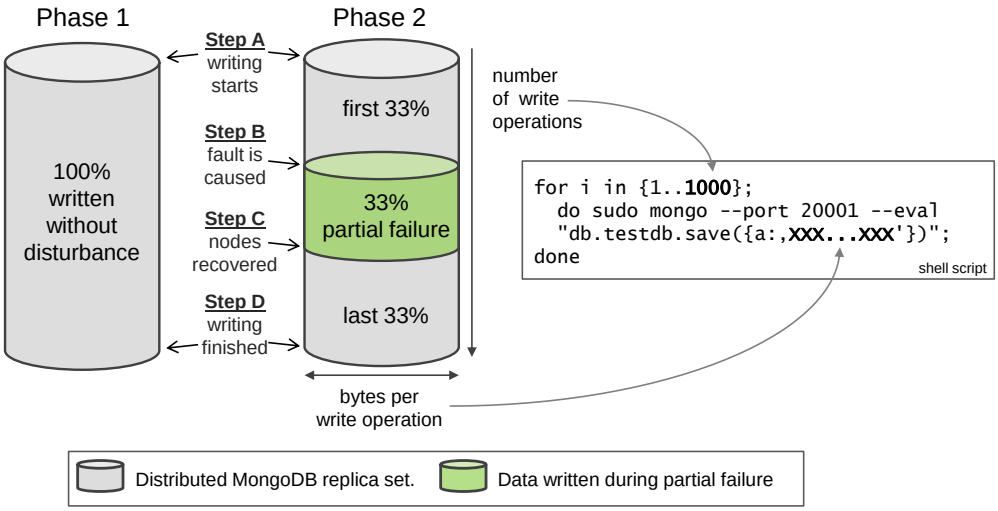


Figure 5.8.: An overview of Test Scenario TS2.2. Step A: the MongoDB replica set is initialized and writing data starts with an empty database. Step B: after one third of the data is written to the system, some instances are terminated. Step C: if two thirds of the data are written then the terminated instances are restarted. D: The scenario finishes after all data is written.

Feature F2.2 can be evaluated using a test scenario that compares write availability of a MongoDB replica set in a phase where no failures occur (phase 1) to a phase with failures (phase 2). Both phases must be otherwise identical so that other influences can be excluded. The time required for a defined number of writing operations serves as measurement for the write availability. The two measured time spans can then be compared using their relative difference:

$$\frac{\text{phase } 2 - \text{phase } 1}{\text{phase } 1}$$

A value of 0.1 would mean that the second phase took 10% longer than the first phase. A negative

result represents a situation where the writing phase with disturbance was faster than the one without disturbance. Every write operation is executed using a new connection to the local mongo service. Thus, a connection to the services is established and terminated for every write operation. This would be a very inefficient way to store larger chunks of data, but represent situations where multiple independent small write operations occur in quick succession.

Figure 5.8 provides a high level overview of the test scenario and its two phases. In the first phase nothing significant happens apart from the writing process itself. The listing of that phase is omitted here but is available in Appendix A. The second phase starts similar to the first phase. A writing stream is created using an asynchronous task. This is the first statement in Listing 5.8. The figure shows the shell script submitted by the SSH agent when executing the task "stream 2". It causes the writing load on the primary node by executing a loop.

Listing 5.8: Writing stream that is disturbed by node failures (step A and B)

```
1  // start writing stream asynchronously
2  async 'stream 2', domain.writeToDbOften('tmp', writeOps, fixture, group('prim'))
3  // wait until 1/3 of the data volume is written
4  sync 'size 1/3', domain.assertDbSizeLargerThan(round(writeOps/3), 'tmp', group('all'))
5  // create group with arbitrary nodes as failing nodes
6  group 'failing', group('sec').any(failingCount)
7  // terminate MongoDB processes
8  sync  domain.killService(group('failing'))
```

Meanwhile, the writing process continues to submit a distributed assertion ("size $1/3$") synchronously in order to block the execution until every MongoDB instance has received at least one third of the data volume. It is noteworthy that the scenario execution is synchronized based on the data volume instead of a fixed time period. Using a time period would be rather unsuitable because it depends on many external factors (e.g. machine dimensions). When the threshold is reached a failure of some nodes is caused using a similar mechanism as in TS2.1 (step B).

Listing 5.9: Restarting failed nodes and await termination of writing process (step C and D)

```
1  // wait until 2/3 of the data volume is written
2  sync 'size 2/3', domain.assertDbSizeLargerThan(round(writeOps/3*2), 'tmp', group('prim'))
3  // restart MongoDB services
4  async domain.startService(group('failing'))
5  // synchronize writing stream 2
6  await 'stream 2'
```

The failed MongoDB instances remain unavailable until the next threshold at two thirds of the data is reached (step C). This is done similarly to step B above and is shown in Listing 5.9. Then the instances are restarted as shown in the second statement in the code listing. After the nodes reappear in the replica set, they must be updated with one third of the data that was written during their absence. This updating mechanism must complete its work while additional data is written (the last 33% of the data volume). Ideally, updating the recovered nodes should not influence the write availability of the primary node.

Finally, in step D it is waited until all data was written. This is also shown as the last await statement in the code listing. The test scenario continues to calculate the relative time differences

in order to determine test success or failure (Listing 5.10) and downloads probed data and log files.

Listing 5.10: Test acceptance criterion

```
1  assertThat delayRelative < toleratedDelay
```

The scenario is customizable using five different parameters: the size of the replica set, the number of failing nodes, a relative threshold that represents the accepted write time deviation, the number of write operations and the number of bytes per write operation. These parameters allow to express a variety of different quantitative acceptance criteria that focus on write availability.

Further test scenarios can be derived from this scenario to explore similar applications of the system under test. One possibility would be to instantiate and submit the task used for the asynchronous data writing ($\mathrm{writeToDbOften}$) more than once. This would allow to use multiple concurrent writing streams. This idea could be further extended to a scenario where the writing processes are migrated from the node that hosts the MongoDB primary service to dedicated nodes in order to create a higher load on the primary node.

Criteria Validation Table 5.9 shows results of some selected test executions of Test Scenario $\mathrm{TS2.2}$ that correspond to the acceptance criteria introduced in Table 5.8. Replica sets with three, five and seven members are shown using one, two and three failing nodes respectively. The tolerated additional time for the situation with failing nodes (phase 2) is set to 10% which succeeds in every shown test execution. If acceptance criteria are formulated using quantitative thresholds then careful attention must be paid to the reproducibility of the test scenario. For instance, the deviation of the measured delay must be taken into account when determining an acceptance threshold. Otherwise, the reproducibility would not be satisfying because the success and failure of a test scenario might vary although input values do not change. The difference in writing speed that is used as metric for an acceptance threshold can potentially be influenced by many factors. Examples include other processes running on the node, side effects between the two test phases, or discrepancy in the network bandwidth or throughput.

criterion	nodes / failing	thresh.	writes / bytes	time (sec.)	test result	costs (USD)	data reference
$\mathrm{AC2.2.1}$	3 / 1	10%	1000 / 100	530.6	success	0.06	☞ 13-06-25-18-54-34
$\mathrm{AC2.2.2}$	5 / 2	10%	1000 / 100	516.7	success	0.10	☞ 13-06-25-19-20-12
$\mathrm{AC2.2.3}$	7 / 3	10%	1000 / 100	481.6	success	0.14	☞ 13-06-25-19-33-13

Costs are calculated assuming a price of 0.02 USD per machine hour and a test scheduling model that does not reuse the infrastructure (see details in 5.1.1). References to experimental data see Appendix B.

Table 5.9.: Summary of execution results of Test Scenario $\mathrm{TS2.2}$.

In order to investigate the reproducibility of the writing speed measurements, the parameters specified by the three Acceptance Criteria $\mathrm{AC2.2.1}$, $\mathrm{AC2.2.2}$ and $\mathrm{AC2.2.3}$ in Table 5.8 were used to repeat test execution five times in a row for every criterion. The reproducibility of the relative difference in write availability is particularly interesting. The means of the measured values are

for three nodes +3.7% (deviation: 2.0%), for five nodes +4.5% (deviation: 0.9%), and for seven nodes +2.8% (deviation: 1.6%) which indicates that a threshold of 10% is reasonable. Interestingly, the write availability of MongoDB is decreased by 3.7% on average in this experiment (15 measurements). The relatively small deviations that are observed demonstrate the reproducibility of the measurements, given the many potential disturbing influences. The threshold of 10% was not exceeded in any of the test executions.

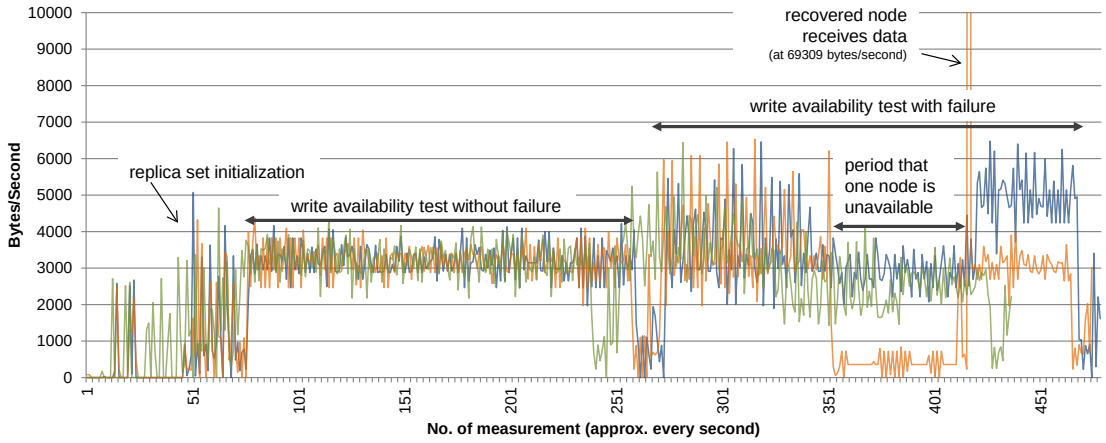


Figure 5.9.: Incoming network traffic on nodes of a MongoDB replica that is used to performing Test Scenario $\mathrm{TS}2.2$ with parameters from Acceptance Criterion $\mathrm{AC}2.2.1$. The primary node (green), the failing secondary (orange), and the other secondary (blue) are shown.

Deeper insights into the execution of test scenarios are possible if the collected metrics on CPU and memory utilization as well as incoming and outgoing network traffic can be investigated. The data can be analysed directly after the test execution. Figure 5.9 shows a graph that summarizes outgoing network traffic when $\mathrm{AC}2.2.1$ is executed that uses a replica set with three members and one failing node. The communication required for the initialization of the replica set and during the two testing phases is visible. In the first phase all writes are conducted without node failure, in the second phase writes are repeated with node failure. The period where one node fails is directly visible because the incoming network traffic on the failing node drops substantially. The MongoDB instances do not receive any updates from the primary node. After recovery, the node is updated very quickly which is demonstrated by the large peak. The timing of the failure and recovery events triggered at one third and two thirds of the writing process are reflected in the measured network traffic. The x-axis only represents the number of the measurements that were taken which translates roughly into the seconds after scenario start. Although all nodes start probing at the same time and have the same probing interval of one second, the measurement time points start to deviate on the nodes as the scenario continues.

6. Summary

The development of distributed systems is challenging because of their complexity. The eight fallacies of distributed computing that summarize common false assumptions (e.g. the network is reliable) give an impression of how the development of distributed systems differs from that of centralized systems. Therefore, testing of distributed applications deserves at least as much attention as that of centralized systems. It is desirable to achieve a high degree of automation when testing typical properties of distributed systems. Examples include fault and partition tolerance, distributed consistency, distributed workflow execution, and failure recovery mechanisms. A high degree of automation is not only important on lower testing levels (i.e. unit and module testing), but also on higher levels (i.e. acceptance testing).

Recent advances in virtualization technology make virtual infrastructure components, such as virtual machines and virtual networks, available in a “pay-as-you-go” model without long-term commitments or contracts (i.e. IaaS). Service-based on-demand infrastructure allocation and deallocation opens up the possibility to exploit resource elasticity through test automation for distributed systems.

Methodology The developed methodology of automated acceptance testing for distributed systems uses a task-oriented model for the abstraction of concurrency and remote communication in distributed systems. The model serves as a foundation for the development of a domain-specific language that is specialized on test scenarios for distributed systems. It allows fine-grained control on the parallel execution of tasks, while hiding the complexity commonly associated with parallel programming and remote, asynchronous communication.

Conclusion As argued, the developed testing method allows for authoring of test scenarios with high quality in terms of efficiency, evolvability, exemplarity, and effectiveness. Efficient testing is made possible by the high degree of automation that was achieved and by making the elasticity of on-demand virtual infrastructure available to the test author. Evolvability and maintainability are ensured by making a wide range of existing tools for source code maintenance applicable to test scenarios and their execution, by the domain-specific, semantic analysis of test scenarios that helps to avoid test defects, and by the integration of test authoring and execution into Eclipse. Tests can be exemplary because scenarios represent requirements to the system under test that are expressed by domain experts. Finally, testing can be accomplished effectively as the high degree of automation together with the flexibility of the domain-specific language facilitates scanning for defects within features that are specific to distributed systems.

Strengths and Weaknesses Apart from the ability to create high quality tests for distributed systems, it is noteworthy that not only code and design defects can be found using the suggested methodology. Additionally, test defects can be automatically detected using the domain-specific semantic analysis of test scenarios and specification defects can be uncovered because test scenarios are used to express acceptance criteria that exemplify features of the system under test. Another strength of the approach is that the integration into the continuous software delivery process is straightforward because the validation of acceptance criteria is fully automated. Finally, the implementation is clearly structured into a three layered reference architecture of acceptance testing: acceptance criteria layer, test implementation layer and application/infrastructure driver layer. The architecture reflects the main concepts of acceptance testing: features, criteria, and test scenarios.

Weaknesses are mainly founded in the prototypical implementation that suffers from some shortcomings which are a result of the limited time frame. Noteworthy is that the implementation currently only allows executing software on UNIX machines, or that the static analysis is currently not possible for advanced test scenarios that use arbitrary statements from the host language. However, these limitations are not of conceptual nature but only apply to the current state of the implementation.

Future Work Future research could be targeted towards more advanced test assertions that also use probed data (e.g. CPU and memory usage) on the individual nodes permitting the evaluation of quantitative thresholds during test execution. For instance, a test could assert that a given threshold for the average CPU usage is not exceeded. Also interesting is the scheduling of tests onto a pool of currently available virtual resources. An efficient scheduling promises optimization in respect of the efficiency of test execution, but should maintain reproducibility and repeatability of a test. Moreover, the internal Groovy domain-specific language could be eventually replaced by an external language that is based on a formal grammar. This would allow more advanced static analysis of the test scenarios and better tool support for the test author.

Glossary and Acronyms

acceptance criteria The criteria that a system or component must satisfy in order to be accepted by a user, customer, or other authorized entity [9]. 8, 10, 13, 22–24

acceptance criteria layer See Section 4.4.1. IV

acceptance testing Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system [9]. IV, 8, 10, 13, 22, 27

ACID atomicity, consistency, isolation, and durability. *see* BASE

anonymous function A function that is not bound to an identifier. *see* closure

API Application programming interface. V, 15

application driver layer See Section 4.4.3. IV

arbitrary failure A node sends arbitrary messages or responses [42]. 12, *see* failure

assertion Compares test outcome with expected outcome. *see* test assertion & comparator

AST abstract syntax tree. *see* abstract syntax tree

asynchronous task See Section 4.3.2. IV

AWS Amazon Web Services. IV, V, VII, 14

BASE Basically Available, Soft-State und Eventual Consitency. *see* ACID

BDD Behaviour-Driven Development. *see* behaviour-driven development

behaviour-driven development Software engineering practice that expects domain experts to express their requirements based on expected behaviour of the system under test. 22, 23, *see* test-driven development

BizUnit See Section 3.2.2. 21

black-box test Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions [9]. III, 7, 22, *see* test strategy

capture-replay test Recording tools can be used to capture the interaction of a (human) tester with the software under test in order to replay the captured test case automatically. 10

CD continuous delivery. *see* continuous delivery

centralized architecture Architectural style for distributed systems. Examples include client-server and n-tire architectures. 10, *see* decentralized architecture

CI continuous integration. *see* continuous integration

CLI Command-Line Interface. 8, 23, *see* GUI

client-server model Centralized architectural style for distributed systems. 10, *see* centralized architecture

closure Anonymous function that captures the state of its context (non-local variables). *see* anonymous function

Cloud 9 See Section 3.2. 20

cloud computing Cloud computing is defined by NIST [31] as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. 10, 13, 15, 20, *see* grid computing

CloudSim See Section 3.1.2. 17

code defect See Section 2.1.1. 7

comparator Compares test outcome with expected outcome. *see* assertion

continuous delivery Software development principle that allows to software to be released any time. 10, *see* continuous integration

continuous integration Continuous integration is a paradigm that emphasizes frequent integration of new developments into the main product line. 10, *see* continuous delivery

CPU central processing unit. V, 13, 16

crash failure A node halts, but it is working correctly until it halts [42]. 12, 13, 28, *see* failure

Cucumber Cucumber is a tool for executing automated acceptance tests. 21, 23, 24

D-Cloud See Section 3.1.2. 18, 24

decentralized architecture Architectural style for distributed systems. Examples include peer-to-peer architectures. 10, *see* centralized architecture

defect See Section 2.1.1. *see* code defect, design defect, specification defect & test defect

design defect See Section 2.1.1. 7

DIECAST See Section 3.1.2. 17, 18

distributed system A distributed system is a collection of independent computers that appears to its users as a single coherent system [42]. 2–4, 6, 10–13, 16–19, 21, 22, 24, 26–29

distributed testing Testing a *distributed* system through deployment to multiple machines. Mainly aspects that are specific to distributed systems are subject of a distributed test. 18–20, 22, *see* remote testing & distributed system

DisUnit See Section 3.2.2. 21

domain-specific language As opposed to a general purpose language a domain-specific language is a programming language that is designed to be applied only in a limited domain. 23, 26, *see*

DSL Domain-specific language. *see* domain-specific language

dynamic assertion Test assertions that are validated while a test case is executed. *see* post-execution assertion & test assertion

EC2 Amazon Elastic Compute Cloud. 15, 20

Eclipse Integrated development environment for multiple platforms. IV

executable specification A system is specified using tools that allow to automatically execute the specification against the system under test. 8, 22, 23, *see* behaviour-driven development

failure The inability of a system or component to perform its required functions [9]. 19, *see* response failure, crash failure, omission failure, timing failure & arbitrary failure

fault injection faults can be injected into a system in order to observe the behaviour of a system in presence of failure. *see* failure

Fit Framework for Integrated Test [33]. 22, 23, *see* FitNesse

FitNesse Automated testing tool based on the Fit framework. 22, 24, *see* Fit

general-purpose language A programming language that is not limited to a specific domain of application. *see* domain-specific language

Gherkin Business readable domain-specific language to describe software behaviour. III, 23, 24

global state The local state of all nodes in a distributed system combined with the state of the communication channels. *see* distributed system

GPL general-purpose language. *see* general-purpose language

grey-box test In between black-box and white-box testing. 7, *see* test strategy

grid computing Distributed computing system with a heterogeneous infrastructure. *see* cloud computing

GridSim See Section 3.1.2. 17

Groovy Groovy is a dynamic language for the Java platform. IV

GUI Graphical User Interface. 8, 23, *see* CLI

HPC High performance computing. *see* high performance computing

hypervisor Software to manage multiple virtual machines on a single server. 15, *see* virtual machine monitor

IaaS Infrastructure as a Service. XIII, 4, 13, 15, *see* Infrastructure as a Service

IDE Integrated development environment. IV

IEEE Institute of Electrical and Electronics Engineers. 8

Infrastructure as a Service Category of cloud computing services that provides infrastructure components as a service. 4, 13, 15

infrastructure driver layer See Section 4.4.4. IV

Java The Java programming language. 15, 19

jClouds A library for Java and Closure that aims to abstract numerous IaaS provider APIs into a common interface. V, 15

Jenkins A server for continuous integration. *see* continuous integration

JUnit Unit-testing framework for the Java language. IV, 19–21, *see* xUnit

JVM The Java Virtual Machine. IV

MongoDB implementation of a document-oriented database management system. V

NIST National Institute of Standards and Technology. 13

node A node is a virtual or physical machine that can be used for computation. Multiple nodes can be connected via a network. *see*

NUnit Unit-testing framework for Microsoft .NET. 20, 21, *see* xUnit

omission failure A node fails to handle an incoming message or omits an outgoing message (i.e. receive and send omissions) [42]. 12, *see* failure

PaaS Platform as a Service. 15, *see* Platform as a Service

pay-as-you-go charging services only based on usage without the need for contracts or sign-up fees. 13, 15, 20

PCM Palladio Component Model. 16, 17, 21

peer-to-peer Decentralized architectural style for distributed systems. 10, *see* decentralized architecture

post-execution assertion Test assertions that are validated after the execution of a test case. *see* dynamic assertion & test assertion

PREFAIL See Section 3.2.2. 22

QEMU Quick Emulator. Open-source hosted hypervisor for hardware virtualization. 18

QoS Quality of Service. 16, 17

RCE Remote Component Environment. V, 3, *see* grid computing

remote testing Testing a system on possibly multiple remote machines without treating the system as *one* distributed system. The system under test is often not a distributed system itself. 18, 19, *see* distributed testing

replica set a set or group of nodes among which data is replicated. *see* MongoDB

resource virtualization Simulation of physical resources such as CPUs or Memory. *see* virtual infrastructure

response failure The response of a node is incorrect (i.e. the response value is wrong or the node performs an incorrect state transition) [42]. 12, 28, *see* failure

RMI Remote Method Invocation. *see* RPC

RTF Running Tested Features. *see* running tested features

running tested features A metric that is represents the number of features that pass all their acceptance criteria. *see* acceptance testing

SaaS Software as a Service. 15, *see* Software as a Service

SCP Secure copy. File transfer using SSH. *see* SSH

software testing . *see* testing

specification by example Requirement to a software are expressed using examples very similar to use cases [3]. 22, 23

specification defect See Section 2.1.1. 7, 29

SSH secure shell. V

SUT system under test. *see* system under test

system under test software or system that is subject to a test. III, 2, 7–10, 13, 16–24, 27–29

task See Section 4.2.2. III–V

task group See Section 4.2.2. IV, V

TCP Transmission Control Protocol. 12

TDD Test-Driven Development. *see* test-driven development

test author The person that designs an builds tests. IV, 6, 9, 11, 13, 22, 23, 27

test case A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement [9]. 8, 9, 11–13, 27, 29, *see* test scenario

test characteristics Tests can be categorized based on the quality attributes that are under test. As suggested by [10] the following characteristics exist: portability, maintainability, efficiency, usability, reliability, functionality. III, 7, 8

test defect See Section 2.1.1. IV, 7

test first Software engineering practice where tests are created prior to the actual implementation. *see* test-driven development

test implementation layer See Section 4.4.2. IV

test level Tests can be categorized into levels based on their complexity and degree of isolation. As suggested by [10] the following levels exist: unit, module, integration, system, regression, alpha/beta, acceptance. III, 7, 8, 10, 12, 13, 19, 22, 29

test scenario See Section 4.2.1. III, IV, VII, 8, 23, 26, *see* test case

test strategy As suggested by [10] test can be categorized by their strategy: white-box, grey-box and black-box. III, 7, 8, *see* test level

test-driven development Software engineering practice where tests are created prior to the actual implementation. 13, *see* behaviour-driven development

timing failure A response of a node lies outside a specified time frame [42]. 12, *see* failure

unit test Testing of individual hardware or software units or groups of related units [9]. 7, 19, *see* test level

UNITE Non-functional Intentions via Testing and Experimentation. See Section 3.2.2. 21, 24

USD US-Dollar. 14

use case A list of steps that define an interaction between a role and a system. 2, 22

virtual infrastructure Infrastructure composed of virtual components such as virtual machines and virtual local area networks. V, 4, 13–15, 20, 24, 26, *see* VM & VLAN

virtual machine Simulation of a physical computer. 15, *see* virtual infrastructure

virtual machine monitor Software to manage multiple virtual machines on a single server. 15, *see* hypervisor

VM Virtual Machine. V, VII, 14, 15, 18, *see* virtual machine

VMM Virtual Machine Monitor. 15, *see* virtual machine monitor

white-box test A system or component whose internal contents or implementation are known [9]. 7, *see* test strategy

workload See Section 4.2.2. IV

XML Extensible Markup Language (XML) is a markup language. 18, 21

xUnit Family of unit-testing frameworks. The "x" is used as a wildcard for many framework names that have the suffix "Unit". *see* JUnit & NUnit

YETI York Extensible Testing Infrastructure. 19, 20

A. Listings

This section contains additional source code listings that are referenced from the text. Below every listing a path is provided that points to the corresponding file on the enclosed CD.

Complete Listing of Test Scenario TS1.1

The complete listing of Test Scenario TS1.1, Section 5.2.2.

Listing A.1: Complete listing of test scenario 1.1

```
// activate Groovy static type checking
@TypeChecked
// activate static checking of execution semantics
@TypeChecked(extensions='Compiler.groovy.typecheck')
protected Closure<Void> define() { return {

def rceDir = session.getWorkingDir('rce')
def workingDir =  session.getWorkingDir()
def localDir = Utils.replaceTokens("d:\\thesis\\SESSION-TOKEN\\", session)
def tmpDir = session.getTmpDir()
def wfFile = tmpDir + "/TEST.wf"

// request nodes from infrastructure provider
sync 'provision nodes', common.provisionNodes(nodeCount, getSession())

// reserver required nodes exclusively
group 'all', allocate(nodeCount)

// delete files that might still be in the session folder
sync  'clear session', common.clearSession(workingDir, group('all'))
// activate CPU and memory profiling on all nodes
async 'activate profiling', common.activateProfiling(group('all'), workingDir)
// write metadata in log file
sync  'write session info', common.writeSessionInfo(group('all'), workingDir)
// install utility software on all nodes
sync  'prepare system', common.prepareSystem(group('all'))
// download and install RCE on every node
sync  'install rce', domain.installRce(group('all'), workingDir, downloadUrl)

// create a group with one node
group 'server', group('all').any(1)
// create a group that contains proxies and clients
group 'clientOrProxy', group('all') - group('server')
// split the group using a ration of 1/3
splitGroup 'clientOrProxy', 'proxy', 'client', 3

// configure the RCE server instance
async 'configure server', {
  // write meta information to execution log
  sync common.execLog(group('server'), 'server', workingDir)
  // configure node to provide a network contact point
  sync domain.configServer(rceDir, group('server'))
  // make general RCE configurations
  sync domain.configureRceGeneral(rceDir, group('server'))
}

// configure the RCE client instances
async 'configrue client', {
  // write meta information to execution log
  sync common.execLog(group('client'), 'client', workingDir)
  // configure nodes connect to one random proxy
  sync domain.configureAsClientRandomProxy(rceDir, group('client'), group('proxy'))
  // upload workflow file to the client node
  sync common.uploadContent(group('client'),
    FileTemplates.getVerySimpleWorkflow(Utils.createRceNodeId(group('server')).any()), wfFile)
}

// configure the RCE proxy instances
async 'configrue proxy', {
  // write meta information to execution log
  sync common.execLog(group('proxy'), 'proxy', workingDir)
  // configure nodes to provide network contact point and to use server
```

```
63   sync domain.configProxy(rceDir, group('proxy'), group('server'))
64 }
65
66 // continue only after server configuration
67 await 'configure server'
68 // run RCE on the server node
69 async 'run server', domain.runRce(rceDir, 0, group('server'))
70 // block until RCE runs on server node
71 sync  'assert server runs', domain.assertRceRuns(rceDir,group('server'))
72 // continue only after proxy configuration
73 await 'configrue proxy'
74 // run RCE on the proxy nodes
75 async 'run proxy', domain.runRce(rceDir, 2000, group('proxy'))
76 // block until RCE runs on all proxy nodes
77 sync  'assert proxy runs', domain.assertRceRuns(rceDir, group('proxy'))
78 // continue only after client configuration
79 await 'configrue client'
80 // run RCE on all client executing a workflow
81 async 'run rce client', domain.runRce(rceDir, wfFile, 2000, group('client'))
82 // ensure that every workflow succeeds
83 sync  'assert wf success', domain.assertWfSuccess(rceDir, group('client'))
84 // scenario should fail if workflow execution failed
85 assertThat 'assert wf success'
86
87 sync 'tear down', {
88   // terminate profiling scripts
89   sync  common.terminateProfiling(group('all'), workingDir)
90   async 'profiling',     common.downloadProfilingResults(group('all'), workingDir, localDir)
91   async 'exec. log',     common.downloadExecLog(group('all'), workingDir, localDir)
92   async 'rce logfile',   domain.downloadRceLogFile(group('all'), rceDir, localDir)
93   async 'comm. config', domain.downloadRceCommunicationConfig(group('all'), rceDir, localDir)
94   async 'workflow',      domain.downloadWorkflowFile(group('client'), wfFile, localDir)
95
96   await([ 'profiling', 'rce logfile', 'comm. config', 'exec. log', 'workflow'])
97   postProcessingScripts(localDir)
98   print "Results: $localDir"
99 }
100 }}
```

➤ Source: CD:/02-source/impl/domain/rce/TestScenario11.groovy

Complete Listing of Test Scenario TS1.2

The complete listing of Test Scenario TS1.2, Section 5.2.3.

Listing A.2: Complete listing of test scenario 1.2

```
1  // activate Groovy static type checking
2  @TypeChecked
3  // activate static checking of execution semantics
4  @TypeChecked(extensions='Compiler.groovy.typecheck')
5  protected Closure<Void> define() { return {
6
7  def rceDir = session.getWorkingDir('rce')
8  def workingDir =  session.getWorkingDir()
9  def localDir = Utils.replaceTokens("d:\\thesis\\SESSION-TOKEN\\", session)
10 def tmpDir = session.getTmpDir()
11 def wfFile = tmpDir + "/TEST.wf"
12
13 // provision nodes from IaaS provider
14 sync 'provision nodes', common.provisionNodes(nodeCount, getSession())
15
16 // test for a precondition
17 assertThat nodeCount == 4, 'Sorry, the scenario does only make sense with exactly four nodes!'
18
19 // reserver nodes for test execution
20 group 'all', allocate(nodeCount); info('all')
21
22 // clean up the session folder in order to remove files from last test
23 sync  'clear session', common.clearSession(workingDir, group('all'))
24 // start profiling script on every node
25 async 'activate profiling', common.activateProfiling(group('all'), workingDir)
26 // write metadata to session log file
27 sync  'write session info', common.writeSessionInfo(group('all'), workingDir)
28 // install other needed software
29 sync  'prepare system', common.prepareSystem(group('all'))
30 // download RCE from Sourceforge and install it on all nodes
31 sync  'install rce', domain.installRce(group('all'), workingDir, downloadUrl)
32
33 // select arbitrary nodes as server
34 group 'server', group('all').any(1)
```

```
35 // create a group with all nodes but the server
36 group 'others', group('all') - group('server')
37 // create a group with the client
38 group 'client', group('others').any(1)
39 // create a group with the two proxies
40 group 'proxy',  group('others') - group('client')
41 // create a group with the first proxy
42 group 'first',  group('proxy').any(1)
43 // create a group with the second proxy
44 group 'second', group('proxy') - group('first')
45
46 // configure the server node
47 async 'configure server', {
48   if (failure) return // skip if scenario has failed already
49   // write some metadata to the test execution log
50   sync 'write log server', common.execLog(group('server'), 'server', workingDir)
51   // make a server contact point available
52   sync 'config topology server', domain.configServer(rceDir, group('server'))
53   // make general RCE configurations
54   sync 'config rce server', domain.configureRceGeneral(rceDir, group('server'))
55 }
56
57 // configure the client node
58 async 'configrue client', {
59   if (failure) return // skip if scenario has failed already
60   // write some metadata to the test execution log
61   sync 'write log client', common.execLog(group('client'), 'client', workingDir)
62   // configure the two proxies p1 and p2 as remote contact points
63   sync 'config topology client', domain.configureAsClientAllProxies(rceDir, group('client'),
       group('proxy'))
64   // make general RCE configurations
65   sync 'config rce client', domain.configureRceGeneral(rceDir, group('client'))
66   // upload a textual workflow description to the client
67   sync 'upload workflow client', common.uploadContent(group('client'),
68       FileTemplates.getLongRunningWorkflow(Utils.createRceNodeId(group('server')).any()), wfFile)
69 }
70
71 // configure the proxy nodes
72 async 'configrue proxy', {
73   if (failure) return // skip if scenario has failed already
74   // write some metadata to the test execution log
75   sync 'write log proxy', common.execLog(group('proxy'), 'proxy', workingDir)
76   // configure the server as remote contact point and offer a server contact point
77   sync 'config topology proxy', domain.configProxy(rceDir, group('proxy'), group('server'))
78   // make general RCE configurations
79   sync 'config rce proxy', domain.configureRceGeneral(rceDir, group('proxy'))
80 }
81
82 // run RCE on server
83 sync 'run server and first proxy', {
84   if (failure) return // skip if scenario has failed already
85   // continue only if server has been configured
86   await 'configure server'
87   // run RCE on the server node
88   async 'run server', domain.runRce(rceDir, 0, group('server'))
89   // block until RCE runs on the server
90   sync  'assert server runs', domain.assertRceRuns(rceDir,group('server'))
91   // the test should fail, if the server did not run
92   assertThat 'assert server runs'
93
94 }
95
96 sync 'run the first proxy', {
97   // skip if scenario has failed already
98   if (failure) return
99   // now that the server runs, run the proxy
100  await 'configrue proxy'
101  // run only the first proxy p1
102  async 'run first proxy', domain.runRce(rceDir, 0, group('first'))
103  // block until the first proxy runs
104  sync  'assert first proxy runs', domain.assertRceRuns(rceDir, group('first'))
105  // and also the proxy must have started
106  assertThat 'assert first proxy runs'
107 }
108
109 sync 'run the second proxy', {
110   // skip if scenario has failed already
111   if (failure) return
112   // run second proxy p2
113   async 'run second proxy', domain.runRce(rceDir, 0, group('second'))
114   // make sure second proxy is runs
115   sync  'assert second proxy runs', domain.assertRceRuns(rceDir, group('second'))
116 }
117
118 sync 'run the client', {
```

```
119   // skip if scenario has failed already
120   if (failure) return
121   // do only continue when the configuration of the client is done
122   await 'configrue client'
123   // run RCE client in the background (async)
124   async 'run client', domain.runRce(rceDir, 0, group('client'))
125   // make sure that running RCE was successful
126   sync  'assert client runs', domain.assertRceRuns(rceDir, group('client'))
127   // make sure the client runs
128   assertThat 'assert client runs'
129 }
130
131 sync 'cause fault', {
132   // skip if scenario has failed already
133   if (failure) return
134   if (shutdownInsteadOfCrash)  sync 'shut down first proxy', domain.shutdownRce(group('first'))
135   if (!shutdownInsteadOfCrash) sync 'crash first proxy', domain.killRce(group('first'))
136 }
137
138 sync 'run workflow on client', { // continue only if test has not failed already
139   if (failure) return // skip if scenario has failed already
140   // wait for 40 sec. before executing the worfklow
141   waitFor 40000
142   // trigger workflow execution on RCE client
143   async 'run wf', domain.runWorkflow(wfFile, group('client'))
144   // check in the log file if the workflow has actually started
145   sync  'assert wf started', domain.assertWfStarted(rceDir, group('client'))
146   // if the workflow did not start, set test status to FAILED
147   assertThat 'assert wf started'
148 }
149
150 if (!failure && workflowSuccessExpected) {
151  sync  'assert wf success', domain.assertWfSuccess(rceDir, group('client'))
152  assertThat 'assert wf success'
153 }
154
155 if (!failure && !workflowSuccessExpected) {
156   sync  'assert wf failure', domain.assertWfFailure(rceDir, group('client'))
157   assertThat 'assert wf failure'
158 }
159
160 sync 'tear down', {
161   // terminate profiling scripts
162   sync  common.terminateProfiling(group('all'), workingDir)
163   sync  domain.killRce(group('all'))
164
165   // download multiple files from nodes
166   async 'profiling',     common.downloadProfilingResults(group('all'), workingDir, localDir)
167   async 'exec. log',     common.downloadExecLog(group('all'), workingDir, localDir)
168   async 'rce logfile',   domain.downloadRceLogFile(group('all'), rceDir, localDir)
169   async 'comm. config', domain.downloadRceCommunicationConfig(group('all'), rceDir, localDir)
170   async 'workflow',      domain.downloadWorkflowFile(group('client'), wfFile, localDir)
171
172   await(['profiling', 'rce logfile', 'comm. config', 'exec. log', 'workflow'])
173
174   postProcessingScripts(localDir)
175   print "Results: ${localDir}"
176   print 'All done'
177 }
178 }}
```

➤ Source: CD:/02-source/impl/domain/rce/TestScenario12.groovy

Complete Listing of Test Scenario TS1.3

The complete listing of Test Scenario TS1.3, Section 5.2.3.

Listing A.3: Complete listing of test scenario 1.3

```
1  // activate Groovy static type checking
2  @TypeChecked
3  // activate static checking of execution semantics
4  @TypeChecked(extensions='Compiler.groovy.typecheck')
5  protected Closure<Void> define() { return {
6
7  def rceDir = session.getWorkingDir('rce')
8  def workingDir =  session.getWorkingDir()
9  def localDir = Utils.replaceTokens("d:\\thesis\\SESSION-TOKEN\\", session)
10 def tmpDir = session.getTmpDir()
11 def wfFile = tmpDir + "/TEST.wf"
12
```

```
13 // provision nodes from IaaS provider
14 sync 'provision nodes', common.provisionNodes(nodeCount, getSession())
15
16 // test for a precondition
17 assertThat nodeCount == 4, 'Sorry, the scenario does only make sense with exactly four nodes!'
18
19 // reserver nodes for test execution
20 group 'all', allocate(nodeCount); info('all')
21
22 // clean up the session folder in order to remove files from last test
23 sync  'clear session', common.clearSession(workingDir, group('all'))
24 // start profiling script on every node
25 async 'activate profiling', common.activateProfiling(group('all'), workingDir)
26 // write metadata to session log file
27 sync  'write session info', common.writeSessionInfo(group('all'), workingDir)
28 // install other needed software
29 sync  'prepare system', common.prepareSystem(group('all'))
30 // download RCE from Sourceforge and install it on all nodes
31 sync  'install rce', domain.installRce(group('all'), workingDir, downloadUrl)
32
33 // select arbitrary nodes as server
34 group 'server', group('all').any(1)
35 // create a group with all nodes but the server
36 group 'others', group('all') - group('server')
37 // create a group with the client
38 group 'client', group('others').any(1)
39 // create a group with the two proxies
40 group 'proxy',  group('others') - group('client')
41 // create a group with the first proxy
42 group 'first',  group('proxy').any(1)
43 // create a group with the second proxy
44 group 'second', group('proxy') - group('first')
45
46 async 'configure server', {
47   // skip if scenario has failed already
48   if (failure) return
49   // write some metadata to the test execution log
50   sync 'write log server', common.execLog(group('server'), 'server', workingDir)
51   // make a server contact point available
52   sync 'config topology server', domain.configServer(rceDir, group('server'))
53   // make general RCE configurations
54   sync 'config rce server', domain.configureRceGeneral(rceDir, group('server'))
55 }
56
57 async 'configrue client', {
58   // skip if scenario has failed already
59   if (failure) return
60   // write some metadata to the test execution log
61   sync 'write log client', common.execLog(group('client'), 'client', workingDir)
62   // configure the two proxies p1 and p2 as remote contact points
63   sync 'config topology client', domain.configureAsClientAllProxies(rceDir, group('client'),
         group('proxy'))
64   // make general RCE configurations
65   sync 'config rce client', domain.configureRceGeneral(rceDir, group('client'))
66   // upload a textual workflow description to the client
67   sync 'upload workflow client', common.uploadContent(group('client'),
68     FileTemplates.getLongRunningWorkflow(Utils.createRceNodeId(group('server')).any()), wfFile)
69 }
70
71 async 'configrue proxy', {
72   if (failure) return // skip if scenario has failed already
73   // write some metadata to the test execution log
74   sync 'write log proxy', common.execLog(group('proxy'), 'proxy', workingDir)
75   // configure the server as remote contact point and offer a server contact point
76   sync 'config topology proxy', domain.configProxy(rceDir, group('proxy'), group('server'))
77   // make general RCE configurations
78   sync 'config rce proxy', domain.configureRceGeneral(rceDir, group('proxy'))
79 }
80
81 sync 'run the server', {
82   if (failure) return // skip if scenario has failed already
83   // continue only if server has been configured
84   await 'configure server'
85   // run RCE on the server node
86   async 'run server', domain.runRce(rceDir, 0, group('server'))
87   // block until RCE runs on the server
88   sync  'assert server runs', domain.assertRceRuns(rceDir,group('server'))
89   // the test should fail, if the server did not run
90   assertThat 'assert server runs'
91 }
92
93 // run first proxy p1
94 sync 'run the first proxy', {
95   if (failure) return // skip if scenario has failed already
96   // now that the server runs, run the proxy
```

```
97    await 'configrue proxy'
98    // run only the first proxy p1
99    async 'run first proxy', domain.runRce(rceDir, 0, group('first'))
100   // block until the first proxy runs
101   sync  'assert first proxy runs', domain.assertRceRuns(rceDir, group('first'))
102   // and also the proxy must have started
103   assertThat 'assert first proxy runs'
104 }
105
106 sync 'run the client', {
107   if (failure) return // skip if scenario has failed already
108  // do only continue when the configuration of the client is done
109  await 'configrue client'
110  // run RCE client in the background (async)
111  async 'run client', domain.runRce(rceDir, 0, group('client'))
112  // make sure that running RCE was successful
113  sync  'assert client runs', domain.assertRceRuns(rceDir, group('client'))
114  // make sure the client runs
115  assertThat 'assert client runs'
116 }
117
118 sync 'run workflow', {
119   if (failure) return // skip if scenario has failed already
120   // wait for 10 sec. before executing the worfklow
121   waitFor 10000
122   // trigger workflow execution on RCE client
123   async 'run wf', domain.runWorkflow(wfFile, group('client'))
124   // check in the log file if the workflow has actually started
125   sync  'assert wf started', domain.assertWfStarted(rceDir, group('client'))
126   // if the workflow did not start, set test status to FAILED
127   assertThat 'assert wf started'
128 }
129
130 sync 'run the second proxy', {
131   // skip if scenario has failed already
132   if (failure) return
133   // run second proxy p2
134   async 'run second proxy', domain.runRce(rceDir, 0, group('second'))
135   // make sure second proxy is runs
136   sync  'assert second proxy runs', domain.assertRceRuns(rceDir, group('second'))
137 }
138
139 sync 'cause fault', {
140   // skip if scenario has failed already
141   if (failure) return
142   // wait for 10 sec. before terminating first client
143   waitFor 10000
144   // assert that the client is still executing the workflow
145   sync  'wf still running', domain.assertWfStillRunning(rceDir, group('client'))
146   // fail if workflow to quickly finished
147   assertThat 'wf still running'
148   if (shutdownInsteadOfCrash)  sync 'shut down first proxy', domain.shutdownRce(group('first'))
149   if (!shutdownInsteadOfCrash) sync 'crash first proxy', domain.killRce(group('first'))
150 }
151
152 if (!failure && workflowSuccessExpected) {
153   sync  'assert wf success', domain.assertWfSuccess(rceDir, group('client'))
154   assertThat 'assert wf success'
155 }
156
157 if (!failure && !workflowSuccessExpected) {
158   sync  'assert wf failure', domain.assertWfFailure(rceDir, group('client'))
159   assertThat 'assert wf failure'
160 }
161
162
163 sync 'tear down', {
164   // terminate profiling scripts
165   sync  common.terminateProfiling(group('all'), workingDir)
166   sync  domain.killRce(group('all'))
167
168   // download multiple files from nodes
169   async 'profiling',     common.downloadProfilingResults(group('all'), workingDir, localDir)
170   async 'exec. log',     common.downloadExecLog(group('all'), workingDir, localDir)
171   async 'rce logfile',   domain.downloadRceLogFile(group('all'), rceDir, localDir)
172   async 'comm. config', domain.downloadRceCommunicationConfig(group('all'), rceDir, localDir)
173   async 'workflow',      domain.downloadWorkflowFile(group('client'), wfFile, localDir)
174
175   await(['profiling', 'rce logfile', 'comm. config', 'exec. log', 'workflow'])
176
177   postProcessingScripts(localDir)
178   print "Results: ${localDir}"
179   print 'All done'
180 }
181 }}
```

➤ Source: CD:/02-source/impl/domain/rce/TestScenario13.groovy

Complete Listing of Test Scenario TS2.1

The complete listing of Test Scenario TS2.1, Section 5.3.2.

Listing A.4: Complete listing of test scenario 2.1

```
// activate Groovy static type checking
@TypeChecked
// activate static checking of execution semantics
@TypeChecked(extensions='Compiler.groovy.typecheck')
protected Closure<Void> define() { return {

def workingDir = session.getWorkingDir()
def localDir = Utils.replaceTokens('d:\\thesis\\SESSION-TOKEN\\', session)
def fixture = '[fixture:timestamp:' + System.nanoTime() + ']'

// provision nodes from provider
sync 'provisoning', common.provisionNodes(nodeCount, session)

print elapsed('provisoning')

// create group with all nodes
group 'all', allocate(nodeCount)

// clean up test session folder
sync  common.clearSession(workingDir, group('all'))
// execute profiling scripts on nodes to collect metrics on resource consumption
async common.activateProfiling(group('all'), workingDir)
// write metadata into log file on nodes
async common.writeSessionInfo(group('all'), workingDir)

sync 'set up mongodb', {
  // install utility software on nodes
  sync  common.prepareSystem(group('all'))
  // install MongoDB on nodes
  sync  domain.installMongoDb(group('all'))
  // kill all services, if any
  sync  domain.killService(group('all'))
  // start 'mongod' service on all nodes
  async 'mongod', domain.startService(group('all'))
  // assert that the MongoDB service is running as demon
  sync  domain.assertServiceRunning('MongoDB daemon should be running and listening', group('all'
      ))
}

sync 'create replica sets', {
  // select arbitrary node as primary
  group 'init-primary', group('all').any(1)
  // all other nodes are secondaries
  group 'init-secondary', group('all') - group('init-primary')
  // initiate the MongoDB replica set on the primary node
  sync  domain.rsInitiate(group('init-primary'))
  // assert that the replica set has been initialized
  sync  'repl set init', domain.assertInited('Replica set should be inited on server', group('
      init-primary'))
  // add all secondary nodes to the replica set
  sync  domain.addToReplicaSet(group('init-primary'), group('init-secondary'))
  // assert that the replica set size has the expected size on all nodes
  sync  'repl set size', domain.assertReplicaSetSize(group('all').size(), group('all'))
  // assert that all nodes are in state PRIMARY or SECONDARY
  sync  domain.assertPrimaryOrSecondary(group('all'))

  assertThat 'repl set init'
  assertThat 'repl set size'
}

sync 'cause failue', {
  if (failure) return; // skip if scenario has failed already

  // create group with arbitrary nodes as failing nodes
  group 'failing', group('all').any(failingCount)
  // create group that contains healthy nodes
  group 'healthy', group('all') - group('failing')
  // kill all services on nodes in group 'failing'
  sync  domain.killService(group('failing'))

  if (toleranceExpected) { // tolerance expected
    // make sure a primary was elected
```

```
71      sync  'assert primary election', domain.assertOnePrimary(group('healthy'))
72      // scenario should fail if no primary can be found
73      assertThat 'assert primary election'
74    }
75
76    if (!toleranceExpected) { // tolerance not expected
77      // make sure no primary was elected
78      sync  'assert no primary election', domain.assertNoPrimary(group('healthy'))
79      // scenario should fail if a primary was elected
80      assertThat 'assert no primary election'
81    }
82  }
83
84
85  sync 'issue db write', {
86    if (failure || !toleranceExpected) return; // skip if scenario has failed already
87
88    // get primary node for write input
89    async 'find primary', domain.findPrimary(group('all'))
90    // get node id that belongs to primary instance
91    group 'primary', extractGroup(taskGroup(await('find primary')))
92    // drop database, if one exists
93    sync  domain.dropDatabase('test', group('primary'))
94    // write unique content to database
95    sync  domain.writeToDatabase('test', '{a:\'' + fixture + '\'}', group('primary'))
96    // start up the crashed nodes
97    async domain.startService(group('failing'))
98    // assert that all nodes are in state PRIMARY or SECONDARY
99    sync  domain.assertPrimaryOrSecondary(group('all'))
100   // find record in database and assert write success on all nodes
101   sync  'write success', domain.assertJsOutputContainsSubstring(
102     'rs.slaveOk(); printjson(db.test.find()[0])',
103     fixture, group('all'))
104   // The test should fail if the fixture was not written to all nodes
105   assertThat 'write success'
106 }
107
108 sync  domain.killService(group('all'))
109 await 'mongod'
110
111 sync 'clean up', {
112   // terminate profiling scripts
113   sync  common.terminateProfiling(group('all'), workingDir)
114   // download MongoDB log files from all nodes
115   async 'log', domain.downloadLogFile(group('all'), localDir)
116   // download profiling data from all nodes
117   async 'profiling', common.downloadProfilingResults(group('all'), workingDir, localDir)
118   // download test execution log files from all nodes
119   async 'exec log', common.downloadExecLog(group('all'), workingDir, localDir)
120
121   await(['log', 'profiling', 'exec log'])
122
123   postProcessingScripts(localDir)
124   print "Results: ${localDir}"
125 }
126 }}
```

➤ Source: CD:/02-source/impl/domain/mongodb/TestScenario21.groovy

Complete Listing of Test Scenario TS2.2

The complete listing of Test Scenario TS2.2, Section 5.3.3.

Listing A.5: Complete listing of test scenario 2.2

```
1  // activate Groovy static type checking
2  @TypeChecked
3  // activate static checking of execution semantics
4  @TypeChecked(extensions='Compiler.groovy.typecheck')
5  protected Closure<Void> define() { return {
6
7
8  def workingDir = session.getWorkingDir()
9  def localDir = Utils.replaceTokens('d:\\thesis\\SESSION-TOKEN\\', session)
10 def fixture = '{a:\'' + StringUtils.repeat("X", bytesPerWriteOperation) + '\'}';
11 def time1, time2;
12
13 // provision nodes from provider
14 sync 'provisoning', common.provisionNodes(replicaSetSize, session)
15
16 print elapsed('provisoning')
```

```
17
18 // create group with all nodes
19 group 'all', allocate(replicaSetSize)
20 info('all')
21
22 // clean up test session folder
23 sync  common.clearSession(workingDir, group('all'))
24 // execute profiling scripts on nodes to collect metrics on resource consumption
25 async common.activateProfiling(group('all'), workingDir)
26 // write metadata into log file on nodes
27 async common.writeSessionInfo(group('all'), workingDir)
28
29 sync 'set up mongodb', {
30   // install utility software on nodes
31   sync  common.prepareSystem(group('all'))
32   // install MongoDB on nodes
33   sync  domain.installMongoDb(group('all'))
34   // kill all services, if any
35   sync  domain.killService(group('all'))
36   // start 'mongod' service on all nodes
37   async 'mongod', domain.startService(group('all'))
38   // assert that the MongoDB service is running as demon
39   sync  'assert mongod', domain.assertServiceRunning('MongoDB listening', group('all'))
40
41   assertThat 'assert mongod'
42 }
43
44 sync 'create replica sets', {
45   if (failure) return;
46
47   // select arbitrary node as primary
48   group 'prim', group('all').any(1)
49   // all other nodes are secondaries
50   group 'sec', group('all') - group('prim')
51   // initiate the MongoDB replica set on the primary node
52   sync  domain.rsInitiate(group('prim'))
53   // assert that the replica set has been initialized
54   sync  'repl set init', domain.assertInited('Replica set inited', group('prim'))
55   // add all secondary nodes to the replica set
56   sync  domain.addToReplicaSet(group('prim'), group('sec'))
57   // assert that the replica set size has the expected size on all nodes
58   sync  'repl set size', domain.assertReplicaSetSize(group('all').size(), group('all'))
59   // assert that all nodes are in state PRIMARY or SECONDARY
60   sync  domain.assertPrimaryOrSecondary(group('all'))
61
62   assertThat 'repl set init'
63   assertThat 'repl set size'
64 }
65
66 // phase 1, first write with no failure
67 sync 'write measurement without failure', {
68   if (failure) return; // do not continue in case of test failure
69   // drop database, if one exists
70   sync  domain.dropDatabase('tmp', group('prim'))
71   // init a profiler for measurement
72   profiling 'measurement 1', 'write without failure'
73   // start writing stream synchronously
74   sync 'stream 1', domain.writeToDbOften('tmp', writeOps, fixture, group('prim'))
75   // assert that the number of records matches the expected value
76   sync 'assert size 1', domain.assertDbSize(writeOps, 'tmp', group('all'))
77   // fail if writes were not successful
78   assertThat 'assert size 1'
79   // store profiling results
80   time1 = profiling 'measurement 1'
81 }
82
83 // phase 2, second write with failure
84 sync 'write measurement with failure', {
85   if (failure) return; // do not continue in case of test failure
86   // drop database, if one exists
87   sync  domain.dropDatabase('tmp', group('prim'))
88   // init a profiler for measurement
89   profiling 'measurement 2', 'write with failure'
90   // start writing stream asynchronously
91   async 'stream 2', domain.writeToDbOften('tmp', writeOps, fixture, group('prim'))
92   // wait until 1/3 of the data volume is written
93   sync 'size 1/3', domain.assertDbSizeLargerThan(round(writeOps/3), 'tmp', group('all'))
94   // create group with arbitrary nodes as failing nodes
95   group 'failing', group('sec').any(failingCount)
96   // terminate MongoDB processes
97   sync  domain.killService(group('failing'))
98   // wait until 2/3 of the data volume is written
99   sync 'size 2/3', domain.assertDbSizeLargerThan(round(writeOps/3*2), 'tmp', group('prim'))
100  // restart MongoDB services
101  async domain.startService(group('failing'))
```

```
102   // synchronize writing stream 2
103   await 'stream 2'
104   // assert that every member of the replica set has all records
105   sync 'assert final size', domain.assertDbSize(writeOps, 'tmp', group('all'))
106   // fail scenario if not all members have all records
107   assertThat 'assert final size'
108   // store profiling results
109   time2 = profiling 'measurement 2'
110
111   // calculate absolute time difference
112   def delayAbsolute = time2 - time1
113   // calculate relative time difference
114   def delayRelative = (delayAbsolute/time1) * 100
115   // print results in log
116   print "Failure scenario took ${delayRelative}% (${delayAbsolute} sec.) longer."
117   // fail if relative difference is too large
118   assertThat delayRelative < toleratedDelay
119 }
120
121 sync 'clean up', {
122   // terminate profiling scripts
123   sync  common.terminateProfiling(group('all'), workingDir)
124   // download MongoDB log files from all nodes
125   async 'log', domain.downloadLogFile(group('all'), localDir)
126   // download profiling data from all nodes
127   async 'profiling', common.downloadProfilingResults(group('all'), workingDir, localDir)
128   // download test execution log files from all nodes
129   async 'exec log', common.downloadExecLog(group('all'), workingDir, localDir)
130   // await all downloading tasks
131   await(['log', 'profiling', 'exec log'])
132   // create R scripts for further processing
133   postProcessingScripts(localDir)
134   // print directory that contains the results of the test
135   print "Results: ${localDir}"
136 }
137 }}
```

➤ Source: CD:/02-source/impl/domain/mongodb/TestScenario22.groovy

Listing of Algorithm: Dependency Graph

See Section 4.3.3 for more details on how the dependency graph is created.

Listing A.6: Java implementation of the recursive algorithm that translates an AST into a dependency graph

```
1   /**
2    * Recursively traverse the AST to derive the dependency graph.
3    *
4    * @param parent Currently analyzed statement.
5    * @param depGraph The dependency graph that is created.
6    * @param depth Current AST depth.
7    */
8   private void constructDependencyGraph(final StatementNode parent, final DependencyGraph
         depGraph, final int depth) {
9
10    StatementNode prevStm = null;
11
12    for (Statement statement : ((BlockStatement) parent.getClosure().getCode()).getStatements())
           {
13      // complexity: O(n)
14      Expression expression = getExpressionStatement(statement);
15      if (expressionInWhitelist(expression)) {
16
17        System.out.println(String.format("Processing: line=%s, statement=%s", statement.
               getLineNumber(), statement.getText()));
18
19        final StatementNode stm = new StatementNode((MethodCallExpression) expression, prevStm,
               parent, depth);
20        depGraph.addStatement(stm);
21        parent.setLastChildNode(stm);
22
23        // if the current statement is the first statement than
24        // add a dependency to the parent statement
25        if (!stm.hasPrevStm()) {
26          depGraph.addDependency(parent, stm);
27        }
28
29        // add dependencies to previous statement on the same level
30        // if it is not a sync statement.
31        if (stm.hasPrevStm() && !stm.getPrevStm().isSync()) {
32          depGraph.addDependency(stm.getPrevStm(), stm);
```

```
33          }
34
35          // recursion
36          if (stm.containsClosure()) {
37            constructDependencyGraph(stm, depGraph, depth + 1);
38          }
39
40          // add dependencies to last statement within the closure of the previous
41          // sibling statement
42          // or to the last sibling statement itself if it has no child statements
43          if (stm.hasPrevStm() && stm.getPrevStm().isSync()) {
44            depGraph.addDependency(stm.getPrevStm().lastChildStmOrSelf(), stm);
45          }
46
47          prevStm = stm;
48        }
49      }
50
51      // collect "await" dependencies
52      List<Pair<StatementNode>> awaitDependencies = new ArrayList<Pair<StatementNode>>();
53      for (StatementNode awaitNode : depGraph.getVertices()) {
54        if (awaitNode.isAwait()) {
55          for (StatementNode asyncNode : depGraph.getVertices()) {
56            // complexity: O(n^2)
57            if (asyncNode.isAsync() && !awaitNode.equals(asyncNode) &&
58                !asyncNode.isAnomymous()) {
59              List<String> labels = awaitNode.getLabelsFromAwaitStatement();
60              for (String label : labels) {
61                // complexity: O(n^2) (average), O(n^3) (worst)
62                if (label.equals(asyncNode.getLabel2())) {
63                  awaitDependencies.add(new Pair<StatementNode>(asyncNode.lastChildStmOrSelf(),
                        awaitNode));
64                }
65              }
66            }
67          }
68        }
69      }
70
71      // add collected "await" dependencies
72      for (Pair<StatementNode> pair : awaitDependencies) {
73        depGraph.addAwaitDependency(pair.getFirst(), pair.getSecond());
74      }
75
76      // connect "END" vertext
77      if (depth == 0) {
78        StatementNode endVertext = new StatementNode("END", prevStm.lastChildStmOrSelf().
              getMethodCallExpression());
79        depGraph.addStatement(endVertext);
80        depGraph.addDependency(prevStm.lastChildStmOrSelf(), endVertext);
81      }
82
83    }
```

➤ Source: CD:/02-source/impl/src/main/java/de/dlr/sc/utils/testing/distributed/platform/core/➥framework/ScenarioAnalyser.java

B. Enclosed CD

The CD contains an HTML file in the root directory. It can be used to browse through the contents of the CD:

➤ Source: CD:/index.html

Thesis

The CD contains a PDF version of this thesis and the exposé:

➤ Source: CD:/01-thesis

Source Code

The source code of the implementation is located in:

➤ Source: CD:/02-source/impl/main/java/de/dlr/sc/utils/testing/distributed/platform

All test scenarios presented are located in:

➤ Source: CD:/02-source/impl/domain/

The source code can be imported into Eclipse using the following resource archive file. It was generated using “Eclipse for RCP and RAP Developers, Kepler (4.3) RC3”.

➤ Source: CD:/02-source/eclipse-import-archive.zip

Test Execution Data

The result of every test scenario execution is a folder that contains artefacts about the test execution. The folders are named by a token which is also used in the text as reference (e.g. “13-05-12-13-07-01”). The test execution data contains log files, profiling, and probing data per node and analysis graphs as well as a copy of the test script that allows to reproduce the test. Data presented in the text is located in:

➤ Source: CD:/03-data/01-test-execution

An example of a folder with test execution data for the token “13-05-12-13-07-01” would be:

➤ Source: CD:/03-data/01-test-execution/13-05-12-13-07-01

C. Additional Figures

Figure C.1.: Complete dependency graph representing the execution semantics of Test Scenario TS1.2. (Vertices represent statements of the test scenario and edges represent execution dependencies of the test scenario. Rectangular vertices visualize `async` and `sync` statements (i.e. task submission) and round vertices visualize all other statements.)

Figure C.2.: Complete dependency graph representing the execution semantics of Test Scenario TS1.3. (Vertices represent statements of the test scenario and edges represent execution dependencies of the test scenario. Rectangular vertices visualize `async` and `sync` statements (i.e. task submission) and round vertices visualize all other statements.)

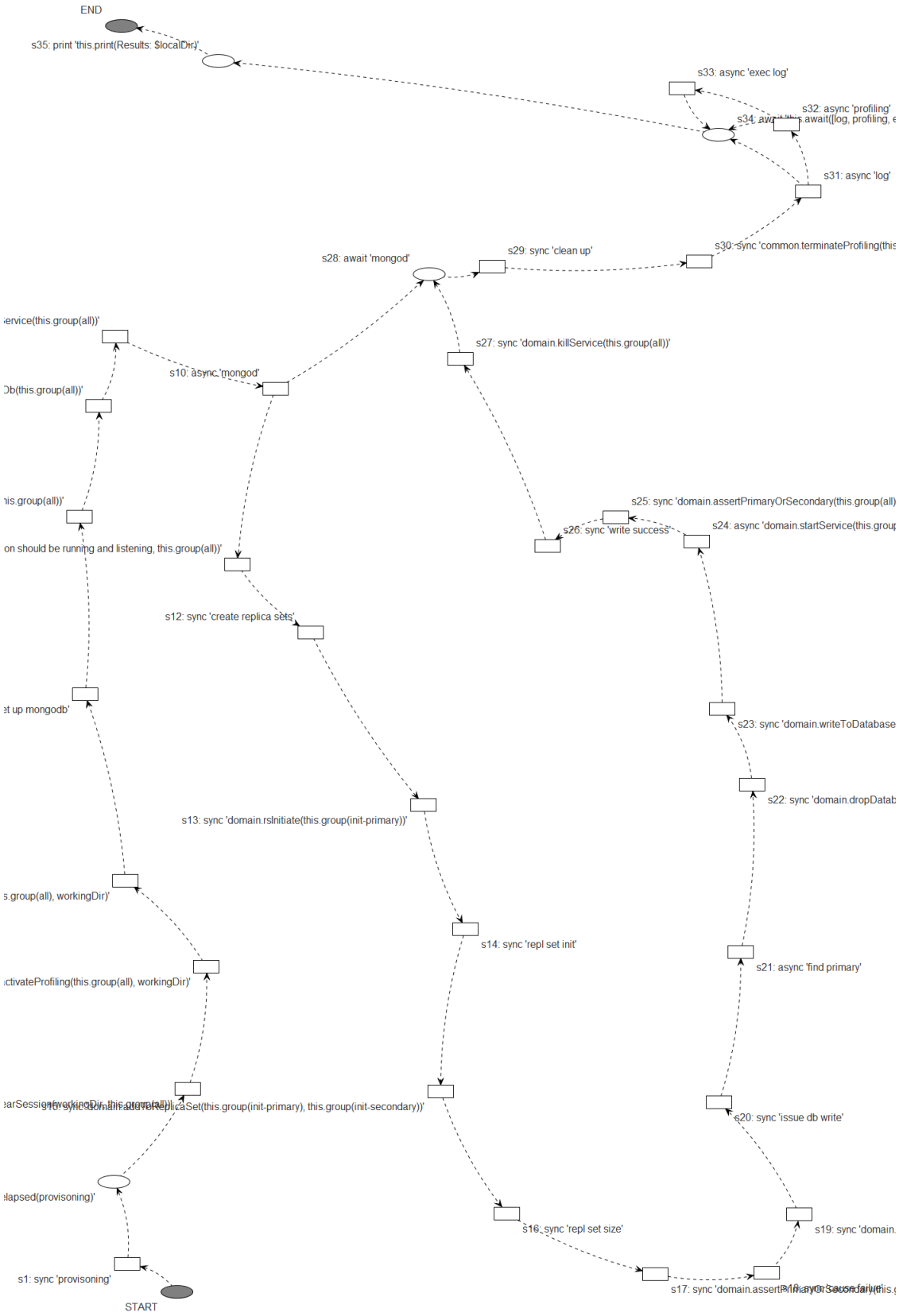


Figure C.3.: Complete dependency graph representing the execution semantics of Test Scenario TS2.1. (Vertices represent statements of the test scenario and edges represent execution dependencies of the test scenario. Rectangular vertices visualize async and sync statements (i.e. task submission) and round vertices visualize all other statements.)

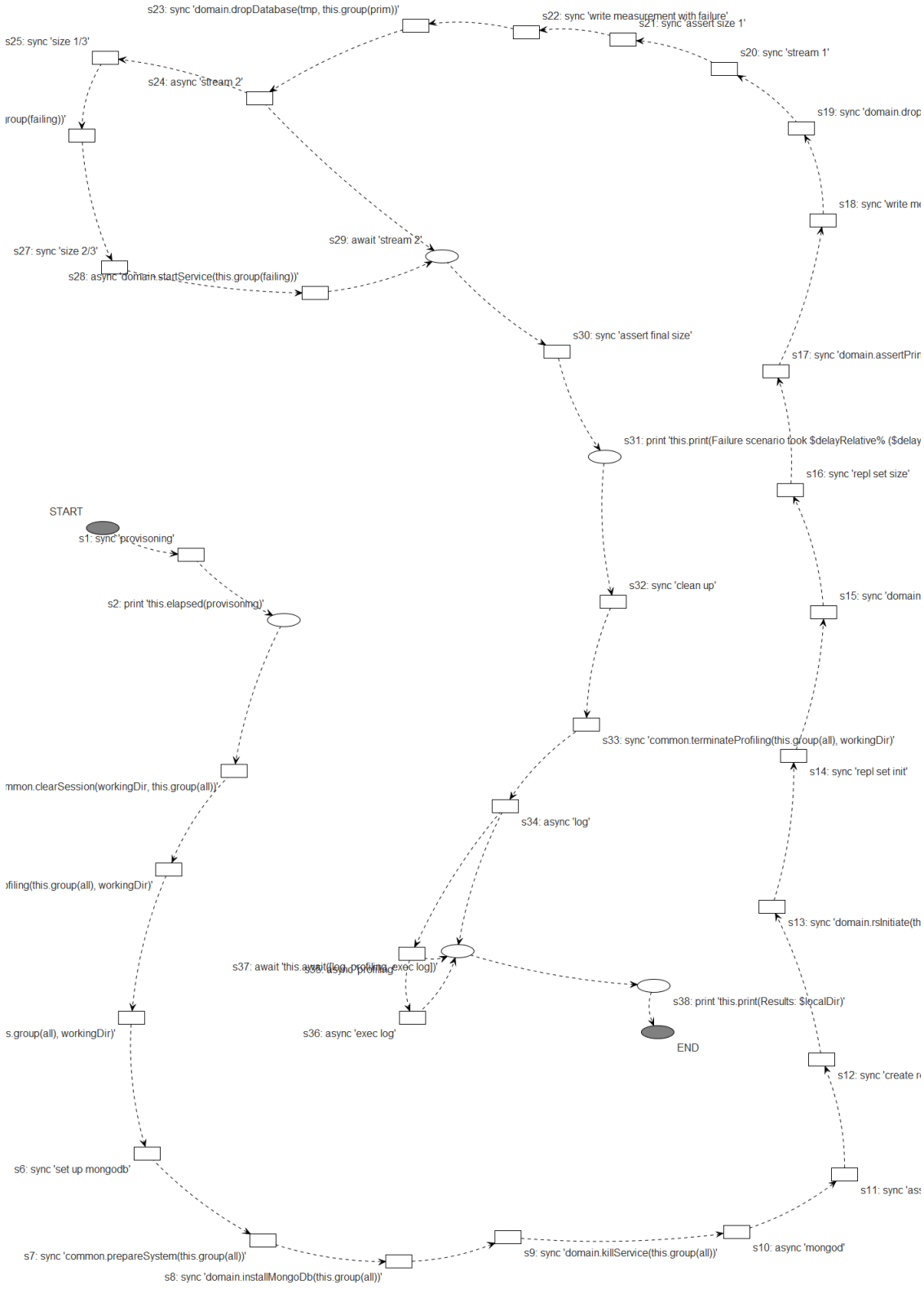


Figure C.4.: Complete dependency graph representing the execution semantics of Test Scenario TS2.2. (Vertices represent statements of the test scenario and edges represent execution dependencies of the test scenario. Rectangular vertices visualize async and sync statements (i.e. task submission) and round vertices visualize all other statements.)

D. Bibliography

[1] Amazon web services, 2012. Available online at `http://aws.amazon.com` visited on 15/11/2012.

[2] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

[3] Gojko Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning, June 2011.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, 2009.

[5] Alessio Barducci, Leo Bärring, Victor Garcia Paje, Petter Hansson, and Hlödver Tómasson. Defining unit testing of distributed systems.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, page 164–177, New York, NY, USA, 2003. ACM.

[7] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th international workshop on Software and performance*, WOSP '07, page 54–65, New York, NY, USA, 2007. ACM.

[8] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, January 2009.

[9] IEEE Standards Board. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*.

[10] Ilene Burnstein. *Practical Software Testing: A Process-Oriented Approach*. Springer, 2003 edition, July 2003.

[11] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, page 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[12] Rajkumar Buyya and Manzur Murshed. GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE (CCPE*, 14(13):1175–1220, 2002.

[13] Rodrigo N. Calheiros, Rajiv Ranjan, Cesar A. F. De Rose, and Rajkumar Buyya. CloudSim: a novel framework for modeling and simulation of cloud computing infrastructures and services. *arXiv:0903.2525*, March 2009.

[14] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. *SIGOPS Oper. Syst. Rev.*, 43(4):5–10, January 2010.

[15] Peter Deutsch. The eight fallacies of distributed computing, 1992. Available online at `https://blogs.oracle.com/jag/resource/Fallacies.html` visited on 30/03/2013.

[16] Markus Litz Andreas Schreiber Andreas Gerndt Doreen Seider, Philipp M. Fischer. Open source software framework for applications in aeronautics and space. IEEE, 2012.

[17] Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, and Patricia Machado. GridUnit: software testing on the grid. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, page 779–782, New York, NY, USA, 2006. ACM.

[18] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Prentice Hall, 1999.

[19] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 1 edition, July 2007.

[20] Mark Fewster and Dorothy Graham. *Software Test Automation*. Addison-Wesley Professional, September 1999.

[21] Martin Fowler and Rebecca Parsons. *Domain Specific Languages*. Addison-Wesley Longman, Amsterdam, 1 edition, September 2010.

[22] S.D. Gribble. Robustness in complex systems. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, 2001*, pages 21 – 26, May 2001.

[23] Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex Snoeren, and Geoffrey M. Voelker. DieCast: testing distributed systems with an accurate scale model. *ACM Trans. Comput. Syst.*, 29(2):4:1–4:48, May 2011.

[24] Irfan Habib. Virtualization with KVM. *Linux J.*, 2008(166), February 2008.

[25] James H. Hill, H.A. Turner, J.R. Edmondson, and D.C. Schmidt. Unit testing non-functional concerns of component-based distributed systems. In *International Conference on Software Testing Verification and Validation, 2009. ICST '09*, pages 406–415, April.

[26] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Longman, Amsterdam, 1 edition, July 2010.

[27] The jClouds Team. jclouds: Multi-cloud library, 2012. Available online at `http://www.jclouds.org/` visited on 12/11/2012.

[28] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: a programmable tool for multiple-failure injection. *SIGPLAN Not.*, 46(10):171–188, October 2011.

[29] Michael Le, Israel Hsu, and Yuval Tamir. Resilient virtual clusters. 2010.

[30] L Leong, D Toombs, B Gill, G Petri, and T Haynes. Magic quadrant for public cloud infrastructure as a service. *Gartner Analasis, December*, 2012.

[31] Peter Mell and Timothy Grance. The NIST definition of cloud computing (Draft). *National Institute of Standards and Technology*, page 7, January 2010.

[32] Jeffrey C. Mogul. Emergent (mis)behavior vs. complex software systems. In *IN EUROSYS*, page 293–304. ACM, 2006.

[33] Rick Mugridge and Ward Cunningham. *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall International, August 2005.

[34] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. Verlag John Wiley & Sons, Inc, February 1979.

[35] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009. CCGRID '09*, pages 124 –131, May 2009.

[36] M. Oriol and F. Ullah. YETI on the cloud. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 434 –437, April 2010.

[37] T. Parveen, S. Tilley, N. Daley, and P. Morales. Towards a distributed execution framework for JUnit test cases. In *IEEE International Conference on Software Maintenance, 2009. ICSM 2009*, pages 425 –428, September 2009.

[38] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, January 1987.

[39] C. Pham, D. Chen, Z. Kalbarczyk, and R.K. Iyer. CloudVal: a framework for validation of virtualization environment in cloud infrastructure. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 189 –196, June 2011.

[40] Runtao Qu, Satoshi Hirano, Takeshi Ohkawa, Takaya Kubota, and Radu Nicolescu. *Distributed Unit Testing*. CITR, 2006.

[41] Hitoshi Koizumi Takayuki Banzai. D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology. pages 631–636, 2010.

[42] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems. Principles and Paradigms*. Prentice Hall International, 2nd rev. ed. edition, October 2006.

[43] Amazon AWS Team. Extend your it infrastructure with amazon virtual private cloud. Technical report, Amazon.com, Inc., 2010.

[44] The CloudStack Team. Cloudstack: A complete software suite fo creating infrastructure as a service (iaas) clouds, 2012. Available online at `http://cloudstack.org/` visited on 03/12/2012.

[45] The GoGrid Team. Gogrid: Complex infrastructure made easy, 2012. Available online at `http://www.gogrid.com/` visited on 03/12/2012.

[46] The OpenNebula Team. Opennebula: Enterprise-class cloud datacenter management., 2012. Available online at `http://opennebula.org/` visited on 03/12/2012.

[47] The Rackspace Team. Rackspace: The open cloud company., 2012. Available online at `http://www.rackspace.com/` visited on 03/12/2012.

[48] J. Turek and D. Shasha. The many faces of consensus in distributed systems. *Computer*, 25(6):8–17, 1992.

[49] Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, January 2013.

[50] Matt Wynne and Aslak Hellesoy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Programmers, March 2012.

Erklärung

Ich versichere an Eides statt, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Phillip Kroll