Habilitation à diriger des recherches
Université de Nantes

Présentée par
Gerson Sunyé
gerson.sunye@univ-nantes.fr

préparée à l'unité de recherche UMR CNRS 6241 Lina
Laboratoire d'Informatique de Nantes Atlantique

A Model-Based Approach for Testing Large Scale Systems

Habilitation soutenue à Nantes le 10 novembre 2015

devant le jury composé de:

Yves Ledru
Professeur à l'Université Joseph Fourier, rapporteur

Alexander Pretschner
Professeur à TU Müchen, rapporteur

François Taiani
Professeur à l'Université de Rennes 1, rapporteur

Claude Jard
Professeur à l'Université de Nantes, examinateur

Jean-Marc Jézéquel
Professeur à l'Université de Rennes 1, examinateur

Patrick Valduriez
Directeur de Recherche, INRIA Sophia Antipolis Méditerranée, examinateur

Abstract

This document summarize the author's experience over six years testing large-scale systems. We outline that experience in four points.

First, we present a methodology for testing large-scale system. The methodology takes into account three dimensions of these systems: functionality, scalability, and volatility. The methodology proposes to execute tests in different workloads, from a small-scale static system up to a large-scale dynamic system. Experiments show that the alteration of the three dimensional aspects improves code coverage, thus improving the confidence on tests.

Second, we introduce a distributed test architecture that uses both, a broadcast protocol to send messages from the test controller to testers and a converge cast protocol to send messages from testers back to the test controller. Experiments show that the architecture is more scalable than traditional centralized architectures when testing systems with more than 1000 nodes.

Third, we present an approach for using models as dynamic oracles for testing global properties of large-scale systems. This approach focuses on global, liveness, observable and controllable properties. We propose to efficiently keep updating a global model of the system during its execution. This model is then instantiated and evolved at runtime, by monitoring the corresponding distributed system, and serve as oracle for the distributed tests. We illustrate this approach by testing the reliability of two routing algorithms under churn. Results show common flaws in both algorithms.

Finally, we present a model-driven approach for software artifacts deployment. We consider software artifacts as a product line and use feature models to represent their configurations and model-based techniques to handle automatic artifact deployment and reconfiguration. Experiments show that this approach reduces network traffic when deploying software on cloud environment.

Contents

Chapter 1

Introduction

1.1 Introduction

Large-scale systems are becoming commonplace with the increasing popularity of peer-to-peer (P2P) or cloud computing. For instance, the Gnutella [2] P2P system shares petabytes of data among millions of users. Data intensive applications, based on Google's MapReduce [42], process several terabytes of data every day, on large clusters of commodity machines, in a way that is also resilient to machine failures.

The high popularity of these systems contrasts with the lack of integrated testing solutions to ensure their general quality under normal and abnormal conditions. A main reason is the complexity of reproducing a real world environment together with a non-intrusive testing environment. This is because the scale of the system has an important effect on several testing components, such as: test controllability [6], observability, fault-injection [71], test data, oracle calculation, among others. The scale, as well as the distribution, amplifies several small details, making the testing environment deal with values that are spread throughout the system. This complexity highlights a need for a more abstract level for testing these systems.

Leveraging abstract levels [100] is precisely the main goal of model-based testing, i. e., the application of model-based engineering to perform software testing. Model-Driven Engineering (MDE) refers to the systematic use of models as primary engineering artifacts throughout the development lifecycle. D. Schmidt [109] summarizes it as a promising approach to address the system complexity allowing to develop technologies that combine:

1. Domain-specific modeling languages that formalize the application structure, behavior, and requirements within particular domains.

2. Transformation engines and generators that analyze certain aspects of models and then synthesize various types of artifacts, such as source code or alternative model representations.

In the context of software testing, the model-driven engineering can be applied to different steps, from data generation to result analysis, through deployment, execution, evaluation, and diagnosis.

In this document, the author reflects on his experience over seven years testing large-scale dynamic distributed systems. Based on that experience, he clarifies the main challenges faced during his work and most importantly, presents the main contributions of his work. He introduces the hypothesis formulated at the start of the work, describes the prototypes that were implemented to demonstrate them, and analyzes the results of the experiments achieved with the prototypes, comparing them with the expected ones. It is important to mention that all experiments presented here were conducted at real-scale.

But before presenting that experience, he motivates his work with a real-world failure example that arrived to a very popular instant-messaging software and that affected several millions of users. This example helps to clarify why large-scale bugs are different and why current techniques are not effective to validate large-scale systems.

1.2 Motivating Case: The 2010 Skype Outage

On December 22^{nd} 2010, the Skype network suffered a critical failure that lasted approximately 24 h from December 22^{nd}, 16:00 GMT to December 23^{rd}, 16:00 GMT. The failure concerned more than 23 000 000 of online users [101]. Figure 1.1 illustrates the outage. When the number of online users was almost reaching its highest point, it suddenly started to drop. In almost 1 hour, there were less than 1 million online users.

Skype is a successful example of combing modern distributed architectures to implement a popular, reliable, portable, and interoperable software. Indeed, Skype architecture is a harmonic combination of different paradigms, merging centralized, peer-to-peer, and cluster architectures. A centralized login server handles all the network connections and a Distributed Hash Table (DHT) stores user information. Communications between nodes are done through a point-to-point connection, and clusters, which act as a private cloud, provide some services such as group chat or offline messaging.

The outage commenced on December 22^{nd}, when a cluster of support servers responsible for offline instant messaging became overloaded. Because of this overload, some Skype clients received delayed responses from the overloaded servers. Clients using a specific version of Skype for Windows (5.0.0152) did not process properly these delayed responses and crashed.

Users running other versions were not affected by this initial problem. Nevertheless, around 50 % of all Skype users globally were running the 5.0.0.152 version of Skype for Windows and the crashes caused approximately 40 % of those clients to fail. Among these clients, there were 25 % to 30 % of the publicly available super-nodes.

Super-nodes are nodes with extra behavior: they help common nodes to join the network and store some user information on the DHT. When users noticed that their clients crashed, they simply relaunched their software. The problem

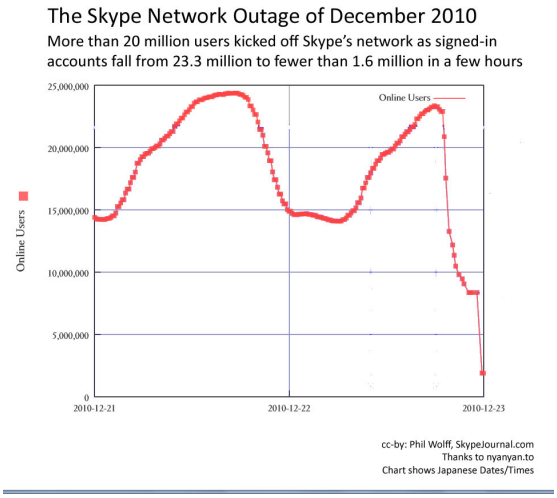


Figure 1.1: The Skype Network Outage[1]

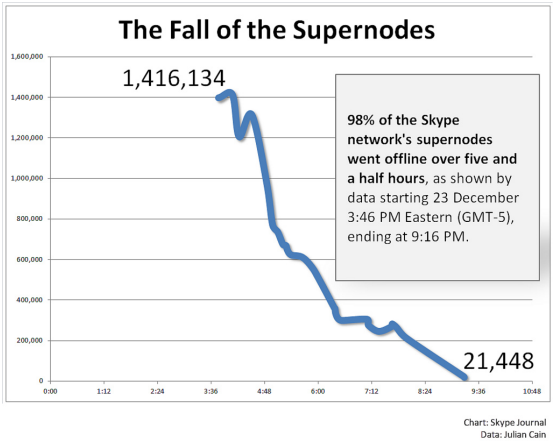


Figure 1.2: The Fall of the Supernodes[1]

is that super-nodes do not start as super-nodes, they start as common nodes and become super-nodes, if they have enough resources and are stable for a while. As the former super-nodes restarted as common nodes and tried to join the system, some of the remaining super-nodes received a traffic one hundred times greater than normal. Since Skype super-nodes are deployed on client machines, they have a built-in mechanism that avoids having a huge overload in the host machine, halting the super-node when a given threshold is reached. Thus, all super-nodes that reached the threshold left the system, surcharging the remaining super-nodes and driving the whole system into an unavoidable cascade of shutdowns. Figure 1.2 illustrates the fall of the super-nodes. From December 22^{nd} at 20:46 until December 23^{rd} 2:16 GMT, 98 % of the Skype

[1]Images by Phil Wolff. Available under Creative Commons Attribution-Share Alike 2.0 Generic License

network super-nodes were offline.

To recover the network, the engineering team added hundreds of new Skype nodes that act as dedicated super-nodes, which should have provided enough capacity to allow the network to bootstrap. However, only a small portion of users (15 % to 20 %) were "healing". The team introduced then several thousands of super-nodes, using the resources that support the Group Video Calling. These new super-nodes and the nightfall helped the network to heal. During the night, the full recovery was beginning. On December 23^{rd} at 16:00 GMT, clients could connect normally to the network. When common nodes start becoming super-nodes, engineering could start removing the dedicated ones.

This is the second major Skype outage; the first one dates back to 2007. When analyzing the causes of this outage, we notice two distinct faults:

1. the misinterpretation of server messages that were delayed causing nodes to crash.

2. the incapacity of super-nodes to treat a large number of join requests, which also prevents the system to bootstrap a large number of nodes at the same time.

A conformance testing approach could catch the first fault, if combined with fault injection (to simulate message delays). A unique test driver could individually test endpoints and reproduce the fault. Nevertheless, finding the correct sequence of messages that can drive the node into a faulty state is a complex task. Indeed, the Windows software that crashed was subject to extensive internal testing and months of beta testing with hundreds of thousands of users, without revealing this fault.

The second fault is more complex, because its reproduction is more challenging. A single test driver cannot generate sufficient load to crash a super-node. Here, a different approach is needed, either using several distributed test drivers or reproducing a real-scale scenario. Contrary to the first fault, the sequence of steps that expose this fault is rather simple, either creating a large system instantaneously or disconnecting super-nodes from a stable system. In both cases, a global model of the topology is necessary to identify the nodes that should be disconnected and to verify that the system is sound.

1.3 Research Topics

The author's first experience with large-scale systems was during the development of APPA [5], a data management system for large-scale Peer-to-Peer and Grid applications. APPA was developed in Java using JXTA [115], an open-source framework for creating Peer-to-Peer systems. Apart from all the technical drawbacks, the development revealed several issues concerning the validation of the system. More precisely, the main issue was to find a convenient approach to validate the system that could deal with its three dimensions: functionality, scalability and volatility.

A similar observation was done by the Skype development team. Indeed, the posteriori analysis of the Skype outage, done few days later, diagnosed the sources of the outage and also revealed possible lacks in the testing process. Skype, as any other large scale system, suffers from the *lack of a testing methodology.*

Along with this main key issue, i. e., the lack of methodology, the author identified three other key-issues. Each key-issue led to one or more challenges that address it. The challenges engendered several hypothesis about how distributed systems should be tested. Finally, practical experimentations verified the accuracy of the hypothesis. The following sections summarize the key issues, challenges, hypothesis, and experimentations presented in this document.

1.3.1 Testing Methodology

The specificities of large-scale systems raise new questions when preparing a test scenario for validating a system. These questions not only concern the kind of properties that can or should be tested, but also the scale of the system (i. e., number of nodes), the amount of test data, the volatility of nodes, etc. The absence of answers for these questions reveals a first issue:

Issue 1 *The absence of a systematic, disciplined, and quantifiable approach to measure the quality of large-scale systems.*

This issue leads to a first challenge:

Challenge 1 *Propose a* ***testing methodology*** *for large-scale systems.*

The methodology must specify when and how non-functional properties should be tested along with the functional ones. The elaboration of the methodology was part of Eduardo Almeida's PhD thesis [35]. Together, we formulated a first hypothesis to take up this challenge:

Hypothesis 1 *Large-scale distributed systems are tested through an incremental methodology.*

We believe that the functionality of the System Under Test (SUT) should be tested along with its scalability and its dynamicity, but that these aspects should be incrementally added to the testbed. We also believe that for improving diagnosis, the test should start with a small-scale static configuration and evolve step by step towards large-scale dynamic configuration. In general, different types of defects appear while the SUT treats different configurations, since different behaviors are needed, making different parts of the source-code to be exercised. To validate this hypothesis, we used code-coverage tools to execute test sequences on different configurations, confirming that indeed, different parts of the code were exercised.

Thus, our first contribution is a testing methodology that deals with these dimensions. The methodology aims at covering functions first on a small system

and then incrementally addressing the scalability and volatility issues [38]. The existence of a testing methodology that explicitly addresses volatility-related properties leads to a second challenge, volatility simulation:

Challenge 2 *Propose an approach to simulate the volatility of the system under test along with functional tests.*

In distributed systems, nodes are volatile by nature, they may join and leave the system at will, either normal or abnormally. Most research efforts and tools propose to randomly stop the execution of nodes [8, 80] or to insert faults in the network [59, 81]. While these approaches are useful to observe the behavior of the whole system under network perturbations, they do not focus on detecting and diagnosing software faults, especially those that are related to volatility. To consider system volatility during validation, we formulated a second hypothesis:

Hypothesis 2 *Volatility is simulated in a systematic way and integrated to test sequences.*

We claim that volatility simulation must be integrated to the test sequence, in a systematic way. That is to say, the test sequence must specify when a node, or a set of nodes, must join or leave the system. The rationale behind this hypothesis is that some behavior is executed only on particular states, which are hard to reach randomly. To validate this hypothesis, we used code coverage to show that a test sequence containing specific volatility directives (node join, exit, etc.) was able to reach a particular state in a small-scale configuration.

The methodology must also specify the system nodes should be tested singly or simultaneously, and if simultaneously, at which scale. While system nodes can be tested singly for functionality and protocol conformance, some errors only appear on heavy load situations, which can not be simulated by a single test driver. This was the case, for instance, of the crash of Skype nodes due to extra load. The same problem concerns platform simulation tools, which can build and simulate systems with thousands of nodes in a single machine, but that cannot simulate load situations with massive concurrence. Based on this, we formulated a third hypothesis:

Hypothesis 3 *Large-scale distributed systems are tested at real-scale.*

To validate this hypothesis, we used code coverage tools to demonstrate that for the same test sequence, the code coverage is directly proportional to the number of involved nodes. The testing methodology, the hypothesis and their validation are further described in Chapter 4. Testing in real-scale requires a distributed test architecture, which is the subject of next section.

1.3.2 Distributed Test Architecture

Distributed algorithms are typically validated using simulation tools [114], such as SimGrid [28], SimJava [60], etc. There are two reasons that explain this

choice. First, experiments can be easily reproduced and results are obtained locally, which simplifies the evaluation of algorithms. Second, simulation tools simplify some complex aspects of distributed software, such as asynchronous messages handling, concurrent programming and knowledge of the middleware.

However, simulation tools are not fully adapted for some types of test, such as scalability, load, or stress tests. For these types of test, a **Distributed Test Architecture**, i. e., an integrated solution for the creation and deployment of test harness in a large-scale environment, is needed. It is important to mention that distributed test architectures and simulation tools are not concurrent, but complementary. They often have different objectives and even when these objectives overlap, their results provide important information for testing and diagnosis.

Along with Issue 1, the lack of a testing methodology, a second issue appears, **the lack of an adapted test architecture**, i. e., a software that is able to deploy, execute, control and observe the system under test, on a real-scale test environment. While some ad hoc solutions exist, tailored either for the system under test or for a specific goal (e. g., test harness deployment, log analysis, fault injection, test case execution), there are few or no comprehensive testing architecture for large scale systems. Some examples of ad hoc solutions are Herriot [118] and MRUnit [32] for testing MapReduce jobs, and PeerUnit [38] and P2PTester [44] for testing peer-to-peer systems.

Issue 2 *The absence of a scalable test architecture that is able to execute, control, and observe a large scale system.*

Distributed software testing requires an efficient test architecture, which must have the ability to simulate fine-grained churn, i.e., to individually create nodes and make them join and leave the system, according to the needs of a test case. The test architecture should ensure the controllability during the execution, ensuring that a test sequence is executed in the correct order. This challenge is resumed as follows:

Challenge 3 *Provide an efficient distributed test architecture, w.r.t. test controllability and observability.*

By *distributed*, we mean that the system must be tested with distributed drivers (one per node) that are able to control individually each node, as opposed to centralized, where a unique test driver controls the whole system.

By *efficient*, we mean that message transfer within the test architecture must be scalable. Indeed, the controllability is ensured by the exchange of synchronization messages between nodes. Considering that the system under test has a large number of nodes and that each node may perform a different set of actions, synchronization messages may become a bottleneck. Therefore, the scalability is required to cope with most large-scale systems, which scale up logarithmically. To take up this challenge, we formulated another hypothesis:

Hypothesis 4 *The distributed test architecture is composed of one controller and a set of drivers (one per node under test), organized in an efficient overlay.*

The controller uses a broadcast protocol to send messages to the drivers, which in their turn, use a converge cast protocol to send messages to the controller.

We claim that the architecture should follow the Conformance Testing Methodology Framework [4] (CTMF), but using an efficient protocol for message exchange. Based on this hypothesis, we designed and implemented PeerUnit [36], an efficient distributed test architecture, our second contribution. PeerUnit was also part of Dr. Almeida's thesis.

Contrarily to traditional centralized testing environments, which scale up linearly, our environment is fully distributed and scales up logarithmically. The environment is based on an overlay network, which organizes the testers in a balanced tree [16]. The test controller and the testers communicate through the overlay, reducing the load of the controller and improving efficiency.

Each node is controlled by one test driver, which ensures the controllability and the observability of the node. The test driver is a process or an application that executes in the same logical node as a system node and controls its execution and its volatility, making them leave and join the system at any time, according to the needs of a test. Thus, testers allow the control of the volatility of the whole system at a very precise level. The architecture is further described in Chapter 3.

1.3.3 Oracle Automation

During test execution, the system under test generates output data, direct or indirectly. The output data is distributed and may be structured (e. g., databases, XML files, etc.) or unstructured (e. g., logs, system information, etc.). Output data is not restrained to a simple set of variables, it may also concern unstructured data, such as logs, monitoring data, energy consumption data, etc. In order to be used as oracle data, i. e., the input of an oracle function, the output data must be retrieved, analyzed and structured.

The oracle data is potentially a large set of values, spread across different nodes with unsynchronized clocks. Gathering all values and building a timeline increases the complexity of the oracle automation. In some cases, oracle can be calculated locally to each node and combined to form a global verdict. In these cases, the interpretation of inconclusive local verdicts (i. e., nodes that are not able to get a result in an acceptable delay) is an issue.

Issue 3 *Lack of means to automate the test oracle.*

This issue leads to another challenge:

Challenge 4 *Provide means to oracle automation.*

By *means*, we refer to efficient languages and tools to automate data transformation and comparison. To take up this challenge, we formulated a another hypothesis:

Hypothesis 5 *Oracle is automated through model-driven engineering.*

Testing properties of large-scale systems implies accessing and manipulating complex and distributed data. We claim that the model-driven engineering can be used to efficiently represent these data, to easily automate complex and distributed oracles, as well as address the lack of languages for automating the oracle. This, either by offering tool support for creating dedicated languages, or by proposing standard tools for validating data, e. g., OCL.

Based on this hypothesis, we have applied MDE techniques to build a dynamic model representing the oracle data and used MDE tools to validate this data. Therefore, our third contribution is the use of **Models to represent Oracle Data**. Further details are provided on Chapter 5.

1.3.4 Test Harness Deployment

Performing tests in real-scale is an expensive, time-consuming task. It consists of reserving a set of nodes on a computer grid or cloud, deploying the test harness on all nodes and executing a suite of tests. The deployment is particularly complex, since the test harness depends often on third-party software, which are not installed on the reserved nodes. This bring us to a last issue:

Issue 4 *Lack of an efficient and automated approach for test harness deployment.*

More precisely, there is a need of an efficient approach for software provisioning in grid/cloud computing that provides an abstraction representation of the deployment process. This need is even more critical for software testing in contrast to software deployment, since tests are performed several times during development and deployments are only performed once per release. This issue leads to a new challenge:

Challenge 5 *Handle the interdependence of software packages and automate software deployment.*

When the test harness needs a given software, the approach must know precisely the software it depends on. Based on this information, and on the selected deployment platform, the approach can choose the correct software variant that should be installed and also detect possible conflicts. The elaboration of this approach was part of Tam Le Nhan PhD thesis [77]. To take up this challenge, we make a new hypothesis:

Hypothesis 6 *Model-Driven Engineering and particularly feature models can provide an abstract representation of the deployment process.*

We claim that feature models are well-adapted to represent software variants and dependencies among them. Feature models offer and common basis for expressing software requirements, detecting invalid configurations.

We validate this hypothesis by an example showing that, given a base model representing all available artifacts, one can easily derive a configuration model (a specific use of a subset of artifacts) and generate all needed configuration scripts to generate its corresponding deployment virtual image.

Further information on model-driven deployment is provided in Chapter 6.

1.4 Structure of this Document

The rest of this document is organized as follows. Chapter 2 introduces the main concepts and techniques for testing large-scale systems and establishes the context of this research work.

Chapter 3 presents Macaw, a distributed test architecture and several experiments that validate the usability and the performance of the architecture on different configurations.

Chapter 4 presents an incremental methodology for testing large-scale systems. The methodology is applied in several experiments that validate the feasibility and the efficiency of the methodology when verifying two popular open-source peer-to-peer systems.

Chapter 5 describes the use of model-driven engineering for building an oracle for testing quality properties in large-scale distributed systems. The oracle is used to validate a particular class of properties that must be calculated globally, they cannot be calculated by a single node or by a portion of the system.

Chapter 6 introduces the use of model-based engineering for deploying software artifacts. The presented approach uses feature models to represent deployment configurations and model-based techniques to handle automatic artifact deployment and reconfiguration.

Chapter 7 enumerates the perspectives of applying model-based techniques to large-scale software testing, discusses the future directions of the research and concludes this document.

Chapter 2

Background

2.1 Introduction

In this chapter we introduce the main concepts related to large-scale dynamic distributed systems to explain the challenges of testing these systems. From now on, we will use simply *distributed systems* as a synonym of large-scale dynamic distributed systems. To clarify the presented concepts, their description relies on a simple example, a Distributed Hash Table [113], DHT.

2.1.1 Running Example

A distributed hash table is a distributed data structure that is used in peer-to-peer applications to store and retrieve data efficiently. It is composed of several equivalent nodes (different instances of the same software) distributed through a network, where each node is able to insert and retrieve data pairs $\langle key, value \rangle$. Nodes use a hash algorithm to affect pairs to system nodes.

Figure 2.1 depicts a UML component diagram representing a typical node of a distributed hash table. The rectangles represent the components of the node and the small rectangles overlapping the component borders represent the ports. Lollypops and sockets connect ports from different components. Lollipops represent the provided interfaces and sockets represent the required ones. UML distinguishes ports from interfaces: interfaces specify the nature of the interactions that occur over a port. A port can be associated to multiple interfaces.

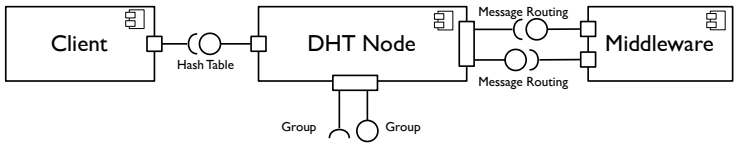


Figure 2.1: UML Component Diagram Representing the Interfaces of a DHT Node

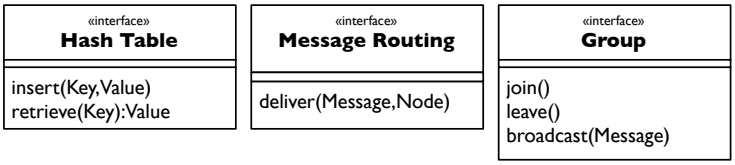


Figure 2.2: UML Interface Specification of a DHT Node

In this diagram, the component `DHT Node` has three ports and provides two interfaces. Clients of the node access its functionalities through the **Hash Table** interface. The node interacts with other nodes through the **Message Routing** interface.

Figure 2.2 depicts these two interfaces. The `Hash Table` interface specifies two methods for inserting and retrieving data pairs. The `Message Routing` interface specifies a method for delivering/routing messages to other nodes.

When a node has enough resources and is stable for a certain time, it becomes a *group-node*, creating a group and allowing other nodes to join and communicate through this group. This new behavior is dynamically provided by an additional port, summarized in the **Group** interface (Figure 2.2). Other nodes can use this interface to join and leave a group, as well as send message to group members.

2.2 Large-Scale Distributed Systems

A distributed system commonly defined as:

> "a piece of software that ensures that a collection of independent computers appears to its users as a single coherent system [117]".

The adjective *large-scale* often relates to systems with thousands or millions of nodes, where each node has only a partial view of the whole system. The system interacts with its environment through a set of distinct interaction points, called ports. The adjective *dynamic* concerns both, the size of the system and the distribution of ports. The number of nodes varies along time, as well as the ports that are available at each node. A port gives access to a functionality of the system and groups coherent input and output messages related to that functionality. We define a distributed system as follows:

Definition 1 *A distributed system is a pair $\mathcal{S} = \langle \mathcal{N}, \mathcal{P} \rangle$ where:*

- *$\mathcal{N}$ is the set of nodes $\mathcal{N} = \{n_1, n_2, \ldots, n_n\}$ that composes this system.*
- *$\mathcal{P}$ is a set of ports $\mathcal{P} = \{p_1, p_2, \ldots, p_m\}$ through which the system receives inputs and sends outputs.*

Definition 2 *The topology of $\mathcal{S}$ can be represented by a directed graph of out-degree $deg^+ = O(\log\ |\mathcal{N}|)$ and diameter $O(\log\ |\mathcal{N}|)$, where each node $n \in \mathcal{N}$ is a vertex and every entry in its routing table is an edge to a neighbor.*

Definition 3 *A port gives access to a functionality of the system and groups coherent input and output messages related to this functionality. The terms upper and lower ports refer to the external and internal points of interaction of the system, respectively.*

The number of available ports on a node depends on the state of the system, which decides to add more functionalities to a node, or to remove them. Nodes can share a same port, which provides an equivalent access to the system. We will let p_i^j denote the port p_i on node n_j. For the purpose of testing, it is important to distinguish the *upper* and the *lower* ports of a node, since the architecture controls directly the upper ports, while the SUT controls the lower ports.

The terms upper and lower ports are common vocabulary in software testing, especially in distributed and conformance test. In our example, the **Hash Table** interface specifies the upper port of the node and the **Message Routing** and the **Group** interfaces specify its the *lower* ports. The lower ports often require a third-party middleware to communicate: CORBA, Java RMI, Rest, etc.

In our running example, nodes share their upper port. Interactions with the system through the **Hash Table** interface are equivalent, independently from the node where interactions occur. Calls to the operations **insert()** or **retrieve()** produce the same functional behavior, albeit with different performances. The port specified by the **Group** interface is dynamic and not shared. Its availability depends on the state of the system and its functional behavior depends on the node where the interactions occur.

2.3 Characteristics of Large-Scale Systems

Large-scale dynamic distributed systems combine the characteristics of several systems: traditional distributed systems [33, 122], grid computing [52], ad hoc networks [96], peer-to-peer computing [86], dynamic adaptive systems [95], and cloud computing [10]. This section presents the main common characteristics of these systems:

Scalability Systems are expected to connect a large number of nodes (from thousands up to several millions), where each node only interacts with an arbitrary small part of the system.

Autonomy Nodes are autonomous, may refuse to answer to some requests, and even unexpectedly leave the system at any time (and rejoin afterwards).

Dynamicity Resources may be dynamically added to or removed from the system. This concerns physical nodes, as well as software services, meaning that nodes may change their behavior through time. Specific requests may be redirected dynamically to different nodes, depending on load. The system is also self-organizing: a specific input data set can cause a different path to be executed because the previous path no longer exists.

Heterogeneity of resources The nodes that compose the system are heterogeneous with respect to hardware and software. Therefore, the quality and the processing power of nodes is variable. In many cases, nodes use different versions of software or communication protocols.

Diversity of purposes These systems are used on different domains, from data sharing to massive data processing applications. Consequently, they have different requirements concerning input data: size, availability, etc.

Stateless protocols Nodes may receive events defined in their interfaces in any order and at any moment. In essence, communications consist of independent pairs of requests-responses.

Volatility Nodes are volatiles, they may join and leave the system at will, either normal or abnormally. In some systems (e. g., peer-to-peer), the volatility is an expected behavior: the system expect them to leave at any time. Some systems (e. g., MapReduce) are usually deployed on large clusters of commodity hardware, where failures happen constantly.

Third-party infrastructure Systems often rely on third-party middleware: Remote Procedure Call, Message Exchanging, Brokers, Service Oriented Architecture, etc. Moreover, parts of the system (e. g. infrastructure, services) may belong to other entities.

Symmetry Several nodes play identical roles, ensuring reliability (there is no single point of failure) and load balance (load is distributed symmetrically across nodes). Nodes may run different instances of the same software and a port may be shared by different nodes.

Non-determinism The thread execution order may be affected by external programs or by the network latency. Thus, it is difficult to reproduce a test execution and some defects do not appear on all executions.

Partial failures A failure in a particular node may prevent a part of the system from achieving its behavior.

Timeouts Timeouts are used extensively to avoid deadlocks. When a response is not received within a certain time, the request is aborted. This is not an error and the system must perform correctly whether a request is answered or not (albeit perhaps differently).

Elasticity The system scales out and in quickly, adapting itself to load.

2.4 Software Testing

Software testing is commonly defined as:

> "The process of operating a system or component under specified conditions; observing or recording the results, and making an evaluation of some aspect of the system or component" [1].

The process of software testing can be either dynamic or static, depending on whether or not the software is executed [17]. Static testing is mainly applied for checking the sanity of the code and/or generating relevant input data for dynamic testing. In contrast, dynamic testing involves the execution of the software and is widely applied to several types of testing: conformance, load, stress, etc.

Figure 2.3 illustrates the main activities involved in dynamic testing. The main goal of dynamic testing is to execute the System Under Test, **SUT** and validate one or more properties. A **Test Sequence** (or Procedure) is a program that reads **Test Data** and interacts with the **System Under Test**, through its public interface, driving it into a given state. This execution engenders a **Result**, i. e., a set of output data, generated directly (e. g., files, logs, graphical interfaces, etc.) or indirectly (e. g., resource usage, energy consumption, etc.) by the SUT.

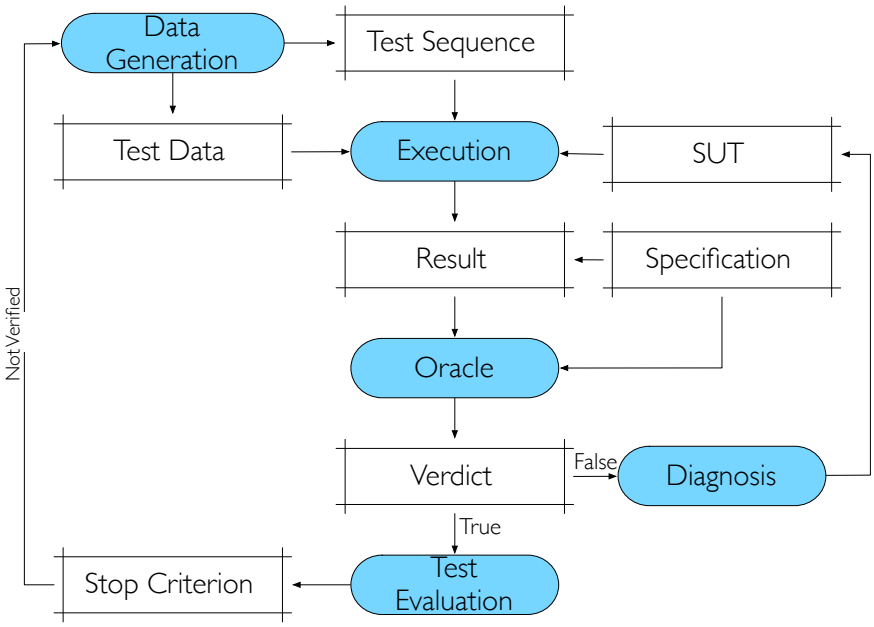


Figure 2.3: Dynamic Test

The **Oracle** is a function that analyzes, total or partially, the result and verifies if it corresponds to the **Specification** of the SUT. The Oracle issues a **Verdict**, which can be *True/Pass*, *False/Fail*, or *Inconclusive*, if the result is insufficient to assess the verdict.

If the verdict is false, i. e., a property is not validated, the **Diagnostic** activity is performed. The goal of the diagnosis is to isolate the portion of source code containing the error that causes the failure detected by the oracle. If the verdict is true, i. e., a property was validated, another activity is performed, the **Test Evaluation**. The goal of the evaluation is to estimate if the test meets the expected quality, i. e., if the SUT was enough tested or not.

If the expected quality is not reached, the test data and script must be improved.

There are different criteria to evaluate the quality of a test: mutation analysis, code coverage, number of errors found, time, number of test cases, size of test data, etc. When the expected quality is met, the test stops.

2.5 Distributed Software Testing

Distributed software testing relates to a system-level functional testing, with a notable need for validating non-functional properties: security, scalability, elasticity, reliability under stress, etc. Distributed test execution follows the same scheme as dynamic test with several particularities: distributed execution, harness deployment, result retrieval, execution synchronization, failure simulation, and scale variation.

Since the SUT has physical distributed ports, the test execution must be distributed and a test harness must be deployed on different nodes. A typical test harness contains the node's software and its dependencies (libraries), (partial) test case sequences, input data, and some mechanism to retrieve and order the output data (i. e., build the timeline). The test execution must ensure the synchronization among the distributed test sequences and simulate node-level failures: network connection removal, node shutdown, threads interruption, etc. When executing in large-scale scenarios, with third party nodes, test repeatability cannot be reached.

The goal of a *Distributed Test Case* is to interact with the distributed ports of the system under test and to verify dynamically if a feature is correctly working according to certain quality criteria.

Definition 4 (Distributed Test Case) *A Distributed Test Case noted τ is a tuple $\tau = \langle \mathcal{N}^\tau, \mathcal{I}^\tau, \mathcal{O}^\tau, \mathcal{A}^\tau, \mathcal{M}^\tau, \Omega^\tau \rangle$ where:*

- $\mathcal{N}^\tau \subseteq \mathcal{N}$ *is a set of nodes,*
- $\mathcal{I}^\tau$ *is a collection of inputs,*
- $\mathcal{O}^\tau$ *is a collection of outputs,*
- $\mathcal{A}^\tau$ *is a sequence of actions,*
- $\mathcal{M}^\tau$ *is a set of coordination messages, and*
- Ω^τ *an Oracle.*

The oracle analyzes the system outputs ($\mathcal{O}^\tau$), compares them with the expected outputs and provides a verdict. Since outputs may be replicated on different nodes, the oracle must deal with distributed data and, depending on the output, the verdict can be calculated locally to each node and the final verdict is calculated in function of all local verdicts.

The volatility and the scale of the system have a direct consequence on oracle assessment. The system does not provide complete and correct answers, but the best k answers (top-k) that can be calculated in a given time. In the case

of replicated ports, some systems may accept that some ports do not find any answer within the expected time and the local oracles cannot assign a verdict. The verdict is thus not only a question of correctness but also a question of response time and of ratio of acceptable answers (w.r.t. conformity).

The distributed test case interacts with the system under test through actions. An action is a sequence of instructions as well as any statement in the Distributed Test programming language. An action is a point of synchronization, it ensures that all its instructions were executed, in all concerned nodes, before allowing the execution of the next action in the test sequence. An action contains the input data sent to a given port, a set of nodes that receive the input data, and the generated output data. Actions are associated to a timeout, avoiding deadlocks during the execution of distributed test cases.

Definition 5 (Action) *An action is a tuple* $a_i = \langle i_i, p, \mathcal{N}_i, o_i, \iota \rangle$ *where:*

- $i_i \in I$ *is an input,*
- p *is a port,*
- $\mathcal{N}_i \subseteq \mathcal{N}$ *is the set of nodes that receive the input,*
- $o_i \in O$ *is an output, and*
- ι *is the interval of time in which the action should be executed (timeout).*

Algorithm 1 presents an example of a distributed test case for testing the **Hash Table** interface, from the running example. This distributed test case has only three actions. The first action has a message-call as input (**insert(33,'France')**), interacts with node n_1 through the port **Hash Table**, has no output and has no timeout. The second action also has a message-call as input (**retrieve(33)**), interacts with all nodes of the system through with port **Hash Table**, has a set of values as output and has no timeout. Lastly, in the third action, it compares all responses (R) with the inserted value. The execution must ensure that the pair was inserted before retrieving it and that all values were retrieved before they are compared with the inserted value.

Algorithm 1: Distributed Test Case Example: HashTable Interface

Input:
$\mathcal{N}$: a set of nodes $\mathcal{N} = \{n_1, n_2, \ldots, n_n\}$
begin
  send $insert(33, "France")$ **to** n_1 ; `/* Action 1 */`
  $R \leftarrow$ **send** $retrieve(33)$ **to** $\forall n \in \mathcal{N}$; `/* Action 2 */`
  assert $\forall r \in R : r =$ "$France$" ; `/* Action 3 */`
end

2.6 Distributed Test Architecture

The term *Controllability* defines the capability of the test architecture to send input events at corresponding ports in a given order [27]. The controllability

of the test depends on the *Observability* of the SUT, which is the capability of the test architecture to determine the output events and the order in which they have taken place at the corresponding ports [27]. The diversity of third-party middleware, protocols, and interfaces hampers the observability of ports, especially for lower ports. While testers control and observe directly upper ports, the observability of lower ports is more complex and requires the use of particular techniques such as packet capture, proxies, code instrumentation, etc.

Test architectures for distributed software typically rely on two components: the *Controller* and the *Tester* or *Test Driver*. The architecture places a tester at each node and the tester at node n only observes events of the ports of node n. The tester of node n also allows the controller to remotely interact with the ports of node n. The controller executes distributed test cases, sending inputs to ports and receiving the generated outputs. The controller sends and receives coordination messages to guarantee the controllability of Distributed Test Cases. In some architectures, some coordination messages are sent directly between testers, without passing through the controller.

For instance, if a distributed test case specifies that an output $!o_i$ on port p_i^n must be observed before sending an input $?i_j$ to port p_j^m, a coordination message must be sent, directly or indirectly (through the controller), from tester t_i to tester t_j. When coming to large-scale systems, the sending of coordination messages, input, and output data, becomes a bottleneck. This bottleneck increases the cost of a test in terms of time and resource consuming and may even prevent its execution.

Chapter 3

Distributed Test Architecture

3.1 Introduction

The high popularity of large-scale systems contrasts with the unavailability of extensive architectures to test them. A main reason is their disparateness of purposes and architectures, which encourages the development of ad hoc architectures, tailored for the System Under Test (SUT), including: Herriot [118], MRUnit [32], PeerUnit [38], and P2PTester [44].

The disparateness of these systems has an important effect on several testing aspects, such as: observability, test data, and test oracle. While some systems, have at least a public (i. e., observable) port at each node (e. g., distributed hash tables), others, only have public observable ports in a master node (e. g., MapReduce), complicating the observability. In these systems, the behavior of most nodes cannot be directly observed and the test architecture must resort to log analysis, resource or network monitoring. The availability of ports spread across several nodes raises another problem for the test architecture that is to provide the correct input data in front of the correct port, without being intrusive. Indeed, the transfer of huge sets of test data during the execution may overload the network and consequently disturb the behavior of the SUT. Furthermore, the calculation of the test oracle also depends on the particularities of the SUT. The oracle can either be calculated locally on a single node (1 verdict in 1 node), distributively on several nodes (n verdicts on n node), or globally (1 verdict in n nodes), when the properties that should be validated depend on values that are spread throughout the whole system.

To deal with this disparateness, an extensive test architecture must provide a rich set of features for testing: e. g., system monitoring, log analysis, test data generation, etc. However, each new feature consumes resources and since the test architecture shares resources with the SUT, the presence of new testing features may disturb the behavior of the SUT and reduce its testability. Under the software architecture perspective, the test architecture should be dynamically adaptive to test requirements: scale, input data, availability of properties, and

observability. In short, the architecture should be able to (i) dynamically deploy components, artifacts, and data; and (ii) modify its topology and its protocols at runtime.

In this chapter, we present Macaw, a distributed test architecture for large-scale dynamic distributed systems. Macaw is composed of a set of components, allowing to control and monitor the SUT, as well as a language for deploying distributed test harnesses. Macaw was build on the top of Kevoree, an open-source dynamic framework, which supports the dynamic adaptation of distributed service-based systems. Kevoree provides a domain-specific language to build and modify the architecture model of the system, and runtime platform implementations to deploy and execute the test architecture on different devices.

The rest of the chapter is organized as follows. Section 3.2 introduces some fundamental concepts of large-scale dynamic distributed systems test. Section 3.3 presents our test architecture. Section 3.4 describes its implementation. Section 3.5 presents a simple example of how the test architecture can be adapted and used to test a distributed system. Section 3.6 concludes.

3.2 Testing Large-Scale Systems

In this section, we introduce some concepts and terms related to large-scale systems, the main difficulties for testing these systems, and the requirements for an architecture to test these systems. To simplify the understanding, the description of concepts relies on the running example (Section 2.1.1), a distributed hash table [113], which is also the subject of the experiment presented in Section 3.5.

3.2.1 Test Architecture Requirements

The main characteristics of large-scale systems (Section 2.3) have a direct impact on the test architecture, and must be considered during its development. This section enumerates the key technical features that should be provided by a test architecture for large-scale systems:

Dynamicity Due to the *diversity of purposes* of the systems, the architecture should adapt itself to the SUT. To tailor specific architectures tailored according to the SUT, the test architecture should be able to deploy dynamically components and artifacts.

Scalability The performance of the test architecture and especially of the coordination messages exchange may impact the behavior of the system. Thus, in order to reduce this impact and improve testability, the architecture should scale at least as well as the SUT. This concerns data transfer, actions, and coordination messages.

Controllability Due to the *heterogeneity of resources*, the execution time of test actions may vary. Thus, the architecture must provide an efficient mechanism to ensure the correct execution of distributed test cases, even

across a distributed setup and upon churn. It must also avoid that the volatility of nodes prevents the correct termination of the execution. Moreover, complex coordination problems, involving shared ports and *multiple instances*, should not lead to an overhead that affects the SUT.

Efficiency The test architecture shares resources with the SUT (e. g., memory, processor, network, etc.) and may disturb its behavior. Therefore, the architecture must be as efficient as possible.

Observability The difficulty to observe lower ports affects the capability of the test system to determine the outputs and the order in which they have taken place at the corresponding ports. Since there is no extensive solution, due to the diversity of the used *third-party infrastructure*, the architecture must provide facilities to create ad hoc solutions to improve the observability of ports.

Volatility simulation *Partial failures* are a frequent cause of system failures, and for some systems, e. g., peer-to-peer, MapReduce, they are also a common and expected behavior. The architecture must provide an individual control of nodes, to correctly control the joins and the departs of each node (e. g., to simulate volatility), allowing to evaluate the tolerance of the whole system to partial failures.

Deployment facilities The test architecture works with a high number of nodes, which must be installed, configured, and cleaned. Therefore, the architecture must provide functionalities to describe the configuration of the test harness for different nodes, and to deploy the harnesses on a distributed environment.

Variable sharing During a test, some properties are only known dynamically, during the execution, and by few nodes, e. g., node ids, number of nodes, etc. The architecture should provide a mechanism that allows testers to share variables.

Complex data structures Due to the *diversity of purposes*, the oracle must deal with large sets of complex data structures: graphs, trees, etc. The architecture must simplify the development of oracles that manipulate this data.

3.3 Macaw Architecture

Macaw is composed of a set of components, allowing to control and monitor the system under test, as well as a language for deploying distributed test harnesses. Macaw is built with Kevoree, an open-source dynamic framework, which supports the dynamic adaptation of distributed service-based systems. Kevoree provides a domain-specific language to build and modify the architecture model of the system, and runtime platform implementations to deploy and execute the test architecture on different devices. The test architecture is also a large-scale system. Its architecture is based on a set of components, which can be combined according to the requirements of a test scenario.

This section presents the main artifacts, components and libraries of the test architecture, their deployment and how a distributed test case is executed. The architecture of Macaw meets most of the requirements presented previously in Section 3.2.1: controllability, observability, volatility simulation, variables sharing, complex data structure, and deployment facilities. The component model, used to implement the architecture and presented in Section 3.4, ensures the remaining requirements: dynamicity, scalability and efficiency.

3.3.1 Architecture Artifacts

Artifacts are pieces of information used during the deployment and the operation of the test architecture (e. g., scripts, configuration files, documents, etc.). Unlike components, which are generic, artifacts are specific to the system under test or to the test goals. Therefore, they must be defined for each test scenario.

The **Adapter** artifact describes the interface of a node under test and adapts its interface. The description associates an information to available methods, explaining whether they are synchronous or asynchronous. The adapter specifies which actions (Definition 5) are accepted by the nodes under test. The adapter also translates proprietary types into standard types, which can be serialized and transmitted through the network. Additionally, the adapter defines two methods, for setting up and cleaning the environment for the node under test. Adapters depend on the system under test: they can be reused in different tests of the same system.

The interface described by the adapter allows the interaction between **Test Sequences** and the system under test. A test sequence is a partial implementation of a distributed test case (Definition 4), it specifies the sequence of steps of a distrbuted test case. It is a program that specifies a sequence of interactions that drive the SUT to a given state, where some property (the test objective) can be checked.

Another artifact implements the oracle part of the distributed test case: the **Oracle**. The oracle is a program that retrieves the output data generated during the execution of the test sequence, analyzes them, and provides a verdict.

The test architecture has two more artifacts: **Test Data** and **Deployment Plan**. The test data corresponds to the input data of the distributed test case. It is read by the test sequence and sent to the nodes under test. The deployment plan artifact is a program that specifies how the SUT and the test architecture should be configured and deployed. It is independent from the test sequence and adapters and meets the *deployment facilities* requirement.

3.3.2 Architecture Components

The test architecture has three mandatory components: the **Controller**, the **Upper Tester** and the **Lower Tester**. The other components are optional and can be deployed according to the particularities of the SUT and the test requirements.

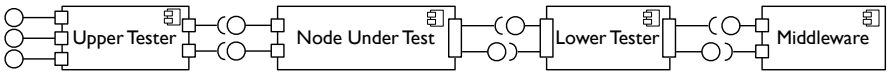


Figure 3.1: UML Component Diagram Representing a Node Under Test Between the Upper and the Lower Testers

The controller is the main component of the test architecture. The controller executes test sequences, sending actions to nodes under test through the upper testers, ensuring their controllability. More precisely, it reads test sequence artifacts, ensures that there are enough available resources, dispatches actions to the testers, and retrieve results. The controller interacts with upper testers, as described below.

Upper testers are deployed on the same logical nodes as the nodes under test, but run in independent processes. They interact with both, the controller and the upper ports of the node under test. Upper testers control the volatility of nodes under test, meeting the *volatility simulation* requirement. Indeed, they make nodes join and leave the system, translate test sequence's actions into method calls, and to force nodes under test to abnormally quit.

Together, the controller, the upper and the lower testers meet the *controllability* and the *observability* requirements. Indeed, the upper ports of the node under test are usually locals and cannot be accessed from remote processes. Thus, the goal of the upper tester is to make ports accessible, occasionally replacing future objects [72] by a return message. Upper testers use adapters to learn how to interact with the node under test.

Figure 3.1 illustrates the difference between the upper and the lower testers. While the upper tester only interacts with the upper ports of the node under test, the lower tester is placed between the node under test and the middleware. Note that the upper tester provides ports to the controller and uses/requires ports from the node under test.

The **Data Provider** component is deployed in the same logical node as the upper tester. Its goal is to store test data, i. e., input and output data, to reduce the network traffic during the execution of a test: input data is sent to components before the execution and output data is returned to the controller after the execution. The traffic reduction meets partially the *efficiency* requirement, with respect to network usage.

The **Dictionary** component is also deployed in the same logical node as the upper tester. It allows testers to share variables during the execution of a test, meeting the *variable sharing* requirement. Contrarily to the data provider component, the data stored in the dictionary are available to all nodes.

The **System Monitor** component may be deployed in the same logical node as the node under test. Its goal is to periodically store monitoring data about the system and the node under test process: memory, processor load, disk, network. The monitoring data is stored locally and is transferred to the controller after the test sequence execution.

At last, the **Logger** is composed of two components: client and server. The

Associative Array	
Grow	Doubles the size of the array.
Shrink	Reduces by 50% the size of the array.
Shuffle	Mixes the contents (values).
Graph and Tree	
Reduce	Removes 20% of the nodes.
Raise	Adds 20% new nodes.

Table 3.1: Data Mutation Operators for Improving Test Data

client component is deployed along the node under test, and stores their logs locally. The server gathers all local logs and rebuilds a timeline. Both, the monitoring data and the logs may be used for oracle purposes, meeting the *observability* requirement.

3.3.3 Architecture Libraries

Since large-scale systems manipulate large sets of complex data, which must be persisted, transferred through the network, and compared, the architecture offers a complex data type library along with components and artifacts. This library meets the *complex data structure* requirement. The library contains the following complex types: graph, tree, and associative array.

For each type, the library proposes operators for comparing, checking properties (e. g., the diameter of a graph) and mutation operators, for modifying test data. Table 3.1 summarizes the data mutation operators. The Associative Array type has three mutation operators: Grow, Shrink and Shuffle. The first one doubles the size of the data, creating new entries. The second one discards half of the entries, and the last one changes the order of the data within the Array. The Graph and the Tree types have two mutation operators: Reduce and Raise. The first one randomly removes nodes, shrinking the size of the Graph (Tree) by 20%. The latter randomly adds new nodes to the Graph (Tree), growing its size by 20%.

3.3.4 Test Sequence Deployment and Execution

The deployment plan artifact drives the deployment of the test architecture, as well as the deployment of the system under test. Figure 3.2 presents the minimum configuration that must be deployed to execute a test sequence. The test sequence is executed by the test controller, which is deployed on an independent node. The controller interacts with several upper testers, which in turn, interact with only one node under test. Since the upper tester is able to start and stop the node under test, it must run on a different process. Conversely, the lower tester must run in the same process as the node under test.

The data provider, the dictionary and the system monitor are deployed on the same logical node (i. e., process) as the upper tester. However, the client part of the logger must be deployed in the same logical node as the node under test.

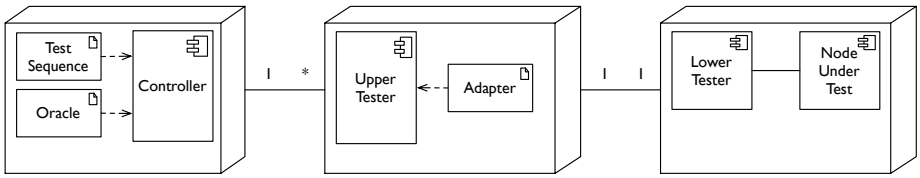


Figure 3.2: UML Deployment Diagram Representing the Minimum Architecture Configuration for a Test Execution

In Macaw, the execution of a test sequence begins with the deployment and the execution of the controller. After starting, the controller reads the deployment plan to create the test architecture. The deployment consists of dynamically deploying the required components and starting all components. When starting, each upper tester reads its adapter, to start the node under test and to discover the types of actions that are available in the node. It also connects to the controller to receive an arbitrary identification. The controller uses this identification to distinguish nodes during the execution of test sequences. Data providers load the data needed for the execution. Finally, loggers connect to their server part. However, they only transfer log data after the execution.

Once all components finish starting, the controller reads and executes the test sequence. Data generated during the execution (i. e., the output) is stored locally. After the execution, upper testers stop the nodes under test and send the output to the controller. Finally, the oracle calculates a verdict for the test.

3.4 Macaw Implementation

Macaw was implemented in Java, using the Kevoree component model, described in the following sections. Kevoree separates the components from the communication channels, simplifying the implementation of components. In this section, we introduce Kevoree and describe the implementation of Macaw.

3.4.1 Kevoree

Kevoree [53, 54] is an open-source dynamic component model[1], which relies on models@runtime [23] to properly support the dynamic adaptation of distributed systems. Kevoree was influenced by previous work that we carried out in the DiVA project [87]. Kevoree provides an architecture model for managing a component-based software architecture. This model relies on concepts of the underlying infrastructure: resources, logical nodes, and their topology. The dynamic nature of Kevoree allows Macaw to meet the *dynamicity* requirement, presented in Section 3.2.1.

Figure 3.3 presents a general overview of the models@runtime approach. The architecture of the system is captured at runtime by a model, which works as an offline reflection layer. Changes to the system architecture are first applied

[1] `http://kevoree.org`

to an unsynchronized version of the runtime model, generating a new model. Once all changes are done, Kevoree validates the new model, to ensure that the new configuration is well-formed. Then, Kevoree compares this new model with the current one, generating an *adaptation model*, which contains the set of reconfiguration commands to migrate from the current model to the new one. Finally, the adaptation engine executes these reconfiguration commands in a transactional way. If the execution of a command fails, the adaptation engine rollbacks every modifications to ensure system consistency.

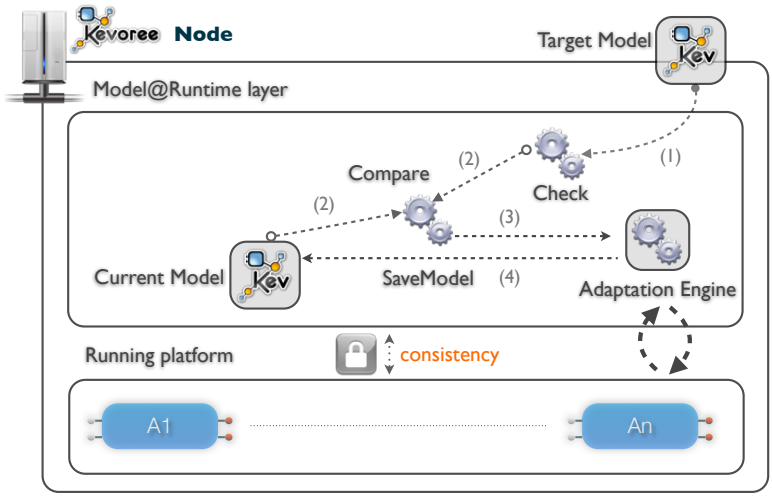


Figure 3.3: Models@Runtime Overview

3.4.2 Kevoree Features

The component-based approach is well-suited to adaptive system design. We describe below the Kevoree features that support software deployment and adaptation.

Type/Instance pattern As the system peforms updates continuously, we need a clear separation between the functionalities that the application must have and the location or configuration of these functionalities. All Kevoree concepts (Component, Channel, Node) follow the Type Object pattern [70] to separate deployment artifacts from running artifacts. While adaptation on instances is done at runtime, modeling activities related to type definitions are typically performed offline, and synchronized with the runtime later, after validation.

Component A component provides a set of functionalities that are exposed to other components. It also requires functionalities from other components. All these functionalities are identified by a (required or provided) component port. One of the most important features of a component-based model is that all the components are substitutable, so a component can be replaced by another one, provided this new one offers at least the

same functionalities. This feature allows (at design time or at runtime) for easy reconfiguration of applications, while maintaining the required functionalities.

Channel An application also defines how components are bound to exchange data. The bindings are done with *channels* that encapsulate communication semantics between components. For instance, a channel semantics can encapsulate broadcast diffusion or distributed transactions. Channels are completely independent from components. Some examples of available channels are Sockets, NIO, and Gossiper.

Node A distributed infrastructure is characterized by the use of multiple computational nodes. Each node instance may host software (components and channels) and other nodes. The nodes are organized hierarchically, where the parent are responsible to start/stop child nodes. In a nutshell, each node instance can be viewed as a container that provides an isolation level and has the responsibility of ensuring the synchronization between the architecture model and the runtime. This responsibility is represented by the adaptation capabilities of the node. These adaptation capabilities are provided by commands, which perform migration actions between two configurations (i. e., two models). Kevoree especially targets heterogeneous systems through its model@runtime approach allowing to tame adaptations on different kinds of devices (e. g., JavaSE, Android, Arduino μController, and cloud virtual nodes).

Group Whereas channels are used to define communication between components, groups are used to add shared communication between nodes, to synchronize the overall system configuration and to disseminate reconfigurations. More precisely, adaptation information (leading to a new architectural model of the system) are sent to nodes using a group to ensure consistency of the overall system.

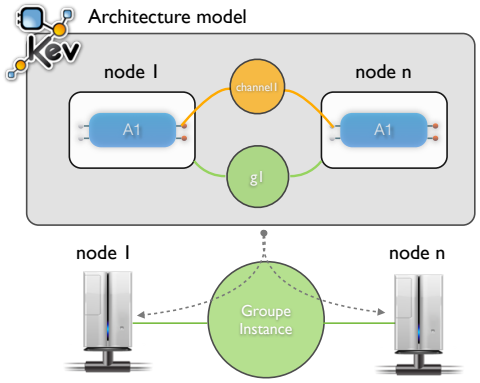


Figure 3.4: Distributed Reconfigurations

Groups can be bound to several nodes (named members), allowing them to explicitly define different synchronization strategies for the overall distributed system. Figure 3.4 illustrates this architecture organization. Additionally, a Group also defines a scope of synchronization, i. e., it defines

which elements of the global model must be synchronized for the group's members. This provides an access control policy.

Kevoree Script (KevScript) Kevoree also provides a domain-specific language, KevScript, which allows the adaptation of software architectures through the instantiation and manipulation of nodes, components, channels, and groups. It was inspired by other software architecture scripting languages, such as FScript [34]. The KevScript language is used by reasoning engines to dynamically build new models (new system configurations) from the current one. The adaptation is done by binding or unbinding components and channels and by migrating components between nodes.

Listing 1 presents the main KevScript commands. The **merge** command loads a library, containing Kevoree types (nodes, components, etc.). For instance, line 1 loads the JavaSE node type. The **set** command changes the values of an instance parameter, e. g., line 2 sets to 8080 the value of port for **component1**. The **add** and **remove** commands adds (removes) an instances to (from) the adaptation model, e. g., line 6 adds **component1** of type **ComponentType1** to the model, and line 12 removes **component3** from the model. The **move** command moves component or child nodes from a node to another, e. g., line 17 moves **component1** from **node1** to **node2**. Finally, **bind** and **unbind** commands binds and unbinds component ports and channels, e. g., line 9 binds the port **providedPort1** from **component2** to channel **channel1**, and line 11 unbinds them.

Listing 1: Main KevScript Commands

```
1  include mvn:org.kevoree.library.javase.javasenode:release
2  add component1:ComponentType1
3  add channel1 : ChannelType1
4  add node1@node2
5  add group1 : GroupType1
6  attach node1 group1
7  set component1 {port="8080"}
8  set channel1 {replay="true"}
9  set group1 {broadcast="true"}
10 set node1 {OS="Ubuntu-10.04"}
11 bind component1.requiredPort1 channel1
12 bind component2.providedPort1 channel1
13 unbind component3.requiredPort1 channel1
14 unbind component2.providedPort1 channel2
15 remove component3
16 remove channel2
17 move component1@node1 node2
18 remove node1@node2
19 move node1@node2 node3
20 remove node1
21 remove group1
```

3.4.3 Component and Artifact Implementation

Macaw components[2] described in Section 3.3, except the lower tester, are implemented as Kevoree components and make use of several third party compo-

[2]The source code is available at https://github.com/sunye/Macaw

nents. The controller uses the Apache Camel[3] integration framework to send synchronous and asynchronous messages to upper testers and to receive their responses. For instance, when the controller invokes an action, it sends a Kevoree message, containing a unique id, a method name, a set of parameters and a timeout. Coordination messages are also implemented with Kevoree messages.

Three components need to store persistent data: the data provider, the system monitor and the logger. All persistent data is stored in H2[4], a lightweight relational database management system written in Java. The system monitor uses the System Information Gatherer[5] (SIGAR) library from Hyperic to retrieve monitoring data. The logger provides standard log handlers that are compatible with the Java Logging framework and with Log4j. This allows the logger to retrieve the logs of systems using one of these two frameworks.

The implementation of the upper tester has an additional behavior. Besides controlling and routing actions to the node under test, it provides *hooks* for incoming and outgoing messages. This allows components that belong to the same node to acknowledge the arrival of actions and the departure of responses. more precisely, the logger and the system monitor can group their records by action, and the data provider can add and retrieve data to actions and responses.

Since the lower tester runs on the same process as the node under test, which is not necessarily a Kevoree component, it cannot be implemented as such. The lower tester exposes the upper interface of the node, to make it accessible to other logical nodes or processes. It uses the adapter to discover the available methods and to translate incoming action into method invocations. The lower tester is also responsible for replacing method arguments with future objects by an asynchronous message, which is sent when the response of the future object is available. This because future objects cannot be used through different nodes.

In Macaw, adapters are implemented as Java classes, containing specific annotations: **@Action**, **@Setup** and **@Cleanup**. The first annotation specifies that the method can be called by the test sequence. The other annotations specify the methods that should be called before and after the execution of a test sequence. Typically, this methods are used to configure a database or a log folder, before starting the node under test, and cleaning up the generated files after the execution. Test sequences are also implemented in Java classes, as Java methods. Deployment plans are implemented in KevScript.

3.5 Experiments

In this section, we present a preliminary experiment of using Macaw for setting up an experimental validation of a popular open-source distributed hash table, FreePastry[6], an implementation of the Pastry algorithm [106] from Rice University. The objective of this experience is to validate the usability and efficiency of Macaw.

[3] `http://camel.apache.org/`
[4] `http://www.h2database.com`
[5] `http://www.hyperic.com/products/sigar`
[6] http://freepastry.rice.edu/FreePastry/

3.5.1 FreePastry Test Specification

A FreePastry system is composed of a set of similar nodes, which run the same software. All nodes have a similar behavior, except for the *bootstrapper* node, which also helps other nodes to join the system. A node provides the bootstrapping behavior at launch, either when it does not find a bootstrapper node, or when it is explicitly asked for (in the command line). The system provides the same interface in all nodes, allowing data insertion and retrieval. This interface is similar to the running example **Hash Table** interface presented in the Section 2.1.1. Interaction through this interface yields the same behavior in any system node.

In this section, we present a distributed test case for testing the main functionality of FreePastry, i. e., its ability to store and retrieve distributed data, as well as its reliability under low churn rates. To achieve these goals, the distributed test case creates a small FreePastry system, with ten nodes. Then, it inserts some random-generated data (i. e., a set of $\langle key, value \rangle$ pairs) in an arbitrary node and retrieves this data (from the previously inserted keys).

In summary, the distributed test case has the following steps:

1. FreePastry system start-up.
2. Test data insertion on an arbitrary node.
3. Churn simulation.
4. Output data retrieval on all nodes.
5. Verdict assessment.

3.5.2 FreePastry Test Implementation

Four artifacts implement the distributed test case presented previously: (i) a deployment plan script, for creating the system under test; (ii) a test data file, used as input data; (iii) a test sequence, for driving the system under test into a given state; and (iv) an oracle, for validating the output data.

Listing 2 presents the deployment plan, i. e., a script that starts-up the test architecture. The script has the following behavior: First, the script creates a JavaSE node and deploys a **TestController** component on it. Second, it creates a set of similar nodes and deploys three components on each node: an **UpperTester**, a **DataProvider** and a **SystemMonitor**. Third, it binds the **UpperTester** and the **DataProvider** using a **LocalChannel**. Fourth, it binds the **UpperTester** and the **TestController** using a **SocketChannel**.

Phases two, three and four are repeated 10 times, for creating the whole test architecture. It is important to mention that since the current version of KevScript does not support loop structures, we generate automatically the Deployment Plan.

Figure 3.5 presents the resulting deployment for only two physical nodes (or devices). Kevoree deploys the controller on its own physical node and virtual

Listing 2: Deployment Plan for FreePastry Test

```
1  add node0 : JavaSeNode
2  add controller@node0 : TestController
3
4  add node1 : JavaSeNode
5  add tester1@node1 : UpperTester
6  add provider1@node1 : DataProvider
7  add monitor1@node1 : SystemMonitor
8  add dpc1 : LocalChannel
9  add channel1 : SocketChannel
10
11 bind tester1.provider dpc1
12 bind provider1.provider dpc1
13 bind tester1.tester channel1
14 bind controller.tester channel1
15
16 // (...)
```

machine, along with the test sequence. Upper testers and FreePastry nodes run on the same physical node, but on different virtual machines. Kevoree deploys system monitors and data providers on the same virtual machine as upper testers.

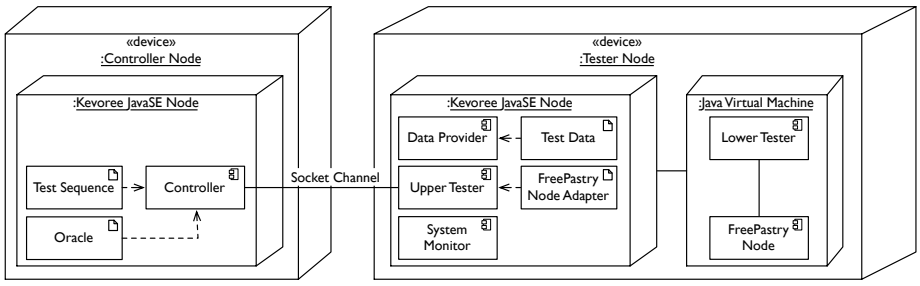


Figure 3.5: UML Diagram Representing FreePastry Test Deployment

The test data contains 1,000 pairs. Keys are a sequence of integers and values are randomly-generated, using the complex data type library, presented in Section 3.3.3. The data provider loads the data and associates it to an action name. When the upper tester asks the data provider for the input data of a given action, the latter returns an array of arguments. The former executes the action as many times as the size of the array.

Listing 3 presents the **PastryTestSequence** class, a simplified Java implementation of the test sequence. This class contains a single method, **execute()**, which receives an array of **Tester** as argument. The **Tester** class is a wrapper, which simplifies the interaction with upper tester components. The method is composed of 5 main steps.

In the first step, the controller sends the **join** message to all available upper testers. The **call()** method is blocking, ensuring the controllability of the test sequence. In the second step, the controller sends the **put** message to the upper tester 5, asking it to use the data provider. In the third step, the controller asks nodes 4, 5 and 6 to leave the system. Then, it asks the remaining nodes to retrieve all the previously-inserted data, using the keys stored in the data

provider. In the fifth step, the controller asks nodes 4, 5 and 6 to rejoin the system, to simulate the churn. In the last step, the controller asks all nodes to retrieve all the previously inserted data again, also using the data provider.

Listing 3: Java Class Representing the FreePastry Test Sequence

```
1  public class PastryTestSequence implements TestSequence {
2
3          @Override
4          public void execute (Tester[] testers) {
5                  Tester[] volatiles = {testers[5], testers[6], testers[7]};
6                  Tester[] stable = {testers[0], testers[1], testers[2],
7                          testers[3], testers[4], testers[8], testers[9]};
8
9                  // Step 1
10                 for(Tester each : testers) {
11                         each.call("join");
12                 }
13
14                 // Step 2
15                 testers[5].callWithProvider("put");
16
17                 // Step 3
18                 for(Tester each : volatiles) {
19                         each.call("leave");
20                 }
21
22                 // Step 4
23                 for(Tester each : stable) {
24                         each.callWithProvider("get");
25                 }
26
27                 // Step 5
28                 for(Tester each : volatiles) {
29                         each.call("join");
30                 }
31
32                 // Step 6
33                 for(Tester each : testers) {
34                         each.callWithProvider("get");
35                 }
36         }
37 }
```

Listing 4 presents the **PastryTestOracle** class, a simplified Java implementation of the FreePastry oracle. This class contains a single method, **execute()**, which receives an array of **DataProvider** as argument and returns a verdict. The **Provider** class, used within the code, is a wrapper used to simplify the access to the distributed data providers.

First, the method retrieves the inserted data from the first data provider and assigns it to the **expected** variable. Then, it retrieves all output data from the first **get** action, from all data providers, compares the output with the expected values, and stores the result. Then, the method repeats the comparison for the second **get** action. Finally, it returns the verdict.

3.5.3 FreePastry Node Adapter Implementation

Listing 5 presents the **NodeAdapter** class, which exposes the interface of FreePastry nodes. This class is composed of three kinds of operations, identifiable by Java method annotations. Methods annotated with **@Setup** and

Listing 4: Java Class Representing the FreePastry Test Oracle

```
1  public class PastryTestOracle implements Oracle {
2
3          @Override
4          public TestResult execute(Provider[] providers) {
5                  Tuple[] expected = providers[0].getArgumentsfor("put");
6                  TestResult verdict = new TestResult();
7
8                  for(Provider each : providers) {
9                          Tuple[] results = each.resultsFor("get", 1);
10
11                         verdict.assertEquals(expected, results);
12                 }
13
14                 for(Provider each : providers) {
15                         Tuple[] results = each.resultsFor("get", 2);
16
17                         verdict.assertEquals(expected, results);
18                 }
19                 return verdict;
20         }
21 }
```

@Cleanup, (**setup()** and **cleanup()**), manage the life-cycle of the node under test. When the upper tester starts (stops) its execution, it calls the **@Setup** (**@Cleanup**) annotated method, which starts (stops) the node under test and prepares (cleans up) its environment.

The third kind of operations are actions, which are annotated with **@Action**. Actions have two main features: they are remotely accessible and send an acknowledge message the controller, after the execution, even when no value is returned. Operations may raise exceptions, which are treated as errors by the controller. The **NodeAdapter** class has four main actions:

1. **join()** makes the FreePastry node join the system and creates its distributed hash table service.
2. **leave()** makes node leave the system.
3. **put()** inserts a pair $\langle key, value \rangle$ in the distributed hash table.
4. **get()** retrieves a value corresponding to a key.

The lower tester has a particular behavior for operations that have a **Future** class as a parameter, which are return values for asynchronous operations. When calling these methods, the lower tester creates an instance of **Future**, uses it to retrieve the return value, and sends this value to the upper tester.

3.5.4 Test Results

This experiment was executed on small networks of machines.

Figure 3.6, the experiment showed that the insert and the retrieve operations of FreePastry behaved correctly on a small network but for a thousand of $\langle key, value \rangle$ pairs.

Listing 5: Java Method Implementing the FreePastry Node Adapter

```
1  @Adapter
2  public class NodeAdapter {
3      private PastryPeer peer;
4      private InetSocketAddress address;
5
6      @Setup
7      public void setup() throws Exception {
8          peer = new PastryPeer(address);
9          peer.createPast();
10     }
11
12     @Cleanup
13     public void cleanup() throws Exception {
14     }
15
16     @Action
17     public void join() throws Exception {
18         peer.join();
19         peer.createPast();
20     }
21
22     @Action
23     public void leave() throws Exception {
24         peer.leave();
25     }
26
27     @Action
28     public void put(String key, String value, Future future)
29                 throws Exception {
30         assert peer != null;
31         peer.put(key, value, future);
32     }
33
34     @Action
35     public void get(String key, Future<String> future)
36                 throws Exception {
37         assert peer != null;
38         return peer.get(key, future);
39     }
40 }
```

The use of Macaw allows testers to create a distributed test case in a modular way, using KevScript to manage the deployment policy, and Java as a test sequence and oracle language. In the presented case study, we obtain a 40 lines of code for the adapter, 37 lines of code to implement the test sequence that drives the deployment of virtual machines and the execution of a distributed scenario, and 21 lines of code to implement the oracle.

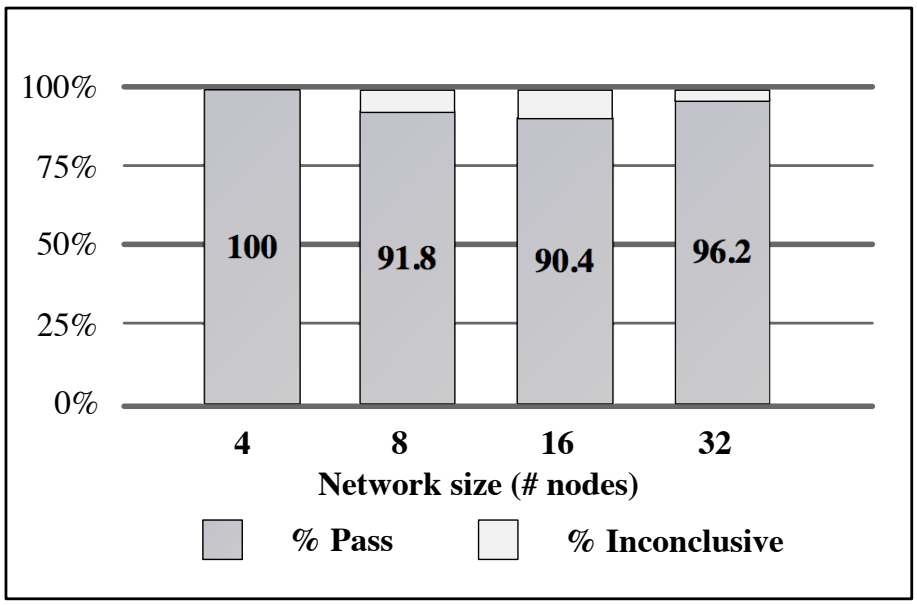


Figure 3.6: Part of Inconclusive Test Conditions

The use of a dynamic component model such as Kevoree [54] does not introduce a major performance overhead. Indeed, the measured average time to initialize a Kevoree runtime and start the JavaSE Node is 1.512 ms, which corresponds to 1 s for the Java Virtual Machine initialization, plus approximately 500 ms to start the node. This is the average value measured for 100 essays on the same machines used for the experiment.

Architectural adaptations of Kevoree nodes consist of receiving a new configuration, comparing it with the running system configuration, and performing the modifications. In Kevoree, the architectural adaptation of a node takes between 30 ms and 300 ms according to the reconfiguration complexity. This value is an approximation obtained in doing adaptations that replace a component by another one, which updates its previous bindings. This delay varies depending on the reconfiguration complexity.

3.6 Conclusion

An important issue when testing large-scale systems is their heterogeneity, which prevents the use of a generic test architecture. The use of component-based models, as well as architectural languages allowing the dynamic configuration

and deployment of components are an interesting approach to deal with this issue.

In this chapter, we presented Macaw, a component-based architecture to test large-scale systems. Macaw was implemented on the top of Kevoree, a framework for developing dynamic distributed software. The main features of Kevoree allows the test architecture to adapt itself in function of the specific requirements of the system under test and the test objectives. These requirements are related to monitoring, logs, data providing, etc.

While Macaw is an interesting experience of building an adaptive test architecture, some more work is needed to improve it. Indeed, distributed test cases and oracles are currently expressed as Java programs. While this approach is pragmatic and works properly, we strongly believe that using domain-specific language is a more elegant approach. Another limit of our architecture is that it addresses experiments where tests have total control of nodes. We intend to study the possibility of deploying test component on runtime non-stop systems.

Chapter 4

A Methodology for Testing Large-Scale Systems

4.1 Introduction

Large-scale systems, such as peer-to-peer, appear as a powerful paradigm to develop scalable distributed systems, as reflected by the increasing number of projects based on this technology ([7]). Among the many aspects of large-scale development, producing systems that work correctly is an obvious target. This is even more critical when large-scale systems are to be widely used. Thus, as for any system, a large-scale system should be tested with respect to its requirements. As for any distributed system, the complexity of message exchanges must be a part of the testing objectives. Testing of distributed systems typically consists of a centralized test architecture composed of a test controller, or coordinator, which synchronizes and coordinates communication (message calls, deadlock detection) and creates the overall verdict from the local verdicts. Local to each node, test sequences or test automata can be executed, which run these partial tests on demand and send their local verdicts to the coordinator. One local tester per node or group of nodes is generated from the testing objectives. Distributed systems are commonly tested using conformance testing [108]. The purpose of conformance testing is to determine to what extent the implementation of a system conforms to its specification. The tester specifies the system using Finite State Machines ([30, 58, 29]) or Labeled Transition Systems ([65, 66, 98]) and uses this specification to generate a test suite that is able to verify (totally or partially) whether each specified transition is correctly implemented. The tester then observes the events sent among the different system nodes and verifies that the sequence of events corresponds to the specification.

In a large-scale system, a node plays the role of an active process with the ability to join or leave the network at any time, either normally (e.g., disconnection) or abnormally (e.g., failure). This ability, which we call volatility, is a major difference with distributed systems. Furthermore, volatility yields the possibility of dynamically modifying the network size and topology, which makes large-scale testing quite different. Thus, the functional behavior of a large-scale system (and

functional flaws) strongly depends on the number of nodes, which impacts the scalability of the system, and their volatility.

As an illustration, Distributed Hash Table (DHT) ([106, 103, 113]) is a basic large-scale system, where each node is responsible for the storage of values corresponding to a range of keys. A DHT has a simple local interface that only provides three operations: value insertion, value retrieval and key look-up. The remote interface is more complex, providing operations for data transfer and maintenance of the routing tables, i. e., the correspondence table between keys and nodes, used to determine which peer is responsible for a given key. Considering the simplicity of the interface, testing a DHT in a stable system is quite simple, but does not provide any confidence in the correctness of implementation for the specific distribution mechanisms. When nodes leave and join the system, the test must check that both the routing table is correctly updated and that requests are correctly routed.

In this chapter, we present a methodology for testing large-scale systems, including testers and coordinator, with the ability to create peers and make them join and leave the system. With this methodology, the test objectives can combine the functional testing of the system with the volatility variations (and also scalability). The correctness of the system can thus be checked based on these three dimensions, i. e., functions, number of peers and volatility. We present an incremental methodology to deal with these dimensions, which aims at covering functions first on a small system and then incrementally addressing the scalability and volatility aspects. Empirical results obtained by running several test cases illustrate the fact that satisfying a simple test criterion such as code coverage is a hard task. Open issues, such as the generation of efficient test objectives are also identified.

The rest of the chapter is organized as follows. The next section introduces the basic concepts and proposes a testing methodology. Section 4.2 presents our methodology for large-scale testing. Section 4.3 describes our validation through implementation and experimentation on an open-source P2P system. Section 4.4 concludes.

4.2 Testing methodology

When testing scalability of a distributed system, the functional aspects are typically not taken into account. The same basic test scenario is simply repeated on a large number of nodes ([45]). The same approach may be used for volatility, but would also lead to test volatility separately from the functional aspect. For a large-scale system, we claim that the functional flaws are strongly related to the scalability and volatility issues.

This because functionalities are specially designed to work with a variable number of nodes (from one up to more than one million) and with the arrival and the departure of nodes. Functionalities do not perform the same in a small system, where each node knows every other nodes, as they perform on large systems, where each node only have a partial view of the whole system. In the last case, the accomplishment of a functionality often leads to more complex communications, such as message routing or node discovery.

When nodes leave or join the system, different functions are performed, other than the update of the routing tables. For instance, in many distributed hash tables, when a node joins the system, it becomes responsible for a range of keys. Thus, before starting to respond to queries, it must receive from other nodes all data associated to these keys. And when the node leaves the system, the inverse transfer of data must be done.

Therefore, it is crucial to combine the scalability and volatility aspects with meaningful test sequences. To take into account the three dimensional aspects of large-scale systems, we present a methodology that combines the functional testing of a system with the variations of the other two aspects. Indeed, we incrementally scale up the SUT either simulating or not volatility. This simulation can be executed with different workloads, such as: shrinking the system, expanding it or both at the same time. These different workloads may exercise different behaviors of the SUT and possibly reveal different flaws.

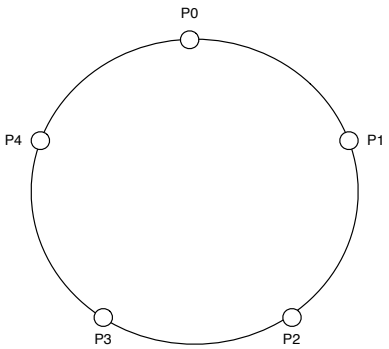
Our incremental methodology is composed by the following steps:

1. small scale application testing without volatility;
2. small scale application testing with volatility;
3. large scale application testing without volatility;
4. large scale application testing with volatility.

Step 1 consists of conformance testing, with a minimum configuration. The goal is to provide a test sequence set efficient enough to reach a predefined test criteria. These test sequences must be parameterized by the number of nodes $TS(P)$, so that they can be extended for large scale testing. Test sequences can also be combined to build a complex test scenario using a test language such as Tela [97].

In our motivating example, we start a stable system with all the nodes set as illustrated in Figure 4.1. The node p_2 will insert some data into a DHT, then the nodes p_3 and p_4 will retrieve them. This first step aims to verify pure functional problems without interference with the size of the system and/or volatility. In the case of a stable and small scale DHT, all the nodes probably know each other representing minimal or even nonexistent routing table updates. Thus, messages may be exchange directly between nodes.

Step 2 consists of reusing the initial test sequences and adding the volatility dimension. The result is a set of test sequences including explicit volatility (TSV). Figure 4.2(a) illustrates a DHT before volatility when data is inserted by node p_2. Then, the nodes p_3 and p_4 join the system and retrieve data as illustrated in Figure 4.2(b). This second step aims to verify functional problems related to volatility at a small scale considering that pure functional problems were isolated at Step 1. Indeed, testing inserts and retrieves upon volatility exercises both data forwarding and routing table update. Furthermore, a small scale system guarantees low forwarding since data tend to be sent to nodes within the routing table.



Order	Peers	Action
1	P2	put(1,'One') put(2,'Two')
2	P3,P4	res1 := retrieve(1) res2 := retrieve(2)

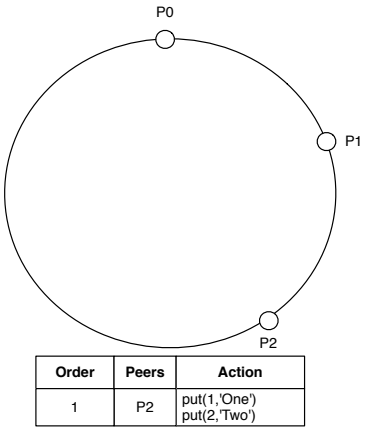
Figure 4.1: Small scale application testing without volatility (Step 1)

Step 3 reuses the initial test sequences of Step 1 combining them to deal with a large number of nodes. We thus obtain a global test scenario GTS. A test scenario composes test sequences. This third step aims to verify functional problems related to scalability. To do so, we test the SUT without volatility in a large scale. As described in Step 1, a stable system represents minimal or even nonexistent routing table updates. Whenever we scale up the SUT, nodes are obligated to perform some tasks like routing messages and forwarding data to unknown peers. Indeed, these tasks could be only tested in large scale systems since nodes are unlikely to know all the others.

Figure 4.3 illustrates a large scale and stable DHT. In our motivating example, node p_2 inserts some data into the DHT respectively at p_1 and p_2. Whenever the nodes p_3 and p_4 try to retrieve data, they probably do not know p_1 and p_2, messages are routed until reach such data. Therefore, aspects related to scalability, such as message routing, can be verified from this third step.

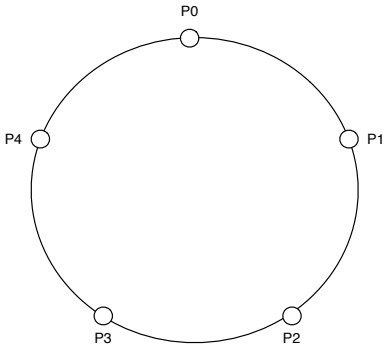
Step 4 reapplies the test scenarios of Step 3 with the test sequences of Step 2, and a global test scenario with volatility ($GTSV$) is built and executed. Figure 4.4 illustrates a large scale DHT upon volatility. In fact, this step aims to verify the problems related to all three dimensions. Therefore, after the insertion of data illustrated in Figure 4.4(a), nodes come and go depending on the type of the volatility. For simplicity, Figure 4.4(b) illustrates the join of new nodes p_3 and p_4. In our example, the successors of both p_3 and p_4 have to update their routing table and route messages. Eventually, the test case can be improved to store something at p_3 or p_4 in order to exercise data forwarding as well.

The advantage of this process is to focus on the generation of relevant test sequences, from a functional point of view, and then reuse these basic test sequences by including volatility and scalability. The test sequences of Step 1 satisfy test criteria (code coverage, interface coverage). When reused at large scale, the test coverage is thus ensured by the way all peers are systematically exercised with these basic test sequences.



Order	Peers	Action
1	P2	put(1,'One') put(2,'Two')

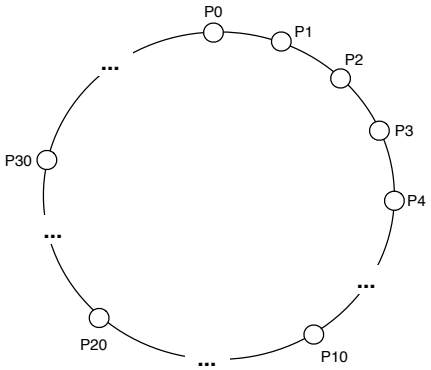
(a) DHT before volatility



Order	Peers	Action
2	P3,P4	res1 := retrieve(1) res2 := retrieve(2)

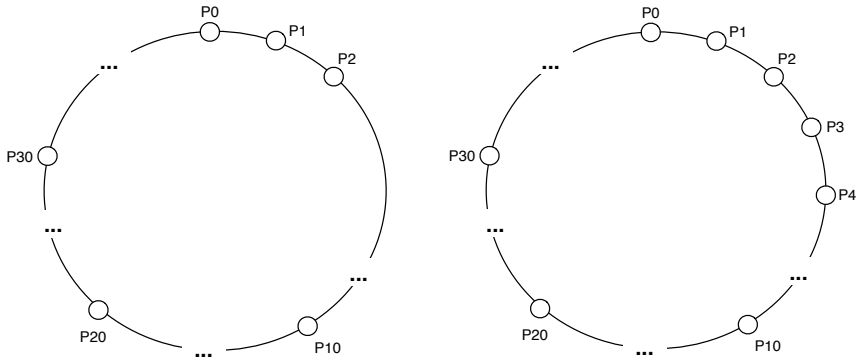
(b) DHT after volatility

Figure 4.2: Small scale application testing with volatility (Step 2)



Order	Peers	Action
1	P2	put(1,'One') put(2,'Two')
2	P3,P4	res1 := retrieve(1) res2 := retrieve(2)

Figure 4.3: Large scale application testing without volatility (Step 3)



Order	Peers	Action
1	P2	put(1,'One') put(2,'Two')

(a) DHT before volatility

Order	Peers	Action
2	P3,P4	res1 := retrieve(1) res2 := retrieve(2)

(b) DHT after volatility

Figure 4.4: Large-scale application testing with volatility (Step 4)

In terms of diagnosis, this methodology allows to determine the nature of the detected erroneous behavior. Indeed, the problem can be linked to a purely functional cause (Step 1), a volatility issue (Step 2), a scalability issue (Step 3) or a combination of these three aspects (Step 4). The most complex errors are the last ones since their analysis is related to a combination of the three aspects. Steps 2 and 4 could also be preceded by two other steps (shrinkage and expansion), to help the diagnosis of errors due to either the unavailability of resources or arrival of new ones. Yet, several rates of volatility can be explored to verify how they affect the functionality aspect of the SUT (e.g., 10% joining, 20% leaving).

Let us illustrate these definitions with a simple distributed test case (see Example 4.2). The aim of this test case is to detect errors on a DHT implementation. More precisely, it verifies whether new nodes are able to retrieve data inserted before their arrival.

[Simple test case]

Action	Testers	Action
(a_1)	0,1,2	join()
(a_2)	2	Insert the string "One" at key 1; Insert the string "Two" at key 2;
(a_3)	3,4	join();
(a_4)	3,4	Retrieve data at key 1; Retrieve data at key 2;
(a_5)	*	leave();
(v_0)	0	Calculate a local verdict;
(v_1)	1	Calculate a local verdict;
(v_2)	2	Calculate a local verdict;

This test case involves five testers $T^\tau = \{t_0 \dots t_4\}$ that control five peers $P = \{p_0 \dots p_4\}$ and five actions $A^\tau = \{a_1^\tau, ..., a_5^\tau\}$. If the data retrieved in a_4 is the same as the one inserted in a_2, then the verdict is *pass*. If the data is not the same, the verdict is $fail$. If t_3 or t_4 are not able to retrieve any data, then the verdict is *inconclusive* (e.g., action timeout). For each tester a local verdict is calculated and send to a test coordinator.

4.3 Experimental Validation

In this section, we present an experimental validation of a popular open-source DHT, FreePastry[1], an implementation of Pastry ([106]) from Rice University. The objective of the experiments is to validate the feasibility of the P2P incremental testing methodology, using a code coverage criteria.

We conducted four experiments, testing FreePastry in different system settings: stable, expanding, shrinking and volatile. These experiments follow steps 1 and

[1] http://freepastry.rice.edu/FreePastry/

2 of our methodology. The goal of the first experiment is to verify that the DHT correctly inserts and retrieves data. The goal of the second experiment is to verify whether nodes that join the system after the insertion of data are able to retrieve this data, i.e., if these nodes integrate correctly the system. Verify the ability of nodes to reconstruct the system when several nodes leave the system is the goal of the third experiment. Finally, the goal of the fourth experiment is to verify whether stable nodes are able to reconstruct the system (and to retrieve the inserted data), when other nodes leave and join the system.

During the experiments, we measured the code coverage to evaluate the impact of the three dimensions (functionality, scalability and volatility) on code coverage, that is, measure to which extent the quantity of inserted data, the system size and the volatility impact on the code coverage. We use a stable system composed of 16 nodes as a reference.

It has to be noticed that the chapter does not focus on how to select the test cases so that they would cover all the code, which is beyond the scope of the chapter. With these four typical scenarios, we want to demonstrate that volatility has an impact on code coverage (i.e., that volatility must be a parameter of a P2P test selection strategy). Additionally, we focus on volatility testing and do not test these systems on more extreme situations such as performing massive inserts and retrieves, or using very large data. Testing different aspects (e.g., concurrence, data transfer, etc.) would increase significantly the confidence on both DHTs. However, these tests were out-of-scope of this chapter. They could be performed through the interface of a single node and would not need the framework presented in this chapter.

For our experiments we use two clusters of 64 machines[2] running GNU/Linux. In the first cluster, each machine has 2 Intel Xeon 2.33GHz dual-core processors. In the second cluster, each machine has 2 AMD Opteron 248 2.2GHz processors. Since we can have full control over these clusters during experimentation, our experiments are reproducible. We allocate equally one node per cluster node. In experiments with up to 64 nodes, we use only one cluster. In all experiments reported in this chapter, each node is configured to run in its own Java VM. The cost of action synchronization is negligible: the execution of an empty action on 2048 nodes requires less than 3 seconds. The execution time and also the synchronization time are out-of-scope of this chapter.

4.3.1 Test Cases Summary

In this section, we describe the test cases used to test the routing table and the DHT. Initially, we describe the test sequences that the test cases are based on. Then, we detail the test cases.

The routing table test sequence

In the routing table test case, testers must analyze the routing table of their nodes to verify if it was correctly updated. More precisely, testers must compare

[2]The clusters are part of the Grid5000 experimental platform: http://www.grid5000.fr/

the ID of nodes from a routing table with the ID of nodes that leave or join the system. This comparison is not trivial, because each tester only knows the ID of its node, which is dynamically assigned. To simplify the analysis of routing tables, we use test case variables to map tester IDs to node IDs, as shown in Section **??**.

We implemented the test case as follows.

Name: Routing Table Test.

Objective: Test the update of the routing table.

Parameters:

- P: the set of nodes that form the SUT;
- P_{init}: the initial set of nodes;
- P_{in}: the set of nodes that join the system during the execution;
- P_{out}: the set of nodes that leave the system during the execution.

Actions:

1. System creation.
2. Volatile nodes are stored in test case variables.
3. Volatility simulation.
4. Routing table verification and verdict assignment.

In the first action, a system is created and joined by all nodes in P_{init}. In the second action, the IDs of P_{in} and/or P_{out} are stored in test case variables. In the third action, volatility is simulated: nodes from P_{in} join the system and/or nodes from P_{out} leave the system by comparing their IDs with the test case variables. In the fourth action, each remaining node ($p \in P_{init} + P_{in} - P_{out}$) verifies its routing table, waiting for κ seconds. Then, the routing table is analyzed whether it has references to the test case variables and a verdict is assigned. Three different test cases were written based on this test sequence:

- **Recovery from node isolation:** The first test case consists in the departure of all nodes that are present in the routing table of a given node p. Then, we test if the routing table of p is updated within a time limit.

  As mentioned, FreePastry uses a lazy approach to update the routing table. Then, we called a *ping* method to force the update of the routing table. We executed this test twice increasing the amount of calls to the *ping* method at each time. In the first time, we called the method just once and FreePastry got an *inconclusive* verdict. Such verdict was assigned since we could not affirm that the routing table was not updated due to the laziness or to a bug. In the second time, we called the method twice within a 1 second delay, then FreePastry got a *pass* verdict.

- **Expanding system:** In the second test case, we test if the nodes that join a stable system are taken into account by the older nodes. To do so, we analyze the routing table of each node that belongs to a set of nodes

P_{init} to test if it is correctly updated within a time limit, after the joining of a set of new nodes P_{in}.

We increased the size of the system exponentially (2^n) up to 1024 nodes[3] to test the update in different system sizes. We set a maximum time to limit the test execution. We also increased this time in exponential scale (2^n), starting from 8 seconds in order to perform at least one update in the routing table. Similar to the node isolation test, FreePastry also got a *pass* verdict. This happened because when a new node joins a FreePastry system, it needs to communicate with all its neighbors inducing the update of their routing tables.

- **Shrinking system:** In this third test case, we test if the nodes that leave a stable system are correctly removed from the routing tables of the remaining nodes, within a time limit.

  We increased exponentially the size of the system and the time limit similarly to the expanding workload. FreePastry got a *pass* verdict in all executions, however, the time to get such verdict increased dramatically compared with the expanding system due to laziness. Differently from the expanding workload, a node does not contact any neighbor when leaving the system. Then, we had to call the *ping* method to force the update of the routing table, otherwise *inconclusive* verdicts were assigned frequently.

The DHT test sequence

Name: DHT Test.

Objective: Test the insert/retrieve operations.

Parameters:

- P: the set of nodes that form the SUT;
- P_{init}: the initial set of nodes;
- P_{in}: the set of nodes that join the system during the execution;
- P_{out}: the set of nodes that leave the system during the execution;
- $Data$ the input data, corresponding to set of pairs (key, value).

Actions:

1. System creation.
2. Insertion of $Data$.
3. Volatility simulation.
4. Data retrieval and verdict assignment.

We describe the DHT test sequence as follows. In the first action, a system is created and joined by all nodes in P_{init}. In the second action, a node $p \in P_{init}$ inserts n pairs. In the third action, volatility is simulated: nodes from P_{in} join

[3] 1024 nodes correspond to 8 nodes per physical node in the clusters.

the system and/or nodes from P_{out} leave the system. In the fourth action, each remaining node ($p \in P_{init} + P_{in} - P_{out}$) tries to retrieve all the inserted data, waiting for κ seconds. When the data retrieval is finished, the retrieved data is compared to the previously inserted data and a verdict is assigned. Four different test cases were written based on this test sequence:

- **Insert/Retrieve in a Stable System:** In this first test case, we configure the system to execute 4 times for different system sizes ($|P| = (16, 32, 64, 128)$). In all executions, no node leaves or joins the system ($P_{in} = \emptyset$, $P_{out} = \emptyset$ and $P_{init} = P$). The same input data is used in all executions ($|Data| = 1,000$). The results show that FreePastry takes at least 16 seconds to get a *pass* verdict for any size of $|P|$.

- **Insert/Retrieve in an Expanding System:** In this second test case, we use a predefined number of nodes ($|P| = 128$) and of input data ($|Data| = 1,000$). The test case uses different configurations, for different rates of nodes joining the system. The rate is set from 10% to 50% ($|P_{init}| \times |P_{in}| = [(116, 12); (103,25); (90,38); (77,51); (64,64)]$). No node leaves the system ($P_{out} = \emptyset$).

  FreePastry takes at least 8 seconds to get a *pass* verdict in an expanding system for any rate of volatility. This is faster than the stable system due to Pastry's join algorithm. Whenever a new node p joins the system it needs to find and contact a successor. Then, Pastry updates the successor list of all the impacted nodes. This update floods a large portion of the system and assists the retrievals.

- **Insert/Retrieve in a Shrinking System:** In this third test case, we also use a predefined number of nodes ($|P| = 128$) and of input data ($|Data| = 1,000$). Initially, all nodes join the system ($P_{init} = P$). After data insertion, some nodes leave the system. The rate of nodes leaving the system was set from 10% to 50% ($|P_{out}| = (12, 25, 38, 51, 64)$). No node joins the system ($P_{in} = \emptyset$). Note that in Pastry, the data stored by a node becomes unavailable when this node leaves the system and remains unavailable until it comes back. Thus, in this test case, we do not expect to retrieve all data, only the remaining data is retrieved to build the verdict.

  The results show that FreePastry takes at least 16 seconds to get a *pass* verdict in a shrinking system for any rate of volatility. This is slower than the expanding one also due to Pastry's algorithm, which is lazy. The update of the successor list only happens when a node tries to contact a successor, for instance, during retrieval.

- **Insert/Retrieve in a Volatile System:** In this fourth test case, we use the same predefined number of nodes and of input data. For this test case, we define a set of stable nodes $P_{stable}, P_{stable} \subset P$ and $P = P_{stable} \cup P_{in} \cup P_{out}$. The rate of stable nodes was set from 90% down to 50% ($|P_{stable}| = (116, 103, 90, 77, 64)$). The initial set of nodes is composed of the stable nodes and the nodes that will leave the system ($P_{init} = P_{stable} \cup P_{out}$). After the data insertion, all nodes from P_{out} leave the system while all nodes from P_{in} join the system. FreePastry also passes this test case, for any rate of stable nodes.

Name	Qualified Name	Sub-packages	Instructions	Description
Past	rice.p2p.past	3	4,606	DHT service
Transport	org.mpisws.p2p.transport	16	19,582	Transport protocol (sockets/messages)
Pastry	rice.pastry	14	26,795	Routing network (join, routing)
Replication	rice.p2p.replication	4	2,429	Object replication

Table 4.1: Main packages summary

4.3.2 Code Coverage

To analyze the impact of volatility and scalability on the different test cases presented above, we conducted several experiments, using the test case presented above, with different parameters. In these experiments, we use two Java code analysis tools for code coverage and code metrics, Emma[4] and Metrics[5], respectively.

According to these tools, FreePastry has 80,897 bytecode instructions and contains 130 packages. About 56 packages are directly concerned by the DHT implementation. The remaining packages deal with behaviors that are not relevant here: tutorials, NAT routing, unit testing, etc. In the code coverage analysis presented in this section, we focus on 4 main packages and their subpackages, which are summarized in Table 4.1. These packages represent the 4 main services affected by our test cases: DHT, data transport, message routing and object replication. In all results presented here, the code coverage rate corresponds to a merge of the code covered by all nodes.

For the first two experiments, we analyze the impact on the code coverage of two parameters, the size of the input data and the number of nodes. As Figure 4.5 shows, the Past package is the most impacted by the growth of the cardinality of the input data, while the impact on the other packages is less significant. The reason for this is that the choice of the node responsible of storing a given data depends on the data key. Thus, when a node stores a large number of data, it must discover the responsible nodes, i. e., use the **lookup()** operation. This operation will behave differently when communicating with known and unknown nodes.

Figure 4.6 shows that the code coverage of the four packages grows when the system scales up. The explanation for this is that in small systems (e.g., 16 nodes), nodes know each other, and messages are not routed. When the system expands up to 128 nodes, each node only knows part of the system, making communication more complex. However, there is a limit on the coverage gains, while scaling up from 128 nodes to 256 nodes. Such limitation is due to some specific portions of the code (e.g., exceptions) that can be covered only by specific test cases.

In the other experiments, we analyze the impact of volatility on the code cov-

[4]http://emma.sourceforge.net
[5]http://metrics.sourceforge.net

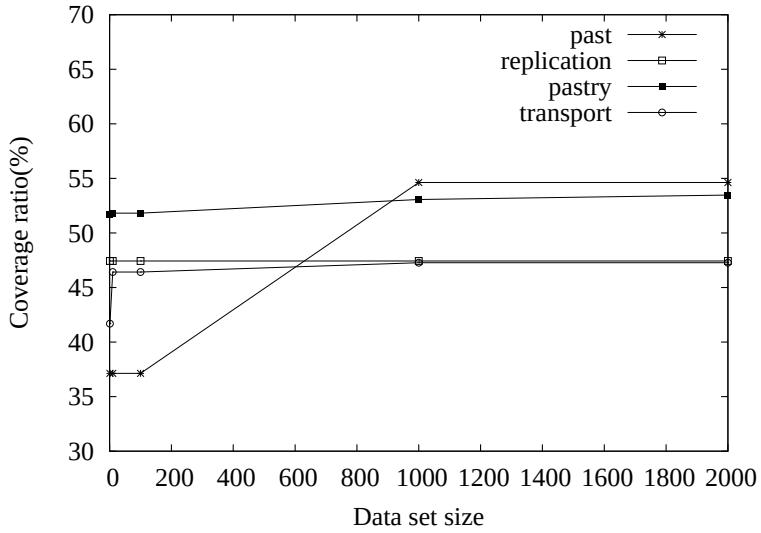


Figure 4.5: Coverage on a 16-node stable system

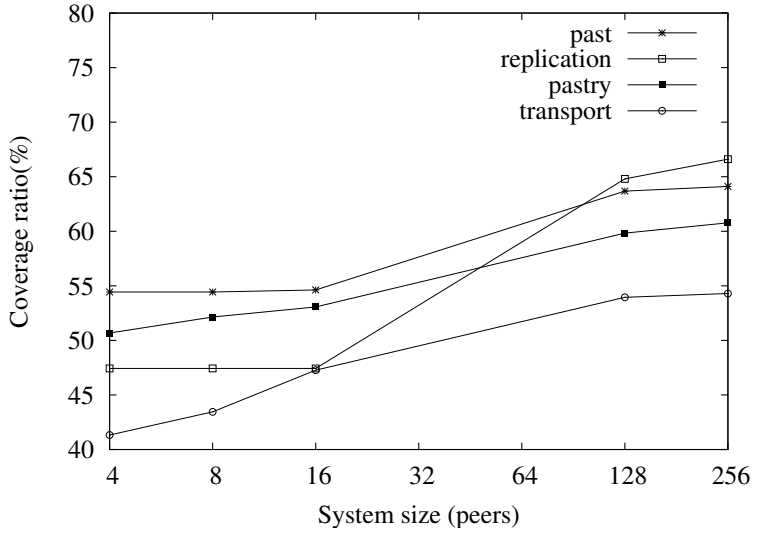


Figure 4.6: Coverage inserting 1000 pairs

erage, using the DHT test cases. We compare these results with the coverage of the 14 original unit tests provided with FreePastry (Figure 4.8), which are executed locally. Figure 4.7 presents a synopsis of the different code coverage results. As expected, our test cases cover more code than the original unit tests, especially on packages that implement the communication protocol.

At first glance, volatility seems to have a minor impact on code coverage, since the stable test case with 256 nodes yields better results than some other test cases (e.g., shrinking 128). In fact, the impact is significant because the different test cases exercises different parts of the code and are complementary. This complementarity is noticeable for the Pastry and the Past packages, where the accumulated results are better than any other result as illustrated in Figure 4.8. The total accumulated coverage (Accum.+Original unit tests) shows that our tests cases and the original unit tests are also complementary.

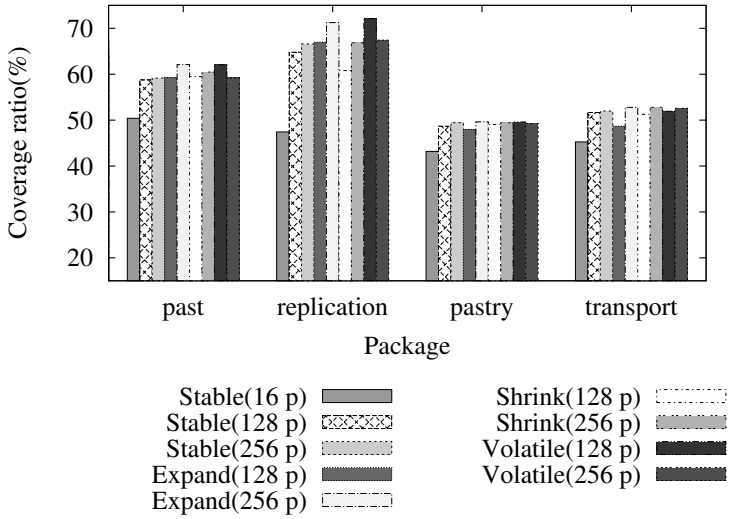


Figure 4.7: Coverage by package (our test cases)

4.3.3 Learned Lessons

As expected, volatility increases code coverage. However, such increase has a limit due to some specific portions of the code (e.g., exceptions) that can be covered only by specific test cases. For instance, a test case that covers the exception threw by a look-up performed with the address of a bogus node. This situation only happens when a node address resides in the routing table after its volatility.

Other DHTs, such as Chord ([113]) or CAN ([103]), have similar behavior to FreePastry for data storage and message routing. Therefore, a similar impact on code coverage of the size of the system and the number of data should be expected.

In spite of the test cases simplicity, the ratio of code covered by all test cases is

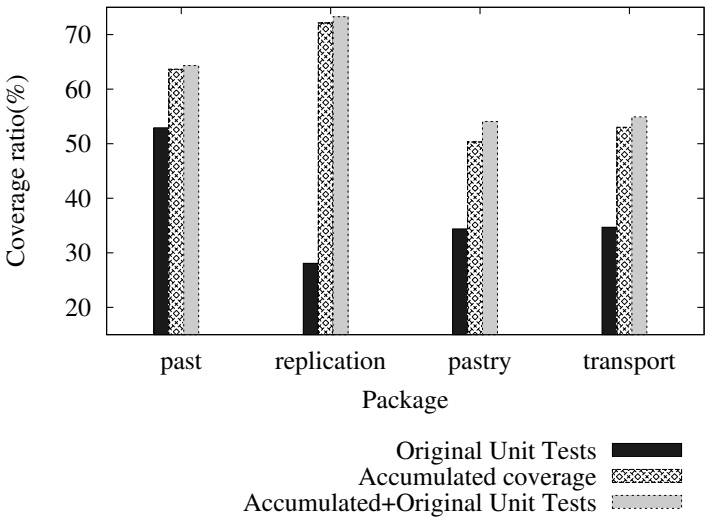


Figure 4.8: Coverage by package

rather important. While the impact of volatility, the number of nodes and the amount of input data on the code coverage are noticeable, the only variation of these parameters is not sufficient to improve code coverage on some packages, for instance, the transport package. A possible solution to improve the coverage of these packages is to alter some execution parameters from the FreePastry configuration file. Most of the parameters deal with communication timeouts and thread delays. Yet, the number of parameters ($\approx$ 186) may lead to an unmanageable number of test cases.

4.4 Conclusion

In this chapter, we presented a testing methodology that considers the three dimensional aspects of P2P systems: functionality, scalability and volatility. We used this methodology to conduct an extensive experimental validation using FreePastry, a popular open-source DHT, on different test scenarios.

We coupled the experiments with an analysis of code coverage, showing that the alteration of the three dimensional aspects improves code coverage, thus improving the confidence on test cases.

The next challenging issue is to propose a solution to select scenarios that guarantees the functional coverage of the P2P functions in combination with the "coverage" of volatility/scalability. Such a multidimensional coverage notion should be defined properly as an extension of existing classical coverage criteria.

Chapter 5

Model-Based Oracle

5.1 Introduction

As stated in Chapter 2, the scale of the system under test affects several test components, such as: test controllability [6], fault-injection [71], logging facilities, and oracle calculation, among others. In the precise case of the oracle, the validation of global properties becomes a major problem, as they depend on values that are spread throughout the system.

This problem is faced when testing, for instance, the global correctness of the routing algorithm of a P2P system. In these systems, efficient message routing depends on the correct state of local routing tables, which must be maintained frequently, according to the dynamic state of the system: arbitrary network latencies, node failures, and churn. Hence, the actual content of a routing table is nondeterministic and highly dynamic, making it non obvious to tell whether it is correct at any given point in time. The global correctness of the routing algorithm depends on the (volatile) content of the routing tables in each local node that is very hard to aggregate into a global view in a timely and scalable way. Furthermore, the correctness of the global view can only be verified at given states: it may be invalid right after churn, but must be valid after a certain delay in a stable state.

A typical approach for testing such a feature in a distributed system consists of a centralized controller and several testers, each one controlling a single port or node interface [121]. The tester is the application that runs in the same logical devices as system nodes, and controls their execution and their volatility, making them leave and join the system at any time, according to the needs of a test. The controller sends the test inputs, controls the synchronization of the test case execution and receives the outputs (or local verdicts) from each tester [38]. However, building a global verdict from the information gathered locally can be a very difficult problem. For instance, in a system where each node maintains a set of references to its physically closest neighbors (e. g., Pastry [106]), the only way to validate the correct construction of the system would be to first gather information from all nodes, then calculate the distance between them, and finally check if the contents of the reference set are actually the closest

neighbors. A similar problem arises when verifying load balance on MapReduce systems, which distribute their load burden across their nodes, including storage, query processing, and computations. To verify their algorithm of load-balancing, one must gather information from all nodes, which can be a large amount of data, and check for system usage information (e. g., partitioning of datasets).

In this Chapter, we present an approach leveraging the idea of model at runtime [87] to provide a dynamically built oracle for testing properties in large-scale distributed systems. This approach focuses on global, liveness, observable and controllable properties. More precisely, it focuses on a particular class of properties that cannot be calculated by a single node or by a portion of the system; that are eventually true; that are observable from the system interface; and that respond to external events. We propose to efficiently keep updating a global model of the system during its execution. This model is then instantiated and evolved at runtime, by monitoring the corresponding distributed system, and serve as oracle for distributed tests. On the implementation side, we show that standard Model-Driven Engineering (MDE) technology such as Kermeta [88] can be used to easily implement the oracle part of such model-based distributed tests. We use this approach to test topology-related properties on two open-source, structured P2P systems. This approach extends the architecture presented in Chapter 3. The addition of a global model and an efficient update mechanism allows also to test global properties.

The rest of the Chapter is organized as follows. The next section presents a real world motivation case. Section 5.2 introduces some fundamental concepts in large-scale distributed systems and some global topology properties these systems must satisfy. Section 5.3 presents our approach to represent and check these properties, as well as our architecture for testing distributed systems. Section 5, describes our validation through implementation and experimentation on two open-source systems. Section 7 concludes.

5.2 Background

5.2.1 Routing Tables

In a large-scale distributed system, nodes have a partial view of the system, i. e., their routing tables keep only a subset of other node addresses. The choice to build the routing table is then crucial for the performance of the whole system. There are as many routing algorithms as there are different systems. In some data sharing systems (e. g., Gnutella, Kazaa), the routing table is built randomly, each node keeps a set of nodes that represent some interest. In structured systems, the routing table is built systematically, each node keeps a set of nodes with the specific ID that are needed for efficient routing.

Currently, there are two major ways to maintain the routing table: actively and lazily. In the former, each node periodically pings all its neighbors and drops the unavailable nodes (e. g., Chord [113]). In the latter, extra status information is added to messages exchanges and the unavailability of a node is only noticed when one of its neighbors does not answer a given query. Some

systems (e. g., Bamboo [104], Pastry [106]) use both approaches to maintain their routing tables.

In Pastry, the routing table is divided into three parts. The first one, named *leaf set* contains all nodes having numerically close node ID (i. e., ID that share the same prefix). The second one, the actual *routing table*, contains nodes with different prefixes (at least one node for each element of the prefix domain). The third one, the *neighborhood set*, contains the physically closest nodes, independently from their ID. Pastry uses both, lazy and active approaches to update the routing table. While the leaf and the neighborhood sets are maintained actively, the actual routing table is only updated when a node communicates with its neighbors.

In Chord, each node maintains a routing table with at most m entries, where 2^m is the maximum size of the system. The i^{th} entry in the table at node n contains the identity of the node, s, that succeeds n by at least 2^{i-1} on the identifier circle, i. e., $s = successor(n+2^{i-1})$ where $1 \leq i \leq m$. When the ID is not taken, the entry is the node with the following ID. Chord uses an active approach to update its routing table, periodically running a process called "stabilization".

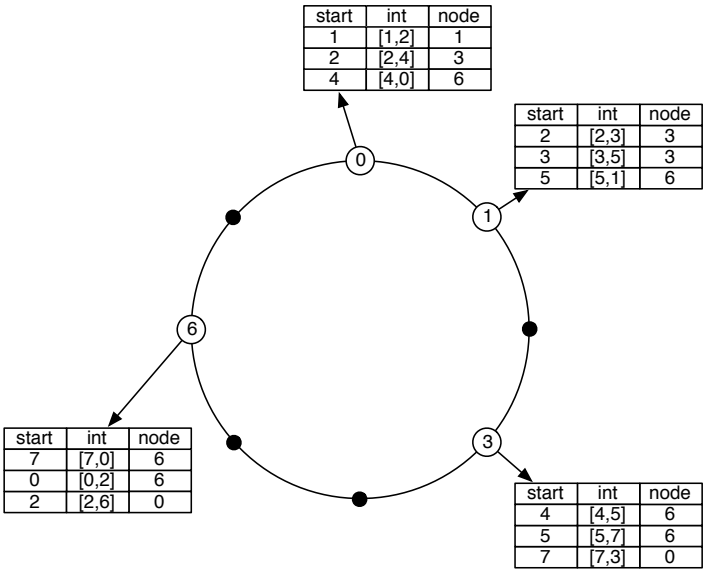


Figure 5.1: Chord routing

For instance, in a Chord system with $m = 3$ (Figure 5.1), containing nodes n_0, n_1, n_3, and n_6, the routing table of n_0 stores the addresses nodes n_1, n_3 and n_6. When a new node n_4 joins the system, then n_0 will update its routing table and replace the address of n_6 with n_4. The routing tables of both systems, Pastry and Chord, are the subject of the experiments presented in Section 5.4.

5.2.2 Properties

As any other distributed system, large-scale ones must satisfy the properties of safety and liveness [75]. Safety properties are predicates that should always be true, ensuring that the system never reaches an unacceptable state. For instance, a Chord routing table must never have a null entry: in a system composed of only one node, all entries in the routing table must point to the node itself.

Liveness properties are predicates that should eventually be true. For instance, in Chord, the routing table of a node is valid until another node has joined the system. At this precise state, table entries are and will remain invalid during a *stabilization* time, i. e., the time necessary to detect the presence of a new node and the consequent routing table update. In real life execution, with frequent churn, these properties may never be established.

We list below several topology-related liveness properties, which are tested in Section 5.4.

Definition 6 *Let S be a large-scale distributed system, $Nodes^S$ the nodes composing this system and N the size of the system in terms of number of nodes.*

Definition 7 *The topology of S can be represented by a directed graph, where each node $n \in Nodes$ is a vertex and every entry in its routing table is an edge to a neighbor.*

The first property, which is common to all systems, is the connectivity of the system.

Property 1 (Connectivity) *S is a strongly connected graph.*

The second property, which is common to several DHT algorithms [125] (e. g., Pastry, Tapestry, Chord, Kademlia), concerns the diameter of the system, which should be $O(logN)$. It is important to note that this property is too lazy for $O(1)$ DHTs [111] and should be strengthened.

Property 2 (Diameter) *The maximum eccentricity over all vertices of S is $O(logN)$.*

While these two first properties are rather simple to verify, if one has a centralized model of the topology, they are essential to test. If they are not respected, the accuracy of the tests presented in Sections 5.4.4, 5.4.5 and 5.4.6, which verify the correct update of the routing table upon churn, would be compromised. Functional tests, such as the correctness of message routing and of data insertion, would not be reliable. The third property concerns the self-organization of DHTs.

Property 3 (Self-organization) *When a node joins (or leaves) the system, the total cost to update the routing tables, in terms of the number of messages exchanged, is $O((logN)^2)$ messages.*

The fourth property is the ability to handle churn and remain working properly. Some routing algorithms are liable to fractionate the structure upon churn preventing fractions to send messages to one another. Some solutions are provided to merge fractions [56], but not all the implementations do that.

Property 4 (Self-healing) *S remains strongly connected upon churn.*

The properties defined above are related to the topology of peer-to-peer systems. There are two additional properties that are not related to the topology of the system and therefore validation is left for future work, but yet need a global view of the system to be verified: load balance and elasticity. Load balancing is the ability to distribute the workload across the nodes of the system for scalability. For instance, MapReduce, distributed database management systems, and P2P systems distribute their load burden across their nodes. In the particular case of P2P, load balance may rely on consistent hashing algorithms (for structured systems) or on satisfaction load balance algorithms. A possible approach to verify the *Load Balancing* property is to gather information from all the nodes of the system, which can be a large amount of data, and then check the consumption of computing resources.

The elasticity property is the ability of a system to add or remove resources at a fine grain and with a small lead time [12]. Elasticity is ensured either manually or automatically, through the interaction with the infrastructure provider of a cloud system. A possible approach to verify this property is to vary the load of the system and observe the system behavior. While the load varies, we expect the allocation, or decommissioning of failed or surplus nodes.

Unstructured systems, which rely on gossiping protocols are also an interesting class of system for testing global properties, other than the Connectivity presented above. Some examples of these properties are [13]: the efficiency of message propagation, message coverage, message delay, degree distribution (number of neighbors by node), clustering coefficient and the reliability under churn. However, the verification of these properties is part of future work.

5.2.3 Kermeta

Kermeta is a MDE workbench for building rich development environments around meta-models using an aspect-oriented paradigm [89, 69]. It has been designed to easily extend meta-models with many different concerns (such as syntactic correctness including context information, execution information, model transformations, tracing information, connection to concrete syntax, etc.) expressed in heterogeneous languages. A meta-language such as the Meta Object Facility (MOF) standard [92] indeed already supports an object-oriented definition of meta-models in terms of packages, classes, properties, and operation signatures. However, MOF does not include concepts for the definition of constraints or operational semantics (MOF only contains operations signatures). Kermeta can thus be seen as an extension of MOF with a language for specifying constraints and operation bodies at the meta-model level.

The action language of Kermeta is especially designed to process models. It is imperative and includes classical control structures such as blocks, conditional

and loops. It implements traditional object-oriented mechanisms for multiple inheritance and behavior redefinition with a late binding semantics. It is statically typed, with generics and provides reflection as well as an exception handling mechanism.

In addition to object-oriented structures, the MOF contains model-specific constructions such as containment and associations between classes. These elements require a specific semantics of the action languages in order to maintain their integrity. For instance, the assignment of a property must handle the other end of the association if the property is part of an association and the object containers if the property is a composition.

Kermeta expressions are a superset of the Object Constraint Language (OCL) ones and have a close syntax. In particular, they include operations similar to OCL iterators on collections such as *each, collect, select* or *detect*. The standard framework of Kermeta also includes all the operations defined in the OCL standard framework. This alignment between Kermeta and OCL allows OCL constraints to be directly imported and evaluated in Kermeta. Classes define invariants and operations define pre- and post-conditions. The Kermeta virtual machine has a specific execution mode, which monitors these contracts and reports any violation.

5.2.4 Models at Runtime

Models at runtime [22] are formal representations of the system which support computer-based processing, unlike most models commonly used in analysis and design. As stated by Bran Selic:

> 'This enables formal coupling between models and the systems they represent, similar to the relationship that exists between a program written in a high-level programming language and its machine code counterpart'(pp. 26).

In our case, we use models at runtime as a "live" oracle within a test architecture to check properties during the execution of a test sequence. Since we are not able to directly observe the current state of the distributed system, we take snapshots of its nodes periodically, aggregate the information and update the model. Since we focus on liveness properties, we do not need to analyze all states of the system but only given states, after exercising the software interface of one or more nodes.

5.3 Testing Global Liveness Properties

In this section, we present our approach for testing global liveness properties on large-scale distributed systems. After presenting our test approach, we introduce test cases, explain their implementation, and discuss the limits of the approach.

We consider a test case as a pair $\langle \mathcal{TS}, \mathcal{O} \rangle$, where $\mathcal{TS}$ is a test sequence, i. e., a sequence of steps that drives the System Under Test (SUT) into a given state,

and $\mathcal{O}$ is the oracle, which reads the output generated by the SUT and provides a verdict.

5.3.1 Testing Approach

There are two major approaches for verifying global properties of a distributed system. The first one is to keep the output data locally in each node during the test execution and to perform a *post-mortem* analysis. This approach is less intrusive since less test data is exchanged during the execution. The second one, which we have adopted, is to perform a *live* analysis of the outputs. While this approach is more intrusive, since the exchange of test data may perturb the network performance, it is also more flexible. It allows tests to adapt themselves according to the output. This is particularly valuable when verifying liveness properties and the duration of the verification is nondeterministic.

The rationale is to gather the output data on a single node, build a centralized model of the system and verify global liveness properties on this model. Once a property is verified the execution can be stopped. An important issue of this approach is finding an update frequency that is adapted to the property verification.

5.3.2 Global Model and Property Specification

As stated in Section 5.2.2, several distributed hash tables share common properties. However, the diversity of routing and updating algorithms complicates the writing of tests to verify these properties through different implementations. The complexity can be reduced if the oracle is specified on a more abstract level, allowing tests to ignore implementation details such as the nature of identifiers, the data structure used to store the routing table, and method signatures.

Figure 5.2 presents a model of the topology of distributed systems. The model is simple, yet sufficient to verify the properties presented in Section 5.2.2. The classes `System` and `Node` are connected by two disjoint associations, `available` (the nodes that joined the SUT) and `unavailable` (the nodes that left the SUT). Each node has a set of neighbors. The model also contains invariants that prevent nodes from been available and unavailable at the same time and from being part of its own neighborhood.

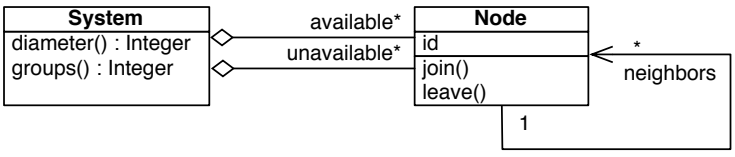


Figure 5.2: UML Class Diagram Describing System Properties

The `System` class contains two operations, **diameter()** and **groups()**, which calculate the diameter of a graph and the number of independent graphs, respectively. This model usually must be modified for testing a specific system or different properties. For instance, if one wants to test a load-balancing property,

mentioned in Section 5.2.2, several new attributes (e. g., CPU load, memory usage, etc.) would be needed and the current associations would be useless.

Once the model and its invariants are done, the problem is to create and update a model instance during the execution. The Oracle component, which is deployed along with the test controller updates and validates the global model. Thus, test sequences can directly access the global model. When a test sequence asks a tester to create a node, it also creates a new instance of `Node` and links it to the only instance of `System` through the `unavailable` association in the global model.

With this model, we can use OCL to specify the strong connectivity property in Listings 6 and 7.

```
context Node
def allNeighbors : Set(Node) =
   self.neighbors→union(
      self.neighbors.allNeighbors())→asSet()
```

Listing 6: All Neighbors Operation

```
context System
inv:
   self. available→forAll(a,b : Node |
      a <>b implies a.allNeighbors→includes(b))
```

Listing 7: Connectivity Invariant

The scalability property is specified in Listings 8 and 9. The first specifies an operation that returns all possible paths between two nodes and the second ensures that for each node of the system, there is a path to all other nodes in at most $\log_2(\mathcal{N})$ steps, where $\mathcal{N}$ is the size of the system.

```
context Node
def allPaths( visited : Sequence(Node), to : Node) :
   Set(Sequence(Node)) =
   if self = to
   then
      Set{}→including(visited→including(self))
   else
      self.neighbors→collect(each : Node | node →
allPaths( visited →including(self), to))
```

Listing 8: All Paths Operation

The model also contains operations allowing the individual control of nodes. The goal is to allow the creation of test scenarios, besides the dynamic oracle. These operations must be *glued* to the SUT through adapters.

```
context System
inv:
   self.available→forAll(a,b : Node |
      a <>b implies a.allPaths(Sequence{}, b) →
exists(each : Sequence(Node)→size() ≤ self.available→size().log2()))
```

Listing 9: Diameter Invariant

5.3.3 Implementation

As stated in Chapter 3 the current version [36] of the test architecture is implemented in Java (version 1.5), global test sequences and adapters are implemented as Java methods. Meta-information about test steps (e. g., the subset of testers that should execute a step, timeout, etc.) are described as Java annotations. Listings 11 and 12 present examples of a node adapter and a test sequence, respectively.

Besides exposing nodes' interfaces, adapters also describe a method for updating the global model. When a tester receives an update request, it queries the node it controls, computes the differences with the previously sent information and sends these differences to the test controller.

We use Kermeta (version 1.4) to implement the oracle part of the test, i. e., the runtime model and the verification methods: **connectivity()** and **diameter()**. Since Kermeta is interoperable with Java (it compiles to Java bytecode), its integration with the test controller is flawless. The former is implemented using a depth-first search algorithm and the latter is implemented using the Floyd-Warshall algorithm [51] for the "all pairs shortest-path problem". Listing 10 presents the implementation of this algorithm.

```
aspect class System {
   operation diameter() : Integer is do
      var distances : Sequence<Matrix<Integer>> init Sequence<Matrix<Integer>>.new
      var size : Integer init nodes.size()

      from var i : Integer init 0
      until i == size
      loop
         distances.add(Matrix<Integer>.new.initialize(size,size))
      end

      distances.elementAt(0). fill (Integer.MAX_VALUE)
      var index : Integer init 0

      nodes.each{node |
         node.index := index
         index := index + 1}

      nodes.each{node |
         node.neighbors.each{neighbor |
            distances.elementAt(0).set(node.index, neighbor.index, 1)}}

      from var k : Integer init 1
```

```
    until k == size
    loop
      from var i : Integer init 0
      until i == size
      loop
        from var j : Integer init 0
        until j == size
        loop
          distances.elementAt(k).set(i,j, distances.elementAt(k-1).get(i,j).min(
            distances.elementAt(k-1).get(i,k) + distances.elementAt(k-1).get(k,j)))
          j := j + 1
        end
        i := i + 1
      end
      k := k + 1
    end

    result := 0

    from var i : Integer init 0
    until i == size
    loop
      from var j : Integer init 0
      until j == size
      loop
        result := result.max(distances.elementAt(size - 1).get(row, col))
        j := j + 1
      end
      i := i + 1
    end
  end
}
```

Listing 10: Floyd-Warshall Algorithm Implementation in Kermeta

5.3.4 Discussion

When we started the development of our experiments, we intended to use OCL to implement the oracle. Indeed, the declarative nature of OCL simplifies the specification of global properties. However, our first attempts to evaluate OCL expressions on models with several hundreds of nodes showed poor performance, which lead us to use an imperative language instead.

In our approach, we separate the oracle, test sequences, and node adapters, allowing these three parts to evolve independently. Adapters depend strongly on system node interface and must be rewritten when testing different systems. Different test sequences and oracles that test the same system share the same adapters. Test sequences and oracles depend on adapters, and can be reused for testing different systems, if adapters provide the same interface. As the global model evolves, allowing the representation of new information and hence the verification of additional properties, testers must collect more information. This implies changes in the adapter, which performs additional queries on the

SUT, and also in the message aggregation function. The latter is only needed if the information can be combined and reduced.

The high level of abstraction of model-based tools eases the representation and the validation of global properties. These tools ensure that the models representing the oracle data are sound (with respect to their meta-model) during and after the execution of test sequences. A possible limit of this kind of tools concerns the size of the models: most model-based tools are based on the Eclipse Modeling Framework (EMF), which is not adapted to deal with large models [55].

Our approach focus on specific classes of properties: global, liveness, observable, and controllable properties. It is not adapted to verify local properties, which do not require a global view of the system. It is not adapted to verify safety properties either, since they require an analysis of all the historical states of the system. Since we do not instrument the SUT, we cannot verify properties that are not observable from the public interfaces of the system and that do not respond to external events.

In the current implementation of the architecture, testers can force nodes to end their execution either normally or abnormally. This allows test sequence to inject "macro-level" faults and implement scenarios that are not interested in the origins of a failure. However, the architecture cannot inject specific faults, e. g., disk, network, bugs. We intend to combine the architecture with fault-injection tools [61] to overcome this limitation.

Another limitation of the current architecture concerns the reproducibility of tests, i. e., it does not provide repeatable automated tests [21]. In our experiments, we relied on the Grid5000 infrastructure to ensure the use of the same environment for different executions. The use of an automated staging system with support to large-scale environments (e. g., Weevil [123] and Mulini [73]) to deploy and execute tests can overcome this limitation.

5.4 Experimental Validation

In this section, we present an experimental validation of our approach. Our objective is to validate the correct implementation and the robustness of two popular open-source DHTs with respect to the properties presented in Section 5.2.2: FreePastry[1] and OpenChord[2]. FreePastry is a Java implementation of the Pastry algorithm, developed by the Rice University. It has 540 classes and 89 interfaces, organized in 90 packages, for a total of 50 875 lines of code. OpenChord is an implementation of the Chord algorithm, developed by the Bamberg University. It has 96 classes and 11 interfaces, organized in 13 packages, for a total of 9245 lines of Java code.

These experiments complete those presented in Chapters 3 and 4, where we used these same implementations to test the functionality of their DHTs (data insertion and retrieval). These former experiments showed us that while some properties could be verified locally (at each node), some others could only be

[1] `http://freepastry.rice.edu/FreePastry/`
[2] `http://open-chord.sourceforge.net/`

verified in a centralized manner. For our experiments, we use an incremental test methodology [38] that copes with both volatility and scalability aspects of large-scale distributed systems. The main goal of this methodology is to simplify diagnosis: tests sequences start with a small-scale system and increases the number of nodes after each execution. Node volatility is also introduced incrementally: the test sequence starts with a stable system, then with a growing system, a shrinking system, and finally with a complete volatile system. We organized the experiments in the following test scenarios:

1. **Bootstrapping**: checks the ability of the SUT to build a connected (Property 1) and efficient (Property 2) system.

2. **Node isolation**: checks the ability of a node to find new neighbors, after the departure of all its neighbors (Properties 3 and 4).

3. **Expanding system**: checks the ability of nodes to update their routing tables when new nodes join the system (Properties 3 and 4).

4. **Shrinking system**: checks the ability of nodes to update their routing tables when nodes leave the system (Properties 3 and 4).

During the experiments, we used two clusters of 64 nodes running GNU/Linux.[3] In the first cluster, each node has 2 Intel Xeon 2.33 GHz dual-core processors. In the second cluster, each node has 2 AMD Opteron 248 2.2 GHz processors. Since we have full control over these clusters (nodes and network routers) during experimentation, our experiments are reproducible. The implementation and tests, produced for this paper and other P2P applications, are available on our web page.[4] We allocate the logical nodes equally through the nodes in the clusters up to 8 logical per physical node. In experiments with up to 64 logical nodes, we use only one cluster. In all experiments reported in this paper, each logical node is configured to run in its own Java VM. Execution configurations, including network, disks, DNS server, node reservation and their usage, are ensured by the OAR2 software deployed on the Grid5000 architecture.[5]

5.4.1 Global Model Extension

Figure 5.3 presents an extension of the topology model introduced in Section 5.3.2. Here, the main superclasses `System` and `Node` have both two subclasses, which are specific to Pastry and Chord. These subclasses allow the specification of properties that only apply to these systems. For instance, we can specify that the Chord ring, built using the **successors** association, should only have one cycle. We can also specify that the Pastry nodes, connected through the **neighborhood** association are actually the physically closest nodes. This model was used to implement the test sequence presented in Section 5.4.

[3]The clusters are part of the Grid5000 platform: `http://www.grid5000.fr/`
[4]Peerunit project, `http://peerunit.gforge.inria.fr`
[5]http://www.grid5000.fr/mediawiki/index.php/OAR2

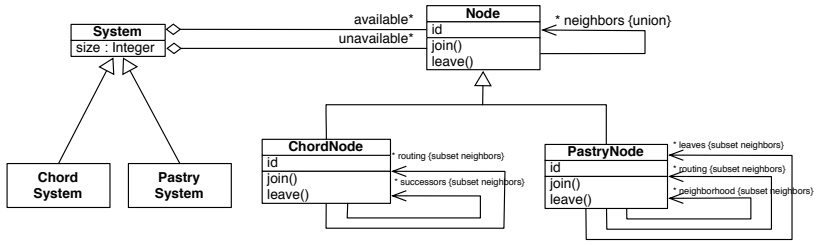


Figure 5.3: UML Class Diagram Representing Chord and Pastry Properties

5.4.2 Adapter and Test Sequence Implementation

The experiments use two adapters, one for each SUT, and three test sequences, one for each test scenario. Listing 11 presents the implementation of the FreePastry adapter. This adapter has five test steps, i. e., methods decorated with the **@TestStep** annotation. The testers use these test steps to start the bootstrap node, start a node, update the global model and quit the system. For instance, when a tester receives the message **start**, it instantiates a peer and calls successively two methods: **join()** and **createPast()**. The execution is bounded by a time constraint to last less than 10 000 ms, otherwise the tester aborts the execution and notifies the controller. We developed a similar adapter for OpenChord.

Algorithm 2 presents an example of a global test sequence. It specifies the test sequence presented in Section 5.4.3, which validates the bootstrapping process. The global test sequence creates a system with $\mathcal{N}$ nodes, waits for system stabilization and then verifies that all nodes belong to the same system (Property 1) and that the diameter of the system is $O(\log\ \mathcal{N})$ (Property 2). Different scenarios execute this test sequence, with $\mathcal{N}$ increasing exponentially from 16 up to 256 nodes.

The global test sequence calls two operations, defined in the global model: **diameter()** and **groups().** They calculate the diameter of a graph and the number of independent graphs, respectively. The global test sequence interacts with the system nodes through the use of two messages, **start** and **bootstrap**, defined by adapters.

Listing 12 presents the Java implementation of the bootstrapping process test. For the sake of simplicity, some parts of the code were omitted. Calls to the **execute()** method actually calls test steps on the testers side.

5.4.3 Bootstrapping

The first test concerns the bootstrapping process, or how a new node joins the system. In some implementations, e. g., FreePastry, OpenChord, Plan-X[6], when a node wants to join the system, it must first contact a *bootstrap* node (i. e., bootstrapper), which will help it to get an *ID* and contact the rest of the

[6] `http://www.thomas.ambus.dk/plan-x/routing/`

```
public class RoutingTableTest {
    private RemoteModel remoteModel;
    // Bootstrapper address:
    private String HOST;
    private String PORT;

    @TestStep(timeout = 40000)
    public void bootstrap() throws Exception {
        InetSocketAddress address = new InetSocketAddress(HOST, PORT);
        peer = new PastryPeer(address);
        peer.bootstrap();
        peer.createPast();
        //Store global variable with bootstrapper address:
        this.put("bootstrap", address);
    }
    @TestStep(timeout = 10000)
    public void start() throws Exception {
        //Delay to avoid bootstrap error:
        Thread.sleep(this.id() * 100);
        //Retrieve bootstrapper address:
        InetSocketAddress address = (InetSocketAddress) this.get("bootstrap");
        peer = new PastryPeer(address);
        peer.join();
        peer.createPast();
    }
    @TestStep(timeout = 2000)
    public void updateModel() throws RemoteException {
        this.push(new NodeUpdate(peer.getId(), peer.getRoutingTable()));)
    }
    @TestStep(timeout = 3000)
    public void quit() throws RemoteException, NotBoundException {
        peer.leave();
    }
}
```

Listing 11: FreePastry Node Adapter (simplified)

Algorithm 2: Global Test Sequence: Bootstrapping Test

Input:
$\mathcal{S}$: a set of nodes;
n_b: the bootstrapper node;
limit: maximum number of checks;
delay: time between checks
begin
  $tries \longleftarrow 0$;
  send *bootstrap* **to** n_b;
  send *start* **to** $\mathcal{N}$;
  wait *stabilization*;
  repeat
    $tries \longleftarrow tries + 1$;
    wait *delay*;
    log $tries, groups(\mathcal{N})$;
  until $groups(\mathcal{N}) = 1$ **or** $tries > limit$;
  log $diameter(\mathcal{N})$;
  assert $diameter(\mathcal{N}) \leq \log_2 |\mathcal{N}|$;
  assert $groups(\mathcal{N}) = 1$;
end

```
public void testcaseExecution(TesterSet testers) {
    short tries = 0;
    int groups;
    testers.execute("bootstrap");
    testers.execute("start");

    do {
        Thread.sleep(10000);
        testers.execute("updateModel");
        tries++;
        groups = model.groups();
    } while (!connectivity() && tries < 100);

    assert connectivity();
    assert diameter() <= (Math.log(testers.size()) / Math.log(2));
    testers.execute("quit");
}
```

Listing 12: Test Case (simplified)

system. The bootstrapping is a delicate process [68], especially when the whole system starts at same time. For instance, after a global outage, Skype took almost 48 h to heal in August 2007 [9].

The first test sequence creates a system with N nodes, waits for system stabilization and then verifies that all nodes belong to the same system (Property 1) and that the diameter of the system is $O(logN)$ (Property 2). Different scenarios execute this test sequence, with N increasing exponentially (2^n) from 16 up to 256 nodes.

Nodes	**FreePastry**	**OpenChord**
16	Pass	Pass
32	Pass	Pass
64	Pass	Pass
128	Pass	Pass
256	Fail	Fail

Table 5.1: Bootstrapping Test Results

Table 5.1 presents the results of the test sequence, which reveals a fault in the FreePastry bootstrapping process. Indeed, from some point between 128 and 255 nodes, the bootstrapper is unable to treat simultaneous requests. It seems that when the bootstrapper is overloaded by several parallel requests, it gives incorrect responses and induces the nodes to create several small and independent systems. This bug is particularly annoying when setting up tests for large systems, but has a workaround. The bootstrapper behaves correctly when the system is built up incrementally, respecting a delay of 100 ms between bootstrap queries.[7] The result of this test helped the developers of FreePastry to repair the bug and improve the robustness of the bootstrap process when preparing the version 2.1.[8]

The results of the test sequence also reveal a fault in OpenChord, leading to the same error found in FreePastry: the creation of several independent systems. However, in this case, the origin of the error is different. Nodes take too much time (more than 15 min) to find their neighbors and create a single system.

We use the global model to analyze the bootstrapping process, presented in Figure 5.4. After initializing all nodes and requesting them to join the system, we take snapshots of the topology every 10 s. While FreePastry nodes take less than 10 s to form a strongly connected graph, OpenChord nodes are unable to do it in less than 10 000 s. After this period, there are still 4 independent systems remaining. In both systems, once Property 1 was verified, Property 2 was also verified.

5.4.4 Node Isolation

Once we were sure that both systems respect Properties 1 and 2 (or at least 256 nodes in the case of OpenChord), we could run more elaborate tests. The second

[7] https://trac.freepastry.org/wiki/Planetlab
[8] https://trac.freepastry.org/changeset/4176

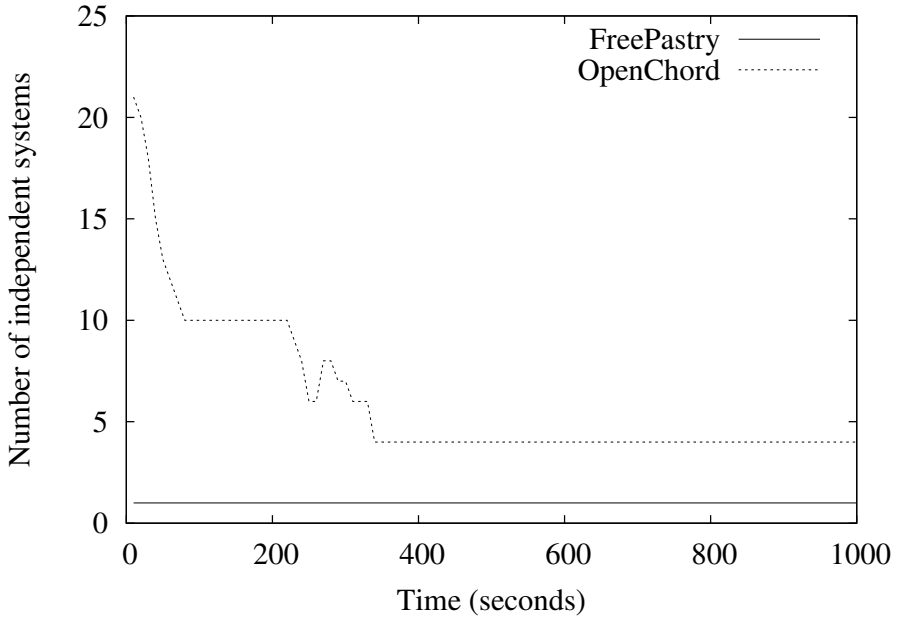


Figure 5.4: Bootstrapping process (512 nodes)

test sequence consists of two parts. First, to isolate a random node from the system. Second, to verify if such a node is able to correctly update its routing table to reach out a living node within a time limit (Properties 3 and 4). The test sequence has four steps:

1. The system is created and a set of nodes joins the system.
2. All nodes send the contents or their routing table to the global model.
3. All neighbors of a node n leave the system.
4. The routing table of n is periodically analyzed, until Property 4 (self-healing) is verified or a timeout is reached.

The routing table analysis happens as follows: the values from the updated routing table are compared with the neighbors of n before the isolation. If the intersection of these two sets of IDs is empty, then Property 4 is verified, the system is strongly connected again.

This test sequence is executed in only one test scenario, a system of 64 nodes. Indeed, creating a system with less than 64 nodes can lead the test to an inconclusive result because n may know all the nodes which are removed in the third step. In a larger system, the results should be similar since the size of the routing table would be the same.

The test showed that both implementations were able to correctly update their routing tables. OpenChord updated its routing table in about 4 seconds. This

delay represents a unique execution of the stabilization process (whose periodicity is set to 6 s). FreePastry needed about 30 s to update its routing table and become strongly connected again. The time was bigger than OpenChord's due to the manner that the routing mechanism is updated. In the first routing attempt, FreePastry always goes through the leaf set, which is promptly updated due to its small number of entries. Proving Properties 3 and 4 is expected to be fast through the leaf set. However, in corner cases when the leaf set is not enough to answer a request (e. g., due to the number of decommissioned nodes in the isolation scenario), the other tables are used and the lazy update approach works on (i. e., only updates any address when asked).

5.4.5 Routing Table Update on an Expanding System

The third test sequence checks the ability of a node to correctly update its routing table when the system is in expansion. More precisely, we verify that the nodes of a stable system take into account the new nodes that join their system. To do so, we use the global model to analyze the routing table of each node that belongs to a set of nodes N_1 to verify if it is correctly updated within a time limit, after the joining of a set of new nodes N_2.

This test case has four steps.

1. The system is started and nodes that belong to N_1 join the system.
2. Wait until the SUT reaches a stable state (Properties 1 and 2).
3. The new nodes (N_2) join the system and the global model is updated.
4. The strong connectivity of the system is verified: if all routing tables are correctly updated, then Property 4 is verified.

This test sequence is executed on different scenarios, with $|N_1| + |N_2|$ increasing exponentially (2^n) from 64 up to 1024 nodes.[9] In all executions, N_1 and N_2 have the same size. A maximum time is set to limit the test execution. This time limit starts from 8 s (allowing OpenChord to do at least one stabilization process) and increases in quadratic-logarithmic scale ($(log_n)^2$), corresponding to Property 3.

Figure 5.5 shows the average time for a node to update its routing table and to get a *pass* verdict. In this scenario, FreePastry had a similar result compared with the stabilization process of OpenChord. When a new node joins a FreePastry system, it needs to communicate with all its neighbors inducing the update of their routing tables. In OpenChord, the update takes a little longer due to the time to stabilize.

5.4.6 Routing Table Update on a Shrinking System

In this last test sequence, we verify that nodes that leave a stable system are correctly removed from the routing tables of the remaining nodes within a time limit. The test sequence is composed of four steps.

[9] 1024 nodes correspond to 8 nodes per machine in the clusters

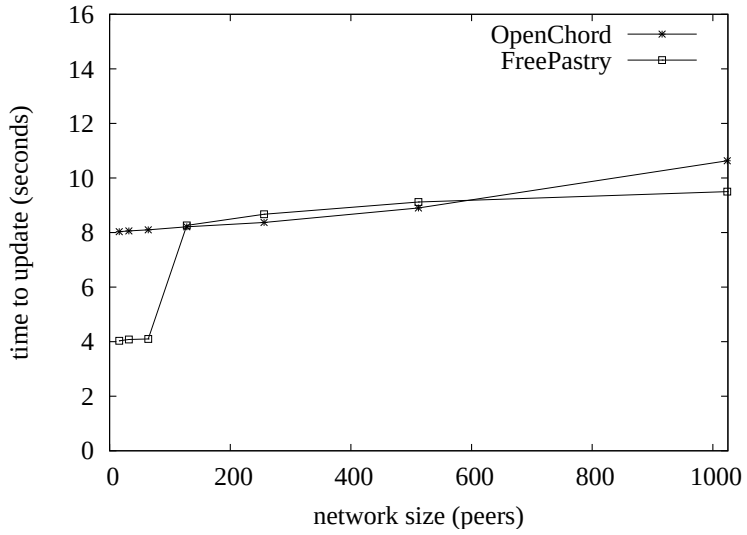


Figure 5.5: Routing table update (expanding system)

1. The system is created and all nodes join the system.
2. Wait until Properties 1 and 2 are verified.
3. Half of the nodes leave the system and the global model is updated.
4. Wait until strong connectivity of the system is verified again (Property 4).

In this scenario, the size of the system and the time limit also increase exponentially as described in Section 5.4.5. Figure 5.6 shows the minimum time necessary for a node to update its routing table and get a *pass* verdict.

As expected, OpenChord shows a faster routing table updating process than FreePastry due to its stabilization process. In fact, this stabilization process showed that it can detect the departures quickly and may be a better update approach compared with FreePastry.

5.4.7 Discussion

While Properties 1, 2 and 4 may be verified using information available at the SUT interface, the verification of Property 3 is more complex. In order to measure the number of exchanged messages, one must monitor the communication on all nodes of the SUT, filter the messages that are not related to the self-organization and verify that the number of messages exchanged corresponds to $O((logN)^2)$. In the tests presented above, we used a different approach: we measure the time needed for self-healing in different scales. If the time increases quadratic-logarithmically, we consider that the property is respected. The tests also showed an error in the bootstrapping process of OpenChord: the time needed to create a valid system with more than 500 nodes is unsatisfactory, making the implementation unusable.

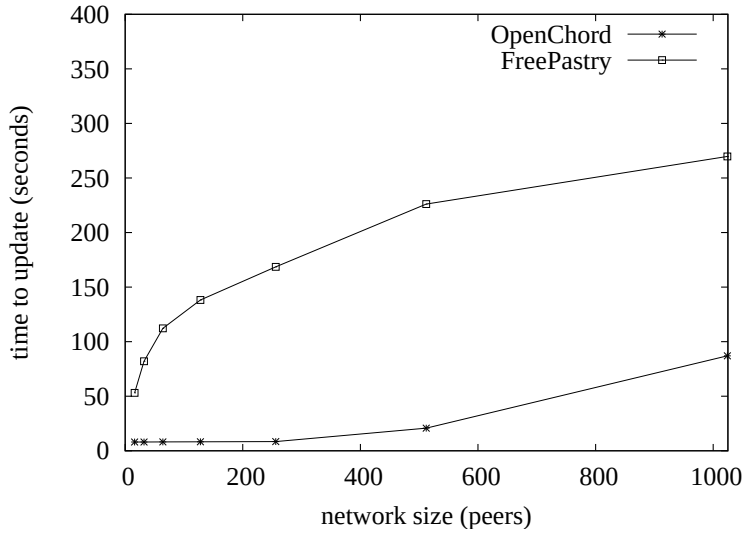


Figure 5.6: Routing table update (shrinking system)

A thorough analysis of the source code, the execution log and of the resource usage revealed a design error. Indeed, when a join request arrives at the bootstrapper node, it creates one thread to process the request.[10] Thus, when several requests arrive at the same time, the node spends more time creating threads (and context switching) than actually processing the requests. Applying the Proactor design pattern [110] would fix this error.

5.5 Conclusion

In this Chapter, we presented a model-based approach for checking global liveness properties that must be ensured by different large-scale distributed systems. We claim that global properties should be checked at runtime, at real scale, using non-invasive distributed testers, and that model-based testing is an expressive and adaptable technique to specify and check the global liveness properties of a system.

In our approach, test sequences put the system into states where such properties may be violated or lead to a degradation of system performance and behavior, while models provide a high abstraction level to represent a global view and the required properties of the SUT. Models are used as live oracles, which have a view of the current state of the system and can detect property violations.

Along with the approach, we extended the architecture presented in Chapter 5, to implement a model-based oracle. The oracle was written in Kermeta, a dedicated language for model transformation.

We illustrated this approach by testing the reliability of two routing algorithms under churn. Results showed common flaws in both routing strategies and clear

[10] Classes **SocketEndpoint** and **RequestHandler**.

differences. For instance, OpenChord could not build a single system (i. e., a strongly connected graph) during real-scale experiments, revealing a defect that would not have been detected without a global view of the system.

It is important to note that the approach is not limited to reliability testing. It can also target other distributed software testing techniques, such as system, load, and elasticity testing. Indeed, system testing [37] and load testing [82] were the subject of previous experiences. The approach is not limited to a specific class of distributed system either, but to specific classes of properties. More precisely, to properties that need a global view of the system to be checked, and that are only observable at specific states of the system.

In our approach, we are only interested in the state of the system at some specific points, reducing exchanged messages during a test. This choice prevents us from checking safety properties; i. e., properties that should always be true. However, these properties are typically local and could be checked using assertions. Since we chose not to instrument the code of the SUT, we can access neither the internal states of a node nor some particular attributes, such as exchanged messages raised during a given query.

Chapter 6

A Model-Driven Approach for Software Deployment

6.1 Introduction

Cloud Computing [11, 83] has been a hot topic in both of research and industry community recently. It can be described as a new kind of computing in which dynamically scalable and virtualized resources are provided as services over the Internet. Cloud users can access cloud system and use the service through different devices and interfaces. They only have to pay what they use according to Service Level Agreement contracts established between Cloud providers and Cloud users [26]. One of the main features of Cloud computing is the virtualization in which all cloud resources become transparent to the user. They do not need any longer to control and maintain the underlying cloud infrastructure. The virtualization in Cloud Computing combines a number of virtual machine images (VMIs) on the top of physical machines. Each virtual image hosts a complete software stack: it includes operating system, middleware, database, and development applications. The deployment of a VMI typically involves booting the image, as well as installation and configuration of software packages. In the traditional approach, the creation of a VMI to fit user's requirements and deploying it in the Cloud environment are typically carried out by the technical division of the Cloud service providers. They provide a platform as a service to the user according to SLA contracts signed between the service provider and the user. Usually, it is a pre-packaged platform with installed and configured software components. The standard VMI contains many software packages, which rarely get used and thus the image is typically larger than what would be necessary. This can lead to several difficulties, such as wastage of storage space, memory, operating costs, and waste of network bandwidth when cloning an image and deploying it on the cloud nodes [3].

In the traditional approach, when a cloud user requests a new development platform, the service provider administrators select an appropriate VMI for cloning and deploying on cloud nodes. If there is no match found, then a new one is created and configured to match the request. It can be generated by

modifying from the closest-fit existing VMI or from scratch. Several concerns would need to be addressed by the cloud providers, such as: (i) How to create an optimal configuration? (ii) Which software packages and their dependencies should be installed? (iii) How to find the best-fit existing VMI and how to obtain a new VMI by modifying this one?

Cloud service providers want to automate this process because the complexity of interdependency between software packages, and the difficulty of maintenance [31] is time-consuming for the creation of standard VMIs. In other words, they want to give users more flexibility when choosing the appropriate VMI to satisfy their requirements, while ensuring benefits for providers in terms of time, operating costs, and resources. In this chapter, we present an approach for managing VMI for Cloud Computing environments, providing a way to adapt to the needs of auto-scaling and self-configuring virtual machine images. In this approach, we consider VMIs as a product line and use feature models to represent VMI configurations and model-based techniques to handle automatic VMI deployment and reconfiguration. We claim that this approach makes the management (i.e., creation, configuration and adaption) of virtual image faster, more flexible and easier than the traditional approach. We validate this approach by an example showing that, given a base model representing all available artifacts, one can easily derive a configuration model (a specific use of a subset of artifacts) and generate all needed configuration scripts to generate its corresponding VMI. The chapter is organized as follows. Section 2 describes our solution of managing virtual machine image configurations by using feature models and using the model-driven approach for virtual machine image deployment in Cloud Computing environment. Section 3 introduces an example about deploying a Java web application development platform. Section 4 shows the experiment evaluations. Section 5 discusses the related work, and is followed by the conclusion and future work in Section 6.

6.2 Model-Driven Approach

In this section, we present a model-based approach for image provisioning. This approach uses an image with a minimal configuration, containing the operating system, some monitoring tools, and an *execution model*. The goal of the execution model is to install and configure software packages, after booting the deployed images.

6.2.1 Feature Modeling for VMI Configuration Management

Our approach uses feature modeling [74] to manage the configuration of virtual-machine images. In terms of configuration derivation, a feature model describes:

- The software packages that are needed to compose a Virtual Machine Image, represented as configuration options.
- The rules dictating the requirements, such as dependent packages and the libraries required by each software component.

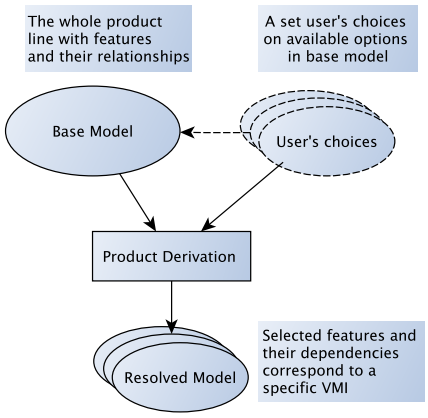


Figure 6.1: Feature Modeling Approach

- The constraining rules, which specifies how the choice of a given component restricts the choice of other components, in the same Virtual Machine Image.

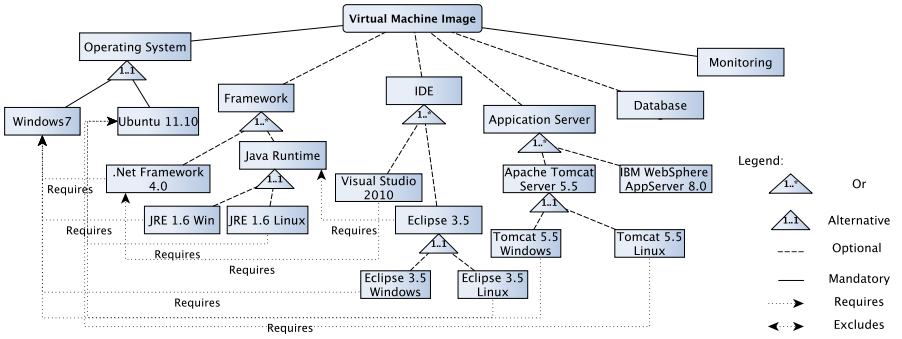


Figure 6.2: Feature Diagram Represents a Base Model

Feature models have a tree structure, with features forming the nodes of the tree and groups of features representing feature variability. There are four types of feature groups: *Mandatory*; *Optional*; *Alternative*; and *Or*. The model follows some rules when specifying which features should be included in a variant. If a variant contains a feature, then:

- All its *mandatory* child features must also be contained;
- Any number of *optional* child features can be included;
- Exactly one feature must be selected from an *alternative* group;
- At least one feature must be selected from an *or* group.

Feature models support two cross-tree constraints: *Requires*; and *Excludes*. Given two features, f_a and f_b: if f_a requires f_b, then the selection of f_a implies the selection of f_b; if f_a excludes f_b, then the selection of f_a prevents the selection of f_b.

Our approach deals with two models: base and resolved. The base model represents the whole product line, with all its features, their relationships, and constraints; The resolved model is obtained after the product derivation process, it contains selected features and their dependencies.

Base Model

The base model represents configuration options which would be used for composing a VMI. The elements of the base model are features of the configuration options of a VMI, they represent software packages and their dependencies. These elements become elements of resolved models, according to the resolutions of the corresponding selection models.

Figure 6.2 depicts a part of based model that represents VMI configuration features. In this model, features and their relationships represent software packages:

- *Operating System* is a mandatory child feature of *Virtual Machine Image*, which must be selected when *Virtual Machine Image* is selected.
- *Operating System* includes alternative child features: *Windows* and *Linux*
- When the *Operating System* feature is selected, then one of *Windows 7* or *Ubuntu 11.10* must be selected.
- If the feature *Ubuntu 11.10* is selected, then all features that require *Windows 7* cannot be selected, for instance: *Visual Studio 2010*, *JRE 1.6 Windows*, etc.

Base models are built by IT experts of cloud providers, who have knowledge about systems and software packages used to compose Virtual Machine Images. The correctness of the base model relies on the correctness of the feature model that represents the base models. Many approaches and tools were proposed to automate analysis of feature models [120, 18, 84]. They offer to validate, check satisfiability, detect the "dead" features and analyze feature models. In our implementation, we use constraints to ensure that the created feature model is valid, and that configurations, which are derived from this feature model, are also consistent with the base model. For example:

- Parent and child features cannot have a mutually exclusive relationship.
- Sibling features cannot be mutually exclusive.
- For two features f_1 and f_2, if f_1 requires f_2, then f_2 cannot require f_1.

Product Derivation

Product Derivation is a process that is responsible for the creation of the final configuration. It supports to derive the VMI configurations from the base model [126]. To create a specific configuration of a VMI, the designer selects some features from the base model and uses a mechanism to produce a suitable configuration. The selection of each feature is checked and validated by the Product Derivation process to ensure the selection is valid. When a feature is selected, the Product Derivation process checks its relationships. Features connected to the selected feature by a mutually exclusive relationship become unavailable on the base model for next selections. All of the features that are required by the selected feature are also selected.

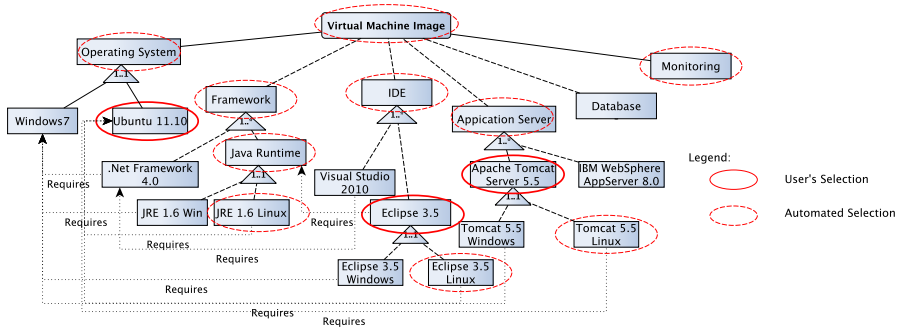


Figure 6.3: A Resolved Model Derived by the Product Derivation Process.

Resolved Models

A resolved model stores user's feature choices of the base model and their dependencies. It is derived from the Product Derivation process based on user 's selection on the base model. A resolved model corresponds to a specific configuration of a Virtual Machine Image. Figure 6.3 is an example of a resolved model that is derived from the base model presented in Figure 6.2 with the following user 's selections: operating system is Ubuntu 11.10, integrated development environment is Eclipse 3.5, and Apache Tomcat 5.5 for application server. According to the base model, Eclipse 5.5 requires Java Runtime. Both features have two alternative children: Windows and Linux. However, since the selected operating system is Ubuntu 11.10, only the Windows version is available. Addition, since Monitoring is a mandatory feature of Virtual Machine Image it must be selected.

By using feature models, cloud providers have flexibility to create Base models representing resources for VMI provisioning. Features represent software packages or hardware options, such as RAM or virtualization technology (e.g., KVM or Xen). These feature could also be used to store other informations: time, cost, memory usage, etc., to support finding optimal configurations. The first time for creating the base models might take time and need experts on software packages and their dependencies. However, once the Base model is created, it

helps cloud users during the selection and the creation of VMI configurations, reducing time, complexity, and errors during the manipulation.

6.2.2 Model-Based Deployment Architecture

Unlike the traditional approach, where software packages are installed and configured when the VMI template is created, the model-driven deployment approach installs and configures software packages at runtime when a VMI template is booted. The approach also supports synchronization of maintenance of the deployed VMIs at runtime. This mechanism allows users to update, remove, and add new components to running VMIs, without image shutdown and re-deployment. It is more flexible than the traditional approach.

In our approach, we create models that drive the creation of VMIs instance on demand. Every time a new virtual machine is created on the cloud node, the cloud provider selects features of VMI, generates configurations and applies the model to it. Figure 6.4 describes an overview architecture of our approach.

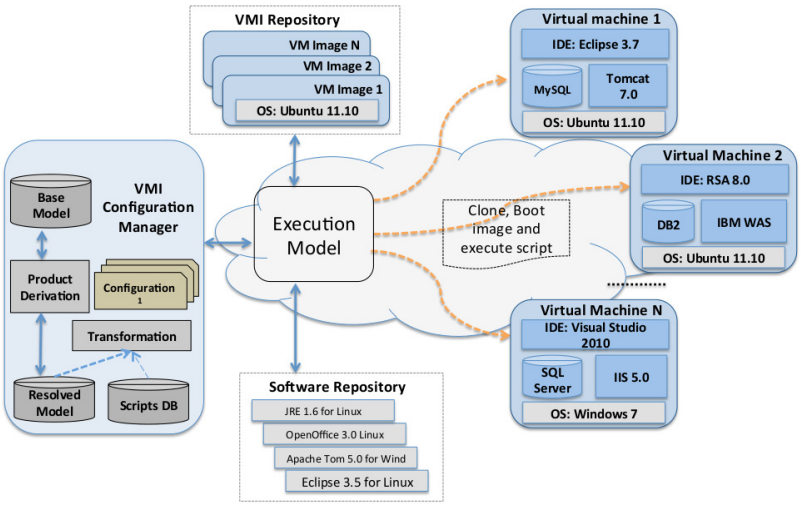


Figure 6.4: An Overall Architecture of Model-Driven Approach for VMI Deployment

- **VMI Repository**
  The VMI Repository contains basic virtual machine images that are used as the initial VMIs: e.g., `Ubuntu11.10.img`, `fedora15.0.img`. These are standard VMIs with minimum configuration, such as operating system and assistance software, like monitoring tools.

- **VMI Configuration Manager**
  The VMI Configuration Manager is responsible for the creation and the management of configurations of virtual machine image to fulfill requested requirements. By using the VMI configuration manager, users can easily

select the required software for creating the appropriate virtual machine. It also helps the cloud providers to manage the preparation and provision of resources as per client requirements.

- **Execution Model**
  The Execution Model is responsible for reserving cloud nodes, deploying virtual machines, and executing the corresponding configuration that resulted from the reasoning of VMI Configuration Manager. It is an encapsulation of Ruby and shell script files.

- **Cloud Nodes**
  Cloud Nodes are reserved nodes in the cloud infrastructure for hosting and running virtual machines.

- **Software Repository**
  The Software Repository stores software packages used to compose a VMI. It can be a file server inside the cloud infrastructure or other repositories from the Internet, such as the Debian repository.

6.2.3 Model-Based Deployment Process

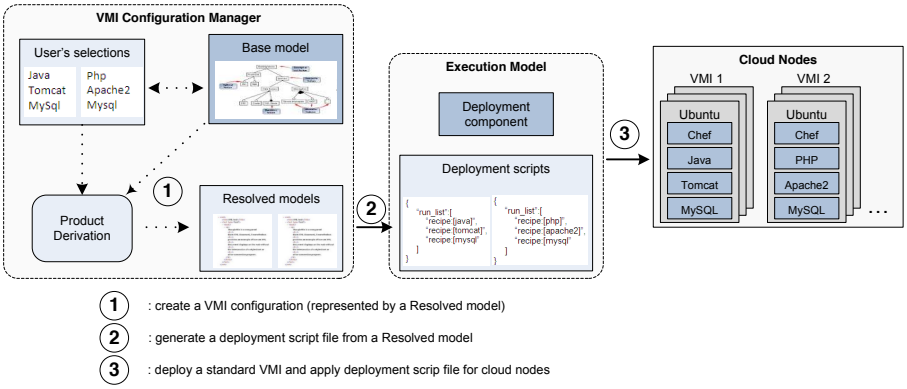


Figure 6.5: Model-Driven Process

The deployment process deals with the VMI Configuration Manager, the Execution Model, the Software Repositories, and the Cloud Nodes. It includes the following steps:

- **Create a VMI configuration.**
  In this step, cloud users interact with the VMI Configuration Manager to select configuration options from the base model. The VMI Configuration Manager analyzes the user's choices and generates a resolved model (i.e., a valid configuration of a VMI).

- **Generate a deployment script file.**
  A resolved model is transformed into a deployment script file for automatic deployment and configuration. In the current implementation, we use

Chef[1] to automatic install and configure software on a virtual machine. Chef is an installation software that cloud providers use to deploy, install, and configure software stacks on the cloud nodes at runtime. Chef requires an input file, describing the node configuration: the required software, as well as their role. Actually, the input file is a Ruby or JavaScript Object Notation[2] (JSON) source code.

- **Deploy a standard VMI and apply the deployment script file to the cloud nodes.**
  The Execution Model, based on the resolved model and deployment script file, selects a standard VMI and launches it on the reserved nodes. After that, it transfers the deployment script to the nodes and executes Chef. Finally, it returns the successful nodes to the cloud user.

6.3 An example of the VMI for Java Web Application

To illustrate our approach, we introduce an example of VMI provisioning for the Java Web Application Development platform. The configuration of this VMI includes an operating system, a web application server, a database management system, and a programming language compiler.

Cloud users select the required features on the base model by the using VMI Configuration Manager, for example: **Ubuntu**, **Eclipse**, **Apache Tomcat**, and **Database**. Figure 6.6 represents the selection of configuration options from the base model.

A VMI includes only one operating system, so the choice of Ubuntu feature is mutual exclusive with other operating systems and their dependencies. For example, the users can select neither the SQL Server nor the Eclipse for Windows because both features require Windows, which is a mutual exclusive feature of Linux Ubuntu. The features JRE 1.6 for Linux, Chef-Linux are auto-selected because Apache Tomcat requires JRE 1.6 for Linux and Chef-Linux is a mandatory feature.

The Product Derivation process generates a resolved model from the user's selections. The transformation from a resolved model into a script file helps to automatic install and configure software stacks that are selected in the resolved model. Figure 6.6 also shows the example of a resolved model and a deployment script file, which are corresponding to the user's selection from the base model. The Deployment script is a JSON file, named **deployscript.json**. The Execution model uses the script file for automatic installing and configuring software into the selected virtual image. Listing 13 presents a partial Ruby code for executing the script file on cloud nodes.

[1] http://wiki.opscode.com/display/chef/About
[2] http://www.json.org/

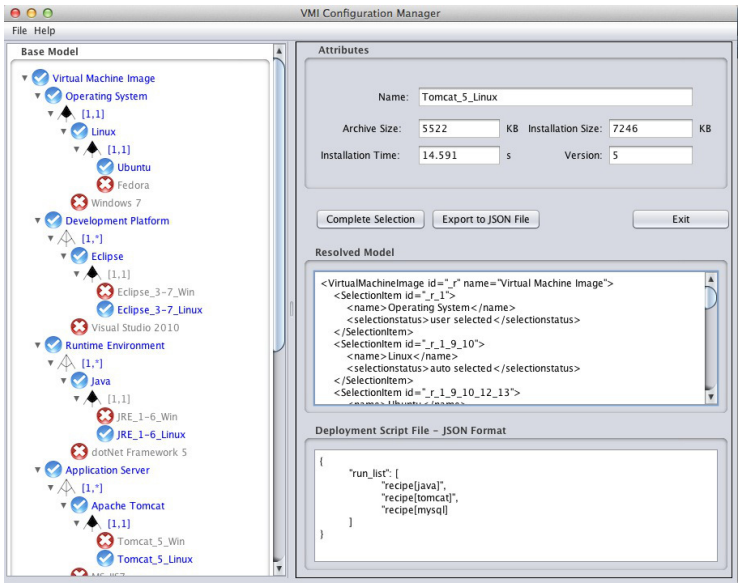


Figure 6.6: Example of VMI Configuration Manager

```
Net::SSH::Multi.start do |session|
  # access servers via a gateway
  session.via STRGATEWAY, CONFIG['username']
  deployment["nodes"].each do |node|
    session.use "root@#{node}"
  end
    url = 'http://public.grenoble.grid5000.fr/~tlenhan/TamlnChefScripts/'
    session.exec 'hostname'
    session.loop
    session.exec 'mkdir -p /tmp/chef-solo'
    session.loop
    session.exec 'wget ' url 'deployscript.json'
    session.loop
    session.exec 'chef-solo -j deployscript.json -r ' url 'cookbooks.tgz'
    session.loop
  end
```

Listing 13: A Partial Code in Ruby of the Execution on the Cloud Nodes

6.4 Experimental Validation

In this section, we present an experimental validation of our approach on the easiness of manipulation and the performance of deployment, in terms of data transfer and deployment duration. The experiment is executed on Grid5000[3], a virtualization infrastructure for research in France. We use Grid5000 's tools to reserve nodes and deploy VMIs to the nodes.

6.4.1 Scenario Description

Our simple scenario deployment generates a VMI that includes selected software stacks in the previous example (Java, Tomcat, MySQL). We deploy this VMI to the reserved nodes on Grid5000. We compare our approach to the traditional approach in terms of time for setting up the environment, amount of data transfer through the network, and operating steps. We evaluate the traditional approach in two cases:

- Case 1: There is no existing VMI that fits the requirements. The cloud provider needs to create a new VMI containing Java, Tomcat and MySQL.

- Case 2: There is an existing VMI that fits the requirements. It is used as a standard VMI for deploying on the cloud nodes. However, for meeting different user requirements, it also contains software that may not be used: Java, Tomcat, MySQL, Apache2, Jetty, PHP5, Emacs, PostgreSQL, DB2-Express C, Jetty, LibreOffice, etc.

6.4.2 Traditional Approach vs Model-Driven Approach

Time and Operations of the Deployment

In the traditional approach most decisions are taken by experts, because they require the knowledge of underlying systems and software dependencies. Our approach provides a graphical interface, the VMI Configuration Manager, which guides cloud users in the selection of a set of configuration options. After that, the Configuration Manager deploys the new VMI on cloud nodes, making easy to update and to maintain the running VMI. Table 6.1 shows a comparison between the traditional and the model-driven approaches, in terms of operations and deployment duration. Experiments show that the deployment duration of our approach is slightly better that the traditional approach, if there is an existing VMI that fits the requirements. However, if there is no appropriate VMI and the cloud provider creates a new one, then our approach is faster than the traditional approach.

Traditional Approach			Model-driven Approach		
Operations	*Handled by*	*Estimated Time*	*Operations*	*Handled by*	*Estimated Time*
1. Find the existing VMI in repository that fit the requirements. • If found: go to operation 3, • If not found: create a new one or modify an existing VMI	Expert (Manually)	1 minute	1. Create a VMI configuration & Generate a deployment script file	Expert & Non-expert (Manually)	2 minutes
2. Create a new VMI • Boot a clean VMI • Install and configure software • Save to an image	Expert (Manually)	11 minutes	2. Reserve 50 nodes & deploy the standard VMI to the nodes	Automatic	8 minutes
3. Reserve 50 cloud nodes & deploy the VMI to the nodes	Expert (Manually)	9 / 12 minutes (*)	3. Copy the deployment file to the running nodes & execute it to install, configure software	Automatic	2 minutes
Total time		21 / 13 minutes (**)	**Total time**		14 minutes

(*) : 9 minutes for case 1, and 12 minutes for case 2
(**): 21 minutes for case 1, and 13 minutes for case 2

Table 6.1: The Operations of Model-Driven Approach and Traditional Approach

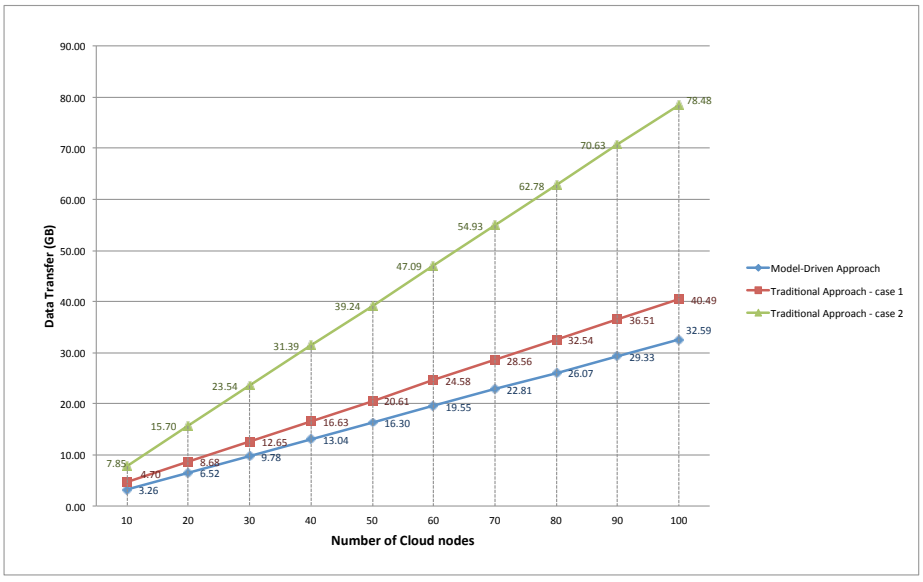


Figure 6.7: Data Transfer Through the Network of the VMI Deployment

Data Transfer Through the Network

In our experiment, we use a clean image **Squeeze-x64-nsf**[4] (333.587 MB), which is available on the Grid5000 's repository. This is also the standard VMI for the case 1 of the traditional approach. In our approach, we use minimal configuration images, only containing an installation software and its dependencies (i.e., Chef).

After the installation of the minimal software, the image size is 339.955 MB. In the case 2, unused software is installed for adapting different requirements from users. This makes the size of a standard VMI is much bigger, 803.60 MB.

Figure 6.7 shows that in both cases, the model-driven approach transfers less data than the traditional approach. Especially when the pre-packaged VMI contains more software installed, and deploy to a large number of cloud nodes. In this example, when we deploy 100 cloud nodes, the traditional approach transfers 40.49GB of data for case 1, and 78.48GB of data for case 2, while the model-driven approach only transfers 32.59GB of data. The traditional approach reduces the amount of data in 19.5% and 58.47%, comparing to cases 1 and 2, respectively.

6.5 Conclusion

In this chapter, we presented a model-driven approach to manage and create configurations, as well as deploy images for virtual machine image provisioning in Cloud Computing. We consider virtual images as product lines, use feature models to capture their configurations, and use model-based techniques for automatic deployment of virtual images. This approach makes the management of virtual image more flexible and easier to use than the traditional approach. On the implementation side, we developed a prototype for validating the approach. It helps cloud users to select configuration options, to create virtual images and to deploy them on cloud nodes. We used Grid5000 as a Cloud Computing environment testbed for deploying virtual images.

The framework includes two major parts: the VMI Configuration Manager and the Execution Model. The VMI Configuration Manager helps cloud users to select configuration options, create a valid configuration of a VMI through a graphical user interface. It also generates deployment script files. The Execution Model uses these files to automatically deploy and configure software into cloud nodes at runtime without any manual intervention.

We compared our approach to the traditional cloud deployment approach in two different scenarios, using an existing compatible VMI and creating a new one. Experiments showed that the model-driven approach helps cloud users to create the configurations and deploy VMIs on demand easily. It minimizes error-prone manual operations. Additionally, our approach reduces the network data transfer, comparing to the traditional approach. Especially, if a pre-packed VMI contains unwanted software. In this case, experiments showed that our approach

[3]https://www.grid5000.fr/mediawiki/index.php
[4]https://www.grid5000.fr/mediawiki/index.php/Squeeze-x64-nfs-1.1

reduces the data transfer up to 58.47%. It saves network resources during VMIs provisioning in Cloud Computing.

Our framework could be extended to support cloud users for estimating the deployment time and operational costs as needed. Therefore, it could improve the performance of virtual machine image provisioning. However, the reasoning engine of our Product Derivation process is still limited with simple constraints of the configuration. It is a challenge to deal with more elaborated configurations that have optimal requirements on the complex constraints of multiple parameters. In the future, we plan to improve the reasoning engine of the Product Derivation process, to deal with more complex configuration options and constraints. We believe that a reasoning engine could enhance the performance of the Product Derivation process in the VMI configuration management.

Currently, our current prototype only works in the Grid5000 environment. We are improving the prototype to have the ability to work with some open-source cloud platforms, such as OpenNebula, Nimbus, etc.

Chapter 7

Conclusion and Perspectives

7.1 Conclusion

Large-scale adaptive distributed systems are becoming more and more popular, meeting the society's needs for faster and wider information exchange, while ensuring reliability, security, privacy, performance, as well as several other quality factors. However, the popularity of those systems contrasts with the lack of verification techniques and tools to detect the presence of errors and to ensure their overall quality. This because the diversity of purposes and implementation paradigms prevents the creation of a generic verification technique, while specific verification techniques and tools are often expensive and not reusable. One could argue that simulation techniques are more adapted to those systems, since more generic and less expensive. Our claim is that large-scale systems must be verified at large-scale; we believe that the most important errors only manifest themselves during specific load conditions created when the system scales. We also believe that model-based testing techniques allows the creation of reusable generic tools that can be tailored to the particularities of each system.

In this document, we presented our experience over seven years testing large-scale systems. Our work focuses on the verification through testing of large-scale distributed systems. During these years, we adopted a four steps research methodology that consists of: (i) problem and challenge identification; (ii) theory elaboration; (iii) prototype development; and (iv) real-scale experimentation and observation. This efforts led to the following contributions:

A Distributed test architecture An important issue when testing large-scale systems is their heterogeneity, which prevents the use of a generic test architecture. The use of component-based models, as well as architectural languages allowing the dynamic configuration and deployment of components are an interesting approach to deal with this issue. We presented Macaw, a component-based architecture to test large-scale systems [41, 6, 36]. Macaw was implemented on the top of Kevoree, a frame-

work for developing dynamic distributed software. The main features of Kevoree allows the testing architecture to adapt itself in function of the specific requirements of the system under test and the test objectives. These requirements are related to monitoring, logs, data providing, etc.

An incremental methodology for testing large-scale systems To take into account the different dimensional aspects of large-scale systems, we presented a methodology that combines the functional testing of an application with the variations of the other two aspects [40, 39, 38, 35, 6, 82]. Indeed, we incrementally scale up the SUT either simulating or not volatility. Our incremental methodology is composed by the following steps:

1. small scale system testing without volatility;
2. small scale system testing with volatility;
3. large scale system testing without volatility;
4. large scale system testing with volatility.

In terms of diagnosis, this methodology allows to determine the nature of the detected erroneous behavior. Indeed, the problem can be linked to a purely functional cause (Step 1), a volatility issue (Step 2), a scalability issue (Step 3) or a combination of these three aspects (Step 4). The most complex errors are the last ones since their analysis is related to a combination of the three aspects. Steps 2 and 4 can be composed of two other steps (shrinkage and expansion), to help the diagnosis of errors due to either the unavailability of resources or arrival of new ones. Several rates of volatility can be explored to verify how they affect the functionality aspect of the SUT (e. g., 10 % joining, 20 % leaving).

A model-based approach for oracle implementation We presented a model-based approach for checking global liveness properties that must be ensured by different large-scale distributed systems [49, 50, 116]. We claim that global properties should be checked at runtime, at real scale, using non-invasive distributed testers, and that model-based testing is an expressive and adaptable technique to specify and check the global liveness properties of a system. In our approach, test sequences put the system into situations where such properties may be violated or lead to a degradation of system performance and behavior, while models provide a high-abstraction level to represent a global view and the required properties of the system under test. Models are used as live oracles, which have a view of the current state of the system and can detect property violations.

A model-based software deployment approach We presented a model-driven approach to manage and create software configurations, as well as deploy images for virtual machine image provisioning in distributed systems. We considered virtual images as product lines [77, 91, 119]. We used feature models to capture their configurations and model-based techniques for automatic deployment of virtual images. This approach makes the management of virtual image more flexible and easier to use than the traditional approach. On the implementation side, we developed a prototype for validating the approach. It helps cloud users to select configuration options,

to create virtual images and to deploy them on nodes. We used Grid5000 as a distributed environment testbed for deploying virtual images.

The framework includes two major parts: the VMI Configuration Manager and the Execution Model. The VMI Configuration Manager helps cloud users to select configuration options, create a valid configuration of a VMI through a graphical user interface. It also generates deployment script files. The Execution Model uses these files to automatically deploy and configure software into cloud nodes at runtime without any manual intervention.

In future work, the author will continue to investigate the application of model-based techniques to the verification of large-scale systems. The next section details this future work.

7.2 Perspectives

In 2020, with more than 50 billions of connected devices exchanging more than 1 zettabyte/year of data[1], will distributed software testing be a matter for software engineering or for data analytics[2]? Clearly, there is a limit for testing software in real scale: systems composed of millions of heterogeneous connected nodes, processing terabytes of data, would require a prohibitive investment to be completely tested. Therefore, software testing will be preceded by a comprehensive analysis of the software ecosystem, to select meaningful patterns and to drive the test campaign.

Data analytics requires a high-abstraction level for querying and manipulating data. In this context, testers will need powerful tools to overcome the complexity of distributed systems testing. We believe that Model-Based Engineering can offer a common base for data analytics, provided that current tools evolve to be able to scale up and process larges models, i. e., models with several thousands of elements. Then, these tools could establish a common base for structured and unstructured test data process, transformation, query, validation, and comparison.

The emergence of cloud infrastructures has a double impact on software testing. In one side, it provides an efficient and economical way to conduct large-scale tests. In the other side, the usage of more flexible infrastructure introduces new software quality factors that must be tested, the elasticity, or the ability of a software to ensure an adequate response time, according to the workload using the minimal configuration; the variability, or its ability to be changed, customized, or configured according to variable functional and non-functional requirements; and the volatility, or its ability to remain reliable during churn.

The introduction of these new factors emphasizes the need for non-functional testing. In our work, we claimed that non-functional properties should be verified along with functional properties. Henceforth, we intend to explore another path: verify non-functional properties independently from the functional ones.

[1] according to CISCO's predictions

[2] Analytics is the discovery of meaningful patterns in data and infrastructure.

The main idea is to ensure that the same test sequence produces the same output (functional equivalence) and look for non-functional *anomalies* on different configurations, versions, workloads, etc. By anomalies, we mean tangible differences during test sequence execution: different response time, resource usage, etc.

Detecting anomalies requires the generation of efficient test sequences. We intend to use evolutionary [15] and novelty search [78] algorithms to modify an initial test sequence and an evaluation function to select (discard) the best (worse) test cases.

After analyzing this approach for test generation, a question arose as how to evaluate the quality of distributed test cases? Indeed, current approaches for test evaluation (e. g., coverage, mutation analysis) are not fully adapted to multi-instances software, i. e., software where nodes share the same replicated code, and new criteria for distributed test case evaluation is needed.

The next sections provide detailed research perspectives in large-scale model-driven testing. Section 7.2.1 proposes new criteria for evaluating distributed test cases. Section 7.2.2 describes the challenges for testing the elasticity of cloud applications and draws a solution for generating test workloads. Section 7.2.3 proposes the application of search-based algorithms for generating test data. Section 7.2.4 describes a new approach for large model persistence. Finally, Section 7.2.5 proposes a domain specific language for deploying software on distributed environments.

7.2.1 Distributed Test Evaluation

Two approaches are widely used to measure the quality of tests: mutation analysis [43] and code coverage [85]. Mutation analysis consists of inserting small errors into the source code, generating erroneous software instances, called *mutants*. The quality of test sequences is then associated to its ability to find the mutants (i. e., produce fail verdicts). Code coverage associates the quality of a test with the amount of code that is covered during its execution. More the code is covered, better is the quality of the tests. While both approaches are perfectly adapted to single-instance software, they can not be directly applied to large-scale distributed software, where the same software is potentially deployed on several nodes. Applying mutation analysis in this case is difficult because on one hand, if the same mutant is replicated all over the system, the inserted error may become less subtle and could be easily detected by any test case. On the other hand, deploying a mutant on a single node at a time, may lead to a combinational problem, which is already a common issue in mutation analysis.

Similarly, code coverage criteria are not adapted to multi-instances software. Indeed, since system nodes often share (total or partially) the same implementation, i. e., the same source code and play different roles in the system, they exercise different parts of the code. Thus, testing evaluation methods based on code coverage must be adapted. For instance, the criterium “the minimum acceptable code coverage” may be interpreted as the percentage of code that should be reached by a single node, by all nodes or by the union of code coverage of all nodes.

We believe that code coverage is more adapted for evaluating test case and data. However, evaluating the results of code coverage on multiple-instances software is still an issue. Since each software instance may have a different role on the whole system and executes different parts of the code, code coverage may be different on each node. For instance, a simple criterium, the minimum code coverage, may have different interpretations:

1. The minimum code coverage that must be observed in at least one node.
2. The minimum code coverage that must be observed in all nodes.
3. The minimum common code coverage that must be observed in all nodes.

In future work, we intend to propose new criteria for distributed software, which consider the multiplicity of software instances. This new criteria distinguish the code covered by a single software instance from the global accumulated code coverage.

Let us denote by T a Distributed Test Sequence, by τ a Distributed Test Case, $\tau \in$ T, $\{1, \ldots, n\}$ denote the set of instances of software S, we name individual code coverage cc_i^τ the code covered by test case τ in instance i, accumulated code coverage $ACC^\tau = \{cc_1 \cup cc_2 \cup \ldots cc_n\}$, the union of all individual code coverage, and common code coverage $CCC^\tau = \{cc_1 \cap cc_2 \cap \ldots cc_n\}$, the intersection of all individual code coverage. Thanks to this distinction, we can introduce new criteria for defining test objectives:

Minimum Global Coverage The minimum coverage rate that should be verified in all nodes. This criterium is expressed as:
$\forall cc \in ACC : cc > min$.

Minimum Individual Coverage The minimum coverage rate that should be verified in at least one node. This criterium is expressed as:
$\exists cc \in ACC : cc > min$.

Minimum Common Coverage The minimum common code coverage rate that should be verified in all nodes. This criterium is expressed as:
$\forall cc \in CCC : cc > min$.

We intend to perform several experiments, using mutation operator to insert errors in the SUT, in order to answer to different empirical questions:

1. What extent of coverage rates (ACC and CCC) should be expected for a multi-instances software?
2. Is it easier to detect simple errors present on a single node than errors present in several nodes?
3. What is the impact of test data and workload on the code coverage?

7.2.2 Elasticity Testing

The unpredictability of web application and services workloads, often created by prompt events (e. g., tragic news, flash sales, instant popularity, etc.), has motivated the adoption of Cloud technologies. Indeed, Cloud Computing infrastructures provide a flexible environment that adds and removes resources from/to running applications, according to the workload, ensuring their elasticity. This elasticity exposes applications to new workload-related states (Scaling Up and Scaling Down), in which the infrastructure is adding or removing resources. These two new states complete other common web application workload-related states, such as Stress, Under-Loading, Pressure, etc.

We believe that these new states may reveal new workload-related errors and that the application should be tested under these states. Testing application in these states leads to two difficulties. First, the states are limited in time, the addition (removal) of resources only takes a few seconds during which the tests should be executed. Second, it is difficult to define an adequate workload variation for reaching these states. In one hand, if the variation is too low, a state transition may last too long to be triggered. In the other hand, if the variation is too high, the application may be directly lead to a Stress or even a Thrash state [82].

In future work, we intend to propose a load generation approach. The rationale of the approach is to divide the generation in two different steps. The first step exercises the web application with gradual workload increasing (or decreasing) until it reaches the desired states. The second step drives the web application through a list of desirable states, based on the levels of workload variation gathered previously. During the second phase different kinds of test may be applied to verify if the application behaves correctly in these elasticity-related states.

To validate our approach, we intend to conduct several experiments on the OpenShift cloud platform. The main idea is to demonstrate the feasibility of generating a workload set that drives the application into the different states and of creating different test scenarios from a given sequence of transitions.

7.2.3 Automatic Test Data Generation

In general, manually creating test cases for testing software systems is time consuming and error-prone, making necessary the automation of this process. In fact, meta-heuristic search techniques such as Genetic Algorithms (GAs) are frequently used in order to automate the test data generation process and gather relevant test cases through the wide search space [105, 124]. These techniques are especially applied for structural white-box testing. For coverage-oriented approaches, applying Evolutionary Algorithms (EAs) to test data generation has been focused on finding input data for a specific path of the program in accordance with a coverage criterion (e. g., longest path executed). The problem with coverage-oriented approaches is that search-based techniques cannot exploit the huge space of possible test data. In fact, some structures of the system cannot be reached since they are executed only by a small portion of the input domain. Applying GAs for test data generation consists in searching for

relevant test data according to an objective function that tries for example to maximize the number of statements or branches covered. The use of a fitness function as a coverage criterion to guide the search to detect relevant test data usually create many local optima to which search may converge. Thus, if the relevant test data, that could coverage the longest path of the program, lie far from the search space defined by the gradient of the fitness function, then some promising search areas may not be reached. The issue of premature convergence to local optima has been a common problem in GAs. Many methods are proposed to avoid this problem [14, 57]. However, all these alternatives use a fitness-based selection to guide the search.

In future work, we intend to introduce the use of Novelty Search algorithm to the test data generation problem. In this approach, we intend to explore the search space of possible test input values without regarding to any objectives (there is no fitness function). In fact, instead of having a fitness-based selection, we rather select test cases based on a novelty score showing how different they are compared to all other test data evaluated so far. So during the evolutionary process, we use to select test data that remain in sparse regions of the search space in order to guide the search through novelty. We intend to use the statement coverage metric as a coverage criterion to our Novelty Search-based test data generation.

7.2.4 Model Scalability

Part of the software industry is embracing the main concepts of Model-Driven Engineering, by putting models and code generation at the center of their software-engineering processes. Recent studies [107], as well as the proliferation of tools related to MDE, testify the increase in popularity of these concepts, which are applied to different contexts and scales. Some examples of contexts, besides software testing, are reverse engineering [24, 25], model transformation [93, 64], and code generation [20, 90]. The scale varies from manual modeling activities with hundred of elements to very large models, VLM, i. e., with millions of elements. Very large models are found in some specific domains, such as the automotive industry [19], civil engineering [112], or software product lines [99], or are automatically generated during software modernization of large code bases.

Among the model-based frameworks currently available, the Eclipse Modeling Framework [47] (EMF) has become a *de facto* standard for building modeling tools, providing a common basis for different contexts: The Eclipse marketplace[3] attests the popularity of EMF: it currently counts more than two hundred EMF-based tools [46] coming from both, industry and academia. However, the technologies at the core of EMF were designed in the first place to support simple modeling activities and exhibit clear limits when applied to very large models. Problems in accessing and persisting models of this size are under-studied and the current standard solution is to use a model/relational persistence layer, e. g., CDO for EMF [48], that translates runtime model-handling operations into relational database queries.

[3] http://marketplace.eclipse.org

As future work, we intend to propose an alternative framework for persisting models using two different persistence backends: a graph database and a raw database engine. The main idea is allow users to choose between query-ability or performance.

The first solution will be build on top of the popular graph database Neo4j, providing users the ability to query models using a dedicated query language as well as several interestinf features: online backups, horizontal scalability and advanced monitoring.

The second solution does not interface the modeling framework with a full-fledged database, but is built around a raw database engine and directly accesses its low-level data structures. We show that working at this level gives more flexibility in selecting the data structures that optimize each model-handling operation. No translation into a database query language is performed, thus reducing overhead.

We intend to evaluate both solutions performing a set of queries in the domain of software modernization, and we compare the execution performance of these queries with the *de facto* standard persistence layers for EMF: XMI and CDO[48].

7.2.5 A Domain Specific Language for test deployment in the Cloud

The recent maturation of Cloud computing technologies is leading companies to develop new applications in the Cloud, or to migrate existing on-premises applications to the Cloud. Cloud computing is primarily used for simplicity and financial reasons. Indeed, customers have a 24/7 access to computing facilities, in the form of servers, on a Pay-As-You-Go (PAYG) pricing model. Cloud providers propose a wide range of resources in terms of machine characteristics, operating systems, locations, etc. Customers, that have different needs, appreciate this variability, but it is important for them not to be dependent on one single provider: They must be able to migrate from one to another if the prices, policies, quality of service, or requirements change.

In order to ensure provider-independence, applications must be deployed and tested on different providers. It is then the responsibility of the Software-Testing Engineer (STE) to write scripts to deploy the application and run the tests. As each provider offers its own Application Programming Interface (API) to interact with the services, the STE has to duplicate and adapt the deployment scripts. For instance, if the STE has to test an application on Google Compute Engine [63], Amazon EC2 [62] and Rackspace [102], three versions of the same script must be written. This way of considering variability is obviously time-consuming, error prone, and counterproductive. The problem considered in this paper can be formulated as follows: How to provide STEs with an automated and provider-independent method to deploy and test Cloud applications?

Several solutions have been proposed to manage the automatic deployment of software and the entire life-cycle of running software [67, 76, 79, 94]. Unfortunately, while they are complete, they are also complex in terms of learning process and architectures, and they are often low-level. Their goal is to make

the deployment faster or easier, but ultimately, the STE suffers from using such complex tools. In addition, these tools are not designed to support the flexibility in choosing a provider, even if some of them are compatible with specific Cloud providers.

Our objective is to automate the deployment of Cloud applications in a provider-independent manner. In order to achieve this objective, we propose the following contributions: First, define a DSL that allow STEs to describe, independently of the provider, how an application has to be deployed, and which Cloud resources are required and available for the deployment. Second, define a mechanism that automatically generates deployment scripts. These scripts fulfill the needs described using the DSL, and are directly executable as they invoke commands of the providers' APIs.

Bibliography

[1] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[2] *The Gnutella protocol specification.* http://dss.clip2.com/GnutellaProtocol04.pdf, 2000.

[3] Model-driven application deployment for cloud computing environments. White Paper, Sun Microsystem Inc., January 2010. Available online (18 pages).

[4] ISO International Standard 9646. Open systems interconnection conformance testing methodology and framework, 1991.

[5] Reza Akbarinia, Vidal Martins, Esther Pacitti, and Patrick Valduriez. Replication and query processing in the APPA data management system. *Distributed Data and Structures 6 - WDAS. Records of the 6th Int. Meeting (Lausanne, Switzerland), Waterloo. Carleton Scientific*, 2004.

[6] Eduardo Almeida, João Eugenio Marynowski, Gerson Sunyé, Yves Le Traon, and Patrick Valduriez. Efficient distributed test architectures for large-scale systems. In *ICTSS 2010: 22nd IFIP Int. Conf. on Testing Software and Systems*, Natal, Brazil, November 2010.

[7] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, December 2004.

[8] Gabriel Antoniu, Luc Bougé, Mathieu Jan, and Sébastien Monnett. Large-scale deployment in P2P experiments using the JXTA distributed framework. Technical report, JXTA - jdf.jxta.org, 2004.

[9] Villu Arak. What happened on august 16, August 2007.

[10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing, Feb 2009.

[11] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A Berkeley

view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[12] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[13] Rena Bakhshi, François Bonnet, Wan Fokkink, and Boudewijn R. Haverkort. Formal analysis techniques for gossiping protocols. *Operating Systems Review*, 41(5):28–36, 2007.

[14] Wolfgang Banzhaf, Frank D Francone, and Peter Nordin. The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets. In *Parallel Problem Solving from Nature—PPSN IV*, pages 300–309. Springer, 1996.

[15] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing: Research articles. *Softw. Test. Verif. Reliab.*, 15:73–96, June 2005.

[16] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[17] Boris Beizer. *Software testing techniques (2. ed.)*. Van Nostrand Reinhold, 1990.

[18] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. *Lecture Notes in Computer Science*, 3520:381–390, 2005.

[19] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental evaluation of model queries over emf models. In *Model Driven Engineering Languages and Systems*, pages 76–90. Springer, 2010.

[20] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. 2013.

[21] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[22] Gordon Blair, Nelly Bencomo, and Robert B France. Models@ run. time. *Computer*, 42(10):22–27, 2009.

[23] Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@runtime. *IEEE Computer*, 42(10):22–27, 2009.

[24] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco: A generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM.

[25] Hugo Brunelire, Jordi Cabot, Grgoire Dup, and Frdric Madiot. Modisco: a model driven reverse engineering framework. *Information and Software Technology*, (0):–, 2014. on press.

[26] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[27] Leo Cacciari and Omar Rafiq. Controllability and observability in distributed testing. *Information & Software Technology*, 41(11-12):767–780, 1999.

[28] H. Casanova, A. Legrand, and M. Quinson. Simgrid: A generic framework for large-scale distributed experiments. In *Computer Modeling and Simulation, 2008. UKSIM 2008. Tenth International Conference on*, pages 126–131, 2008.

[29] Kai Chen, Fan Jiang, and Chuan dong Huang. A new method of generating synchronizable test sequences that detect output-shifting faults based on multiple uio sequences. In *SAC*, pages 1791–1797, 2006.

[30] Wen-Huei Chen and Hasan Ural. Synchronizable test sequences based on multiple uio sequences. *IEEE/ACM Trans. Netw.*, 3(2):152–157, 1995.

[31] T.C. Chieu, A. Mohindra, A.A. Karve, and A. Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on*, pages 281 –286, oct. 2009.

[32] Cloudera. MRUnit Project, 2011.

[33] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems - concepts and design (2. ed.)*. International computer science series. Addison-Wesley, 1994.

[34] Pierre-Charles David, Thomas Ledoux, Thierry Coupaye, and Marc Léger. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *annals of telecommunications*, 2008.

[35] Eduardo Cunha de Almeida. Test et validation des systémes pair-à-pair. Technical Report ED 503-036, Université de Nantes, 2009.

[36] Eduardo Cunha de Almeida, João Eugenio Marynowski, Gerson Sunyé, and Patrick Valduriez. Peerunit: a framework for testing peer-to-peer systems. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 169–170. ACM, 2010.

[37] Eduardo Cunha de Almeida, Gerson Sunyé, Yves Le Traon, and Patrick Valduriez. Testing peer-to-peer systems. *Empirical Software Engineering*, 15:346–379, 2010. 10.1007/s10664-009-9124-x.

[38] Eduardo Cunha de Almeida, Gerson Sunyé, Yves Le Traon, and Patrick Valduriez. A framework for testing peer-to-peer systems. In *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Redmond, Seattle, USA*. IEEE Computer Society, 2008.

[39] Eduardo Cunha de Almeida, Gerson Sunyé, Yves Le Traon, and Patrick Valduriez. Testing peers' volatility. In *23nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), September 15-19, 2008, L'Aquila, Italy*, 2008.

[40] Eduardo Cunha de Almeida, Gerson Sunyé, and Patrick Valduriez. Action synchronization in p2p system testing. In Anne Doucet, Stéphane Gançarski, and Esther Pacitti, editors, *Proceedings of the 2008 International Workshop on Data Management in Peer-to-Peer Systems, DaMaP 2008, Nantes, France, March 25, 2008*, ACM International Conference Proceeding Series, pages 43–49. ACM, 2008.

[41] Eduardo Cunha de Almeida, Gerson Sunyé, and Patrick Valduriez. Testing architectures for large scale grids. In *International Workshop on High-Performance Data Management in Grid Environments, HPDGrid, Toulouse, France, June 24, 2008*, 2008.

[42] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[43] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41, April 1978.

[44] Florin Dragan, Bogdan Butnaru, Ioana Manolescu, Georges Gardarin, Nicoleta Preda, Benjamin Nguyen, Radu Pop, and Laurent Yeh. P2PTester: a tool for measuring P2P platform performance. In *A demonstration in BDA conference*, 2006.

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

[46] Eclipse Foundation. Eclipse Marketplace - Modeling Tools, 2014. URL: `http://marketplace.eclipse.org/category/categories/modeling-tools`.

[47] Eclipse Foundation. Eclipse Modeling Framework Project (EMF), 2014. URL: `http://www.eclipse.org/modeling/emf/`.

[48] Eclipse Foundation. The CDO Model Repository (CDO), 2014. URL: `http://www.eclipse.org/cdo/`.

[49] Olivier Finot, Jean-Marie Mottu, Gerson Sunyé, and Christian Attiogbé. Partial test oracle in model transformation testing. In Keith Duddy and Gerti Kappel, editors, *Theory and Practice of Model Transformations* -

6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings, volume 7909 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2013.

[50] Olivier Finot, Jean-Marie Mottu, Gerson Sunyé, and Thomas Degueule. Using meta-model coverage to qualify test oracles. In Benoit Baudry, Jürgen Dingel, Levi Lucio, and Hans Vangheluwe, editors, *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, September 29, 2013*, volume 1077 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[51] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5:345–, June 1962.

[52] I Foster and C Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufman Publishers, Inc., San Francisco, CA, USA, 1999.

[53] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.M. Jezequel. A dynamic component model for cyber physical systems. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, pages 135–144. ACM, 2012.

[54] François Fouquet, Erwan Daubert, Noël Plouzeau, Olivier Barais, Johann Bourcier, and Jean-Marc Jézéquel. Dissemination of reconfiguration policies on mesh networks. In Karl M. Göschka and Seif Haridi, editors, *DAIS*, volume 7272 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2012.

[55] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. An eclipse modelling framework alternative to meet the models@runtime requirements. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *MoDELS*, volume 7590 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2012.

[56] Prasanna Ganesan, P Krishna Gummadi, and Hector Garcia-Molina. Canon in G Major: Designing DHTs with Hierarchical Structure. In *ICDCS*, pages 263–272. IEEE Computer Society, 2004.

[57] Chris Gathercole and Peter Ross. An adverse interaction between crossover and restricted tree depth in genetic programming. In *Proceedings of the 1st annual conference on genetic programming*, pages 291–296. MIT Press, 1996.

[58] Robert M. Hierons. Testing a distributed system: generating minimal synchronised test sequences that detect output-shifting faults. *Information and Software Technology*, 43(9):551–560, 2001.

[59] William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles. Fault injection in distributed java applications. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*, 2006.

[60] F. Howell and R. McNab. simjava: A discrete event simulation library for java. *In 1998 International Conference on Web-Based Modeling and Simulation. Society for Computer Simulation International (SCS), January 1998.*, 1998.

[61] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[62] Amazon Web Services Inc. Aws amazon elastic compute cloud (ec2) scalable cloud servers, 2014.

[63] Google Inc. Compute engine - google cloud platform, 2014.

[64] INRIA and LINA. ATLAS transformation language, 2014.

[65] Claude Jard. Principles of distribute test synthesis based on true-concurrency models. Technical report, IRISA/CNRS, 2001.

[66] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 2005.

[67] Apache jClouds. Apache jclouds, 2014.

[68] Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. The bootstrapping service. In *ICDCS Workshops*, page 11. IEEE Computer Society, 2006.

[69] Jean-Marc Jézéquel. Model Driven Design and Aspect Weaving. *Journal of Software and Systems Modeling (SoSyM)*, 7(2):209–218, may 2008.

[70] Ralph Johnson and Bobby Woolf. The Type Object Pattern, 1997.

[71] Pallavi Joshi, Haryadi S Gunawi, and Koushik Sen. Prefail: Programmable and efficient failure testing framework. Technical Report UCB/EECS-2011-3, University of California at Berkeley, Jan 2011.

[72] Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[73] Gueyoung Jung, Calton Pu, and Galen Swint. Mulini: an automated staging framework for qos of distributed multi-tier applications. In *Proceedings of the 2007 workshop on Automating service quality: Held at the International Conference on Automated Software Engineering (ASE)*, WRASQ '07, pages 10–15, New York, NY, USA, 2007. ACM.

[74] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[75] E Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53, Jan 1994.

[76] Puppet Labs. Puppet labs: It automation software for system administrators, January 2014.

[77] Tam le Nhan. *Model-Driven Software Engineering for Virtual Machine Images Provisioning in Cloud Computing.* PhD thesis, Université de Rennes, 2013.

[78] Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, 2011.

[79] Apache Libcloud. Apache libcloud python library - apache libcloud is a standard python library that abstracts away differences among multiple cloud provider apis, 2014.

[80] Yunhao Liu, Xiaomei Liu, Li Xiao, Lionel M. Ni, and Xiaodong Zhang. Location-aware topology matching in p2p systems. In *INFOCOM*, 2004.

[81] Eliane Martins and Maria de Fátima Mattiello-Francisco. A tool for fault injection and conformance testing of distributed systems. In Rogério de Lemos, Taisy Silva Weber, and João Batista Camargo Jr., editors, *LADC*, volume 2847 of *Lecture Notes in Computer Science*, pages 282–302. Springer, 2003.

[82] Jorge A. Meira, Eduardo Cunha de Almeida, Yves Le Traon, and Gerson Sunyé. Peer-to-peer load testing. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *ICST*, pages 642–647. IEEE, 2012.

[83] Peter Mell and Timothy Grance. The nist definition of cloud computing. Technical report, National Institute of Standard and Technology - NIST, 2011.

[84] M. Mendonça, A. Wasowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *13th International Conference on Software Product Lines (SPLC 2009), San Francisco, CA, USA*, 2009.

[85] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, February 1963.

[86] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing. Technical Report HPL-2002-57R1, HP Labs - Hewlett Packard, 2003.

[87] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models at runtime to support dynamic adaptation. *IEEE Computer*, pages 46–53, October 2009.

[88] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.

[89] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[90] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe, William Piers, Cédric Brun, Laurent Goubet, Yvan Lussaud, and Freddy Allilaire. Acceleo user guide, 2006.

[91] Tam Nhan, Gerson Sunyé, and Jean-Marc Jézéquel. A Model-Driven Approach for Virtual Machine Image Provisioning in Cloud Computing. In *European Conference on Service-Oriented and Cloud Computing*, pages 107–121, 2012.

[92] OMG. MOF 2.0 core specification. Technical Report formal/06-01-01, OMG, April 2006. OMG Available Specification.

[93] OMG. MOF 2.0 QVT final adopted specification (ptc/05-11-01), April 2008.

[94] opscode. Chef, it automation for speed and awesomeness, 2014.

[95] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[96] C. Perkins. *Ad Hoc Networking*. Addison Wesley, 2001.

[97] Simon Pickin, Claude Jard, Thierry Heuillard, Jean-Marc Jézéquel, and Philippe Desfray. A uml-integrated test description language for component testing. In *UML2001 workshop: Practical UML-Based Rigorous Development Methods*, Lecture Notes in Informatics (LNI), pages 208–223. Bonner Köllen Verlag, October 2001.

[98] Simon Pickin, Claude Jard, Thierry Jéron, Jean-Marc Jézéquel, and Yves Le Traon. Test synthesis from UML models of distributed software. *IEEE Trans. Software Eng.*, 33(4):252–269, 2007.

[99] Risto Pohjonen, Juha-Pekka Tolvanen, and MetaCase Consulting. Automated production of family members: Lessons learned. In *Proceedings of the Second International Workshop on Product Line Engineering-The Early Steps: Planning, Modeling, and Managing (PLEES'02)*, pages 49–57. Citeseer, 2002.

[100] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. *Electr. Notes Theor. Comput. Sci.*, 116:59–71, 2005.

[101] Lars Rabbe. Cio update: Post-mortem on the Skype outage, December 2010.

[102] US Inc. Rackspace. Rackspace: The leader in hybrid cloud, 2014.

[103] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[104] Sean C. Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. Opendht: a public dht service and its uses. In Roch Guérin, Ramesh Govindan, and Greg Minshall, editors, *SIGCOMM*, pages 73–84. ACM, 2005.

[105] Marc Roper. Computer aided software testing using genetic algorithms. *10th International Quality Week*, 1997.

[106] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, nov 2001.

[107] Davide Di Ruscio, Richard F. Paige, and Alfonso Pierantonio. Guest editorial to the special issue on success stories in model driven engineering. *Science of Computer Programming*, (0):–, 2013.

[108] Ina Schieferdecker, Mang Li, and Andreas Hoffmann. Conformance testing of tina service components - the ttcn/ corba gateway. In *IS&N*, pages 393–408, 1998.

[109] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[110] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.

[111] Herry Imanta Sitepu, Carmadi Machbub, Armein Z. R. Langi, and Suhono Harso Supangkat. Unohop: Efficient distributed hash table with o(1) lookup performance. In Johnson I. Agbinya, Elmarie Biermann, Yskandar Hamam, Ntsibane Ntlatlapa, and Keith Ferguson, editors, *BroadCom*, pages 76–81. IEEE Computer Society, 2008.

[112] Jim Steel, Robin Drogemuller, and Bianca Toth. Model interoperability in building information modelling. *Software & Systems Modeling*, 11(1):99–109, 2012.

[113] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[114] Anthony Sulistio, Chee Shin Yeo, and Rajkumar Buyya. A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools. *Software: Practice and Experience*, 34(7):653–673, 2004.

[115] Inc. Sun Microsystems. *JXTA Specification*, 2001.

[116] Gerson Sunyé, Eduardo Cunha de Almeida, Yves Le Traon, Benoit Baudry, and Jean-Marc Jézéquel. Model-based testing of global properties on large-scale distributed systems. *Information & Software Technology*, 56(7):749–762, 2014.

[117] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms (2. ed.)*. Pearson Education, 2007.

[118] The Apache Software Foundation. Herriot, Large-scale Automated Test Framework, 2011.

[119] Adrien Thiery, Thomas Cerqueus, Christina Thorpe, Gerson Sunyé, and John Murphy. A DSL for deployment and testing in the cloud. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA*, pages 376–382. IEEE Computer Society, 2014.

[120] Thomas Thüm, Don S. Batory, and Christian Kästner. Reasoning about edits to feature models. In *ICSE*, pages 254–264. IEEE, 2009.

[121] Andreas Ulrich and Hartmut Konig. Architectures for testing distributed systems. In *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems*, pages 93–108, Deventer, The Netherlands, 1999. Kluwer, B.V.

[122] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In Jan Vitek and Christian F. Tschudin, editors, *Mobile Object Systems*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 1996.

[123] Yanyan Wang, Matthew J. Rutherford, Antonio Carzaniga, and Alexander L. Wolf. Automating experimentation on distributed testbeds. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *ASE*, pages 164–173. ACM, 2005.

[124] Alison Lachut Watkins. The automatic generation of test data using genetic algorithms. In *Proceedings of the 4th Software Quality Conference*, volume 2, pages 300–309, 1995.

[125] Jun Xu, A. Kumar, and Xingxing Yu. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. *IEEE J.Sel. A. Commun.*, 22(1):151–163, September 2006.

[126] Tewfik Ziadi and Jean-Marc Jézéquel. Software product line engineering with the UML: Deriving products. In Timo Käkölä and Juan C. Dueñas, editors, *Software Product Lines*, pages 557–588. Springer, 2006.