Chair of
Software Engineering

Distributed Testing Sessions for AutoTest

Master Thesis

2014

Victorien Elvinger

Option Systèmes d'Information et Réseaux, École Supérieure des Sciences et Technologies de l'Ingénieur de Nancy, Université de Lorraine.
2, rue jean l'amour, 54519 Vandoeuvre, France

Master Services, Sécurité des Systèmes et des Réseaux, Université de Lorraine.
BP 70239 - 54506 Vandoeuvre-Lès-Nancy CEDEX, France

Supervised by the Chair of Software Engineering, ETH ZUrich.
Clausiusstrasse 59, 8092 Zurich, Switzerland

Supervisors	**Dr. Chris Poskitt** **Alexey Kolesnichenko** **Yu Pei**
Chair	**Prof. Dr. Bertrand Meyer**
University tutor	**Dr. Pascal Urso**

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

UNIVERSITÉ DE LORRAINE

Abstract

AutoTest is an automatic testing framework for Eiffel programming language. It exploits the presence of contracts - executable preconditions, post-conditions and invariants - to automate the entire testing process, variously using them as built-in filters for test inputs and oracles for test execution. It demonstrated its efficiency by automatically unearthing previously undetected bugs in production-level libraries.

Despite its merits, a possible factor in limiting its effective use is the amount of time required to test a program. The large computational power offered by Cloud Computing could be capitalized to improve the efficiency of AuoTest and others black-box testing frameworks with minimal changes into their architecture.

The present thesis explores the launching of multiple testing sessions of AutoTest over the Cloud. A central question is: Is the discovery of software faults scalable? A specially designed prototype attempts to answer to this question.

Keywords: Distributed software testing, Cloud Computing, AutoTest

Résumé

AutoTest est un framework de tests automatiques pour le language de programmation Eiffel. Il prend avantage de la présence de contrats dans les programmes (préconditions, postconditions et invariants exécutables) pour automatiser l'ensemble du processus de test. Il utilise les contrats à la fois comme filtres d'entrées, et comme oracles. Il démontra son efficacité en détectant automatiquement des bugs dans plusieurs bibliothèques de production.

Malgré ses mérites, le temps nécessaire pour tester un programme est un facteur potentiel de limitation de son efficacité. La puissance de calcul offerte par le Cloud Computing pourrait être capitalisée pour améliorer l'efficacité de AuoTest et des frameworks de tests similaires, avec des changements minuers de leur architecture.

La présente thèse explore le lancement de sessions de tests multiples sur le Cloud. Une question centrale est: la découverte de fautes logicielles est-elle scalable? Un prototype a été spécialement conçu pour tenter de répondre à cette question.

Mots-clefs : Distributed software testing, Cloud Computing, AutoTest

Acknowledgments

I would like to express my gratitude to Prof. Dr. Bertrand Meyer for giving me the opportunity to realize my master thesis at the Chair of Software Engineering.

I would also like to thank Chris Poskitt, Alexey Kolesnichenko, and Pei Yu for giving me their complete confidence and valuable advices.

I also thank Dr. Pascal Urso for a warm welcome at Loria and for his support during my work.

I appreciated the technical help from Mischael Schill, and Emmanuel Stapf, Lead Engineer at Eiffel Software.

I cannot forget to mention Jocelyn Fiat for offering me a technical help and to put me into contact with Prof. Dr. Bertrand Meyer.

I extend these thanks to all those with whom I was in contact at ETH Zurich and at Loria.

Contents

Abstract i

Acknowledgments iii

1 Introduction 1

1.1 Motivation 1

1.2 Contribution 2

2 Background 3

2.1 AutoTest 3

2.1.1 Black-box testing 3

2.1.2 Contacts as oracles 3

2.1.3 Architecture 5

2.2 Cloud Computing 7

2.2.1 Service models 7

2.2.2 Cloud providers 8

3 Explored solution 9

3.1 Direction choice 9

3.2 Logical architecture 10

3.3 Scenario 11

3.4 Discussions . . . 12

3.4.1 Separated compilation . . . 12

3.4.2 Input diversity . . . 13

3.5 Criteria of assessment . . . 13

3.5.1 Identity of faults . . . 13

3.5.2 origin of faults . . . 14

3.6 Related work . . . 14

4 Assessment **15**

4.1 Implementation . . . 15

4.1.1 Cloud platform . . . 15

4.1.2 Effective architecture . . . 15

4.2 Benchmarks . . . 16

4.2.1 Parameters . . . 16

4.2.2 Outcomes . . . 17

4.2.3 Threats to validity . . . 19

5 Conclusion **21**

5.1 Future work . . . 21

References **23**

Chapter 1

Introduction

Nowadays software testing is a central part of a software development process. It enhances the quality and the reliability of programs. Software testing represents 40% to 50% of software development costs [16]. However it adds no value to the product in terms of functionality. Thereby its costs should be minimized.

Although software testing takes up an important place in enhancing software correctness, it is still mainly a manual or partially automated task. Some frameworks, such as AutoTest[1], propose a fully automated testing. They randomly generate test-cases and run them against the specification wrote by the developers.

1.1 Motivation

AutoTest has demonstrated its efficiency by automatically unearthing previously undetected bugs in production-level libraries [1, 7]. Despite the merits of the system a possible factor in limiting its effective use is the amount of time required to test a program.

The large-scale distribution facilitated by Cloud Computing already addresses performance issues in several software areas. In this context it could be used to setup multiple testing sessions executing in parallel across multiple computational resources.

Most of existing distributed testing approaches are not applicable for AutoTest. Indeed these testing frameworks exploit the source code for testing while AutoTest and other black-box testing frameworks have no access to the code under testing.

[1] http://se.inf.ethz.ch/research/autotest/

The aim of this thesis is to select a direction of exploration for capitalizing Cloud facilities with AutoTest and more generally black-box testing frameworks, in order to find faults faster.

1.2 Contribution

The present thesis attempts to learn if the discovering of more faults is possible using Cloud Computing facilities with minimal architecture changes on the black-box testing frameworks, in particular AutoTest.

The designed prototype relies on the parallel execution of multiple testing sessions. It is entirely orthogonal to AutoTest. As such, with minimal changes, it can be used for other black-box testing frameworks.

Chapter 2

Background

2.1 AutoTest

AutoTest is an automatic black-box testing framework for the Eiffel programming language. It is a part of Eiffel Verification Environment (EVE)[1], a research version of EiffelStudio[2], the Eiffel Software Integrated Development Environment (IDE). AutoTest takes advantage of Eiffel paradigms, mainly Design by Contracts (DbC), to automate the entire testing process.

2.1.1 Black-box testing

There exists two types of testing methods, the black-box testing and the white-box testing. The difference is in the possessed knowledge. The white-box testing method can exploit the internal implementation details. While the black-box testing method has no information about the internal details of the program. The advantage of the last one is the possibility to test binary parts of softwares.

Automatic testing frameworks can be separated according to the used method. The first ones use the internal program implementation for guiding input generation and the second ones use software specification if it exists.

2.1.2 Contacts as oracles

Design by Contracts (DbC) is a programming paradigm which considers the construction of software systems as the implementation of contractual relations be-

[1] http://se.inf.ethz.ch/research/eve/
[2] https://www.eiffel.com/eiffelstudio/

tween modules [11, 12]. These relations are expressed in terms of runnable assertions which annotate classes, and routines - functions and procedures.

An annotated routine explicits the requirements or preconditions that its callers - clients - must fulfill for benefiting of its guarantees or post-conditions. Preconditions and post-conditions form a routine contract between the routine and its clients.

```
square_root (n: INTEGER): REAL
        require
                n >= 0
        ensure
                Result >= 0
                n = Result * Result
```

Figure 2.1: Contract example in Eiffel

Figure 2.1 shows a typical routine interface in Eiffel. Its precondition specifies that the integer input must be a natural. If it is the case then the routine - the supplier - ensures that the result is a natural and then the input is equal to the multiplication of the result by itself.

```
class
    PERSON

feature

    age: INTEGER

invariant
    age >= 0
end
```

Figure 2.2: CLass invariant example in Eiffel

Class invariants allow to express assertions characterizing the created instances from the considered class. For example all instances of the class PERSON in Figure 2.2 have a positive age.

AutoTest relies on the presence of contracts to automate the entire testing process [5]. It uses preconditions as built-in input filters. Post-conditions and invariants act as oracles. If the preconditions are not hold then the input is not conforming, the test-case is invalid. Otherwise the test is executed and is either

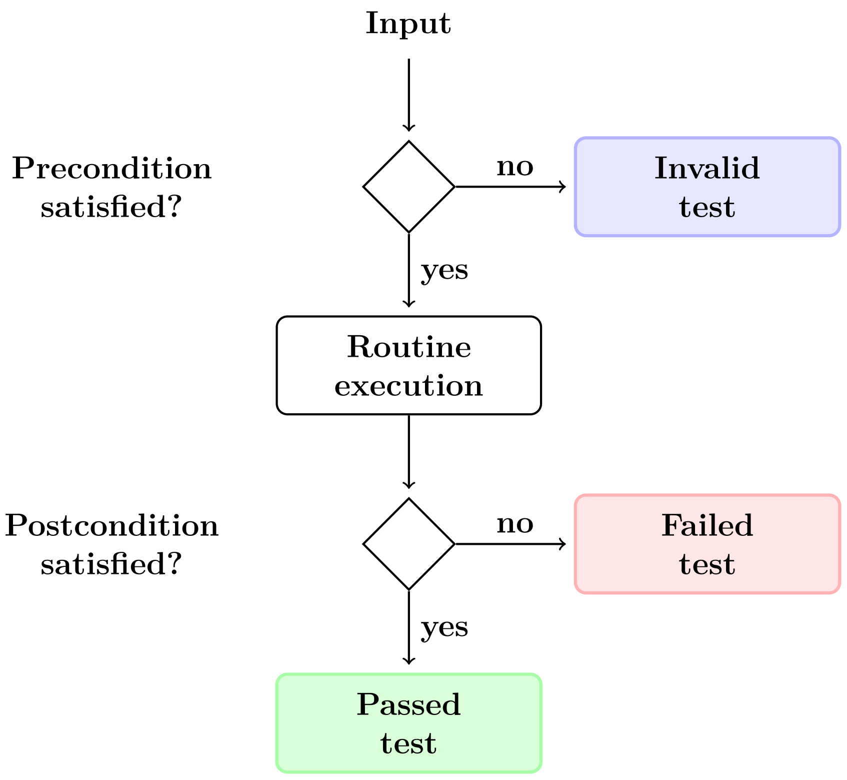

Figure 2.3: Contract-based testing

passed or failed according to if the post-conditions and the invariants are respected or violated (Figure 2.3).

For example, a negative integer is not included in the input domain of the square root routine. Enter this one as routine input raises a precondition violation. A test-case using such input is then an invalid one.

AutoTest has no information about the implementation details of the routine. This characteristic makes AutoTest as a black-box testing framework.

2.1.3 Architecture

AutoTest is based on a two-process model, the interpreter process and the master process [2]. The first one is in charge of test executions and the update of the pool of test inputs. The master process includes a proxy to communicate with the interpreter, a manager of test outcomes - the oracle - and a test-case generation strategy (Figure 2.4).

The testing framework allows to use several testing generation strategies. A generation strategy decides how to generate a test-case. It implements a selecting

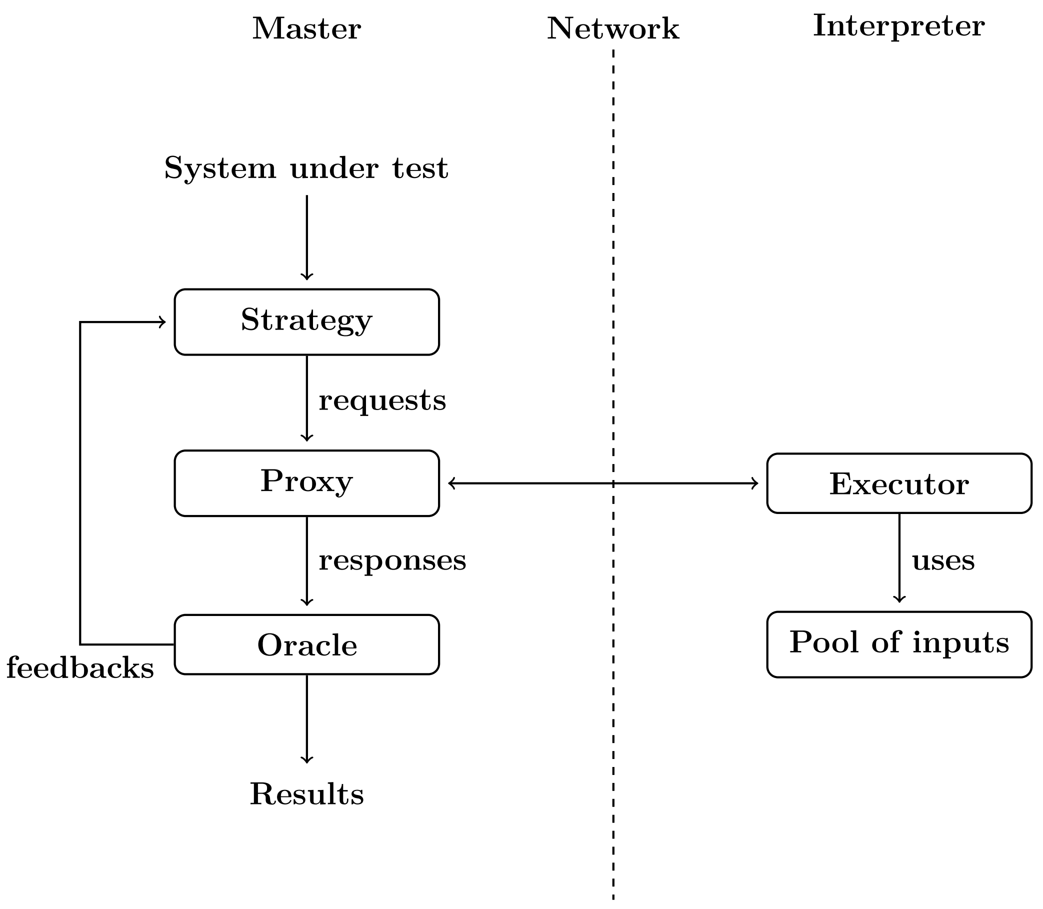

Figure 2.4: AutoTest Architecture

algorithm of inputs and routines to test.

The original AutoTest strategy selects randomly objects among the pool of inputs and generates new inputs with a predefined probability. The routines have a static and dynamic priority level for testing. The dynamic priority value allows to test uniformly the routines under testing.

The created data during a testing session join the pool of inputs. This allow to improve the diversity of the pool.

For example, if the strategy decides to test the square root routine specified in the Figure 2.1, it needs for an integer input. The input can be created or selected among the pool of inputs. Once an input is obtained, a test-case is created making a call to the square root routine with the considered input as actual argument. If the precondition has not passed the test-case is declared invalid, otherwise the routine is executed. If the post-condition is held then it is a passing test-case or it is a failing one. In the last case the failing status signals a potential fault.

Smarter strategies were designed [18, 17, 8]. Although the fault catching was improved or the test of routines quipped of strong preconditions was possible, theses strategies are not used systematically due to performance issues. The original strategy is simple and efficient.

2.2 Cloud Computing

Cloud computing [13] is a recent computing paradigm for delivering and managing a service over the Internet. It provides on-demand access to a shared pool of configurable resources, such virtual networks or computational unit. The pool can be dynamically released and scaled in order to reach the computational needs with a minimal management effort.

Cloud solutions allow an access to a large amount of computational power and to pay only for the resources actually used. It is particularly convenient for testing since software testing requires a large amount of power for a short period of time.

Cloud computing is also interesting in an ecologic viewpoint since multiple organizations and users share the same resources.

2.2.1 Service models

Cloud provider offers basically three levels of management [14] (Figure 2.5):

Software as a Service (SaaS) allows to deploy a turnkey application. The application can be directly used by the developers or end-users. This model saves costs on the maintenance part. Google Apps[3] is an example.

Platform as a Service (PaaS) is intended to setup a specific application using an available execution environment. The customer doesn't need to purchase expensive hardware or applications. For example Amazon Web Services (AWS) Elastic Beanstalk[4] and Heroku[5] provides a such service.

Infrastructure as a Service (IaaS) is a service model more permissive than the last one. It allows to choose the Operating System and generally create its own images (instance model) from basic instances configured with its needs.

[3] http://www.google.fr/intx/fr/enterprise/apps/business/
[4] http://aws.amazon.com/fr/elasticbeanstalk/
[5] https://www.heroku.com/

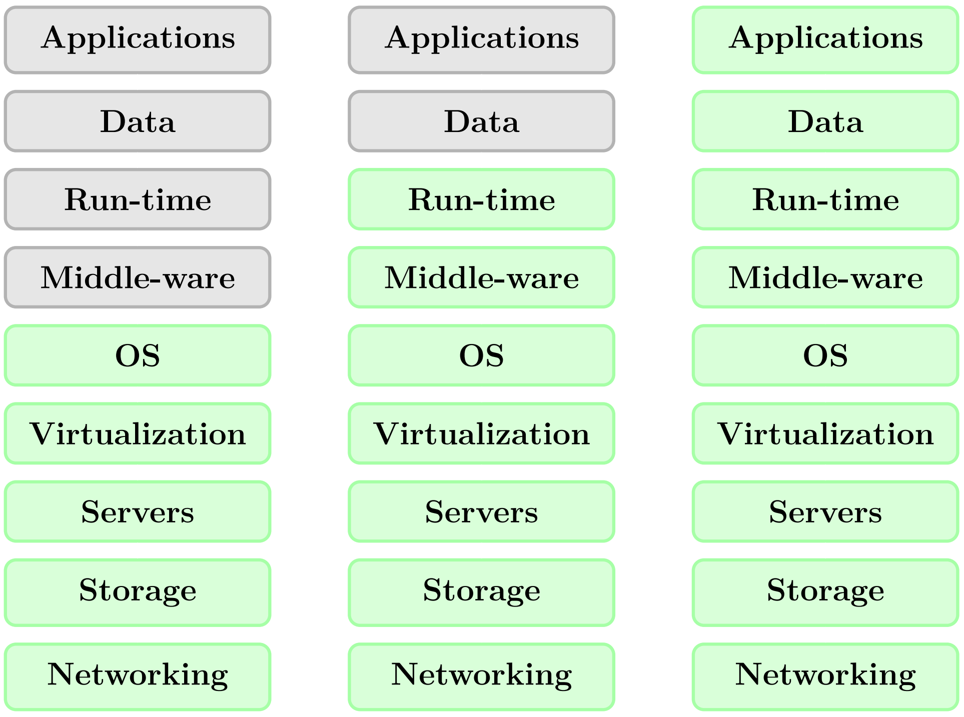

Figure 2.5: IaaS, PaaS, and SaaS

2.2.2 Cloud providers

Today, there exists many Cloud providers. The most delivered service model is SaaS. Some of these providers support the IaaS model and further service useful for designing an application. For example AWS[6], Google App Engine[7], and Microsoft Azure[8] offer Cloud storage, and more.

The cited providers are pretty similar although AWS offers extra services. It is possible to launch both Linux and Windows instances, an image can be created from instances and then serve as a template for next instance creation.

The difference appears in the interaction mode. While AWS and Google App Engine deliver Software Development Kit (SDK) for several language, Microsoft Azure proposes also a command-line interface.

Provider	**AWS**	**App Engine**	**Azure**
Interface	mainstream languages SDK	Java & Python SDK	Command-line
Free-use	educational grant	Free quotas	30-months trial

Table 2.1: Cloud provider comparison

[6] http://aws.amazon.com/fr/
[7] https://developers.google.com/appengine/
[8] http://azure.microsoft.com/fr-fr/

Chapter 3

Explored solution

3.1 Direction choice

AutoTest has evolved over several years. Now it has a large and historical code base (Table 3.1). Turning its architecture on a high parallelable architecture requires work longer than a master thesis period. It could involve to design from scratch the framework. The adopted solution should be reduced to a minimal architecture change with pretty good benefits.

Classes	256
Routines	24 829
Lines of code	68 248
Routines per class	96
used libraries	30

Table 3.1: AutoTest metrics

Basically two approaches are possible. The first one consists of parallelizing test-case executions. This solution requires sharing of the pool of inputs. The second approach consists of running multiple testing sessions and merging the results on the distributed session end.

A noticeable advantage of the second solution is its orthogonality. Indeed no architecture change is needed on the considered framework. Therefore the solution is suitable for any automatic black-box testing framework. In addition this approach allows to re-factor AutoTest without disturbing the current work.

The multiple testing sessions is the retained solution for assessment.

3.2 Logical architecture

The proposed architecture is designed to be adapted for future changes or reaching new needs. In particular adding a support for a new cloud platform is simple. This objective is ensured through 5 logical components exhibited in Figure 3.1.

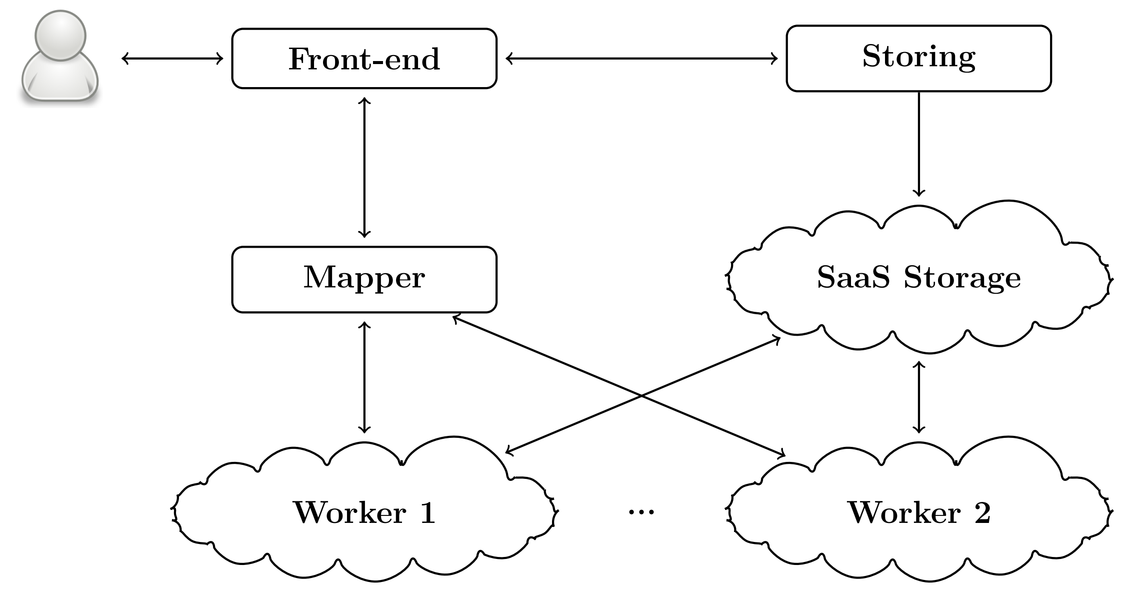

Figure 3.1: Logical architecture of the prototype

The Front-end directly interacts with the user. It invokes services of both Mapper and Storing components to satisfy the user requests. The Front-end is capable of ordering to the Mapper the testing of the same class cluster under several testing sessions or the mapping of multiple class clusters to multiple testing sessions. A class cluster is a class group ensured to be tested together over one or several workers.

The Mapper is responsible for launching, monitoring, and discarding the workers. A worker is a cloud instance or virtual machine. The Mapper or the machine where it is located needs some information, such as authentication, for leading Cloud services. Then, it is a sensitive security point of the structure. It is for this reason that the architecture enables the Mapper to be placed on a different machine that the user one, for example a server. Beyond the security advantage, setup the Mapper on a server enables to share a single Cloud provider account with more than one user without spreading its confidential authenticating information.

Each worker has an installation of EVE and then of AutoTest. A service is also present, making some preparation before and after the testing session of the worker. In particular, it retrieves the project under test from any storage reachable over the Internet, it compiles the project, runs AutoTest, and finally stores testing results in a Cloud storage facility. Since EVE is a single unit application the IaaS model is required.

The Storing component is in charge to retrieve the results stored by each workers and to merge them. It can also be used to store the project in a SaaS storage facility. In this case the user must inform the Storing from its confidential authenticating information.

3.3 Scenario

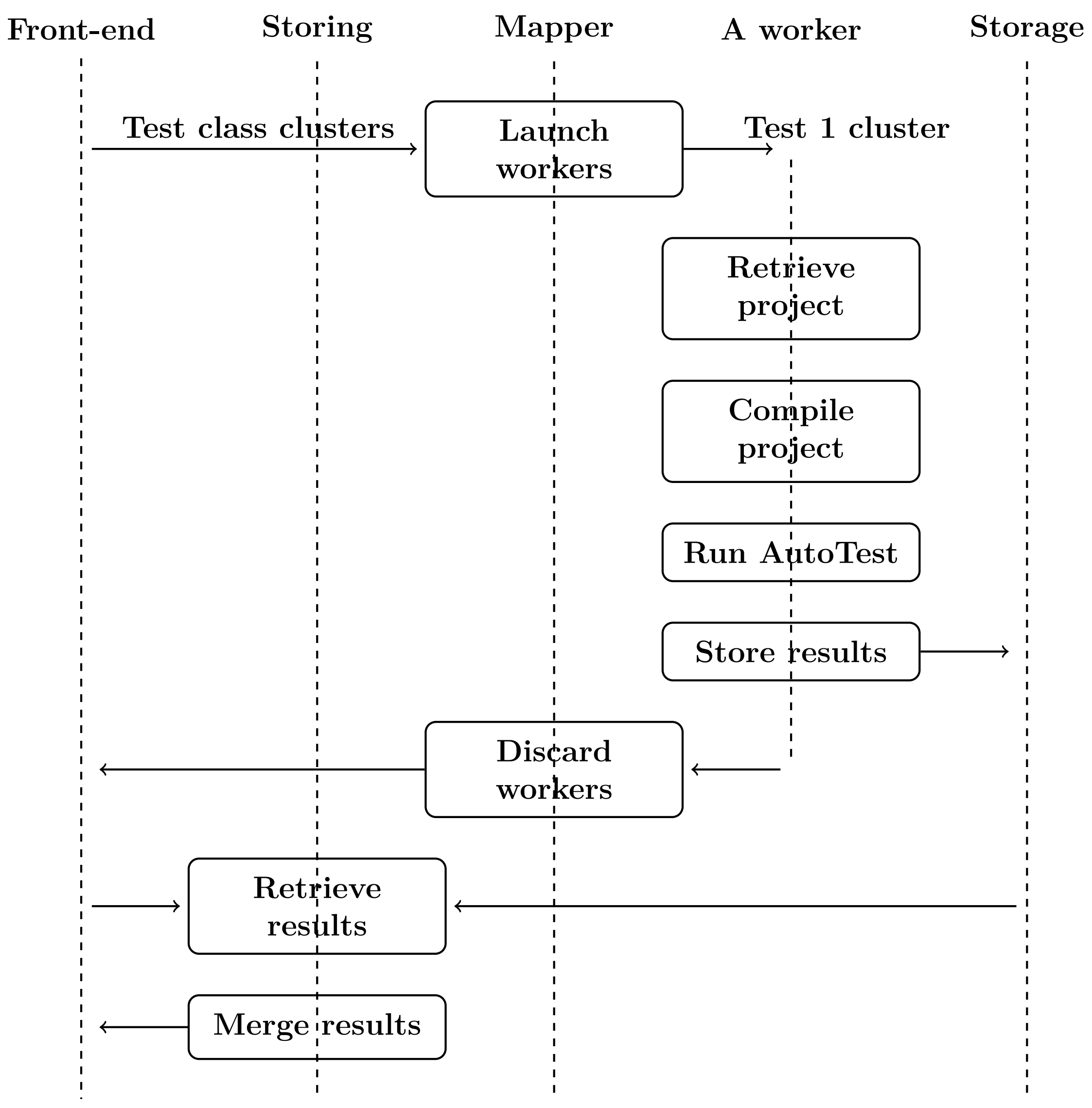

Figure 3.2: Typical distributed testing session

A typical distributed testing session is represented in the Figure 3.2. The project storing step is not visible. This step is optional since any storing facility reachable over Internet can be used. Only the source code of the project is required.

User informs Front-end about the project location, the time for testing and the class clusters. If the project location is local then it solicits the Storing component to make the project reachable over Internet.

Front-end relays the user request to Mapper. Mapper launches as many workers as class clusters under testing. Once all the launched workers are reachable, Mapper distributes the testing effort to the worker pool. One cluster is assigned for each worker.

The service on each worker retrieves and compiles the project before running a usual AutoTest session. When testing results are available the system stores it into the SaaS storage and notifies Mapper.

Mapper discards the worker just after it has been notified. When all workers are discarded or when a specific time threshold is over-passed, Mapper notifies the Front-end with respectively the location of the overall testing results, or an error.

The Front-end invokes Storing for retrieving and merging the last results.

3.4 Discussions

3.4.1 Separated compilation

In the proposed architecture each worker compiles the same project. A more optimised architecture could separate compilation and testing tasks. In this alternative architecture a dedicated worker is allocated to the project compilation. Once the project compiled, it is mapped on the testing workers.

The mapping step has basically two implementations. The first consists in storing the compiled project in a SaaS Storage and the second one in creating an image from the compiling worker and using it as template for the creation of dedicated testing workers.

This alternative architecture could be more efficient for large projects. It allows to optimize the instance type according to the worker type. Indeed, testing and compiling could have different hardware needs.

The proposed and alternative architectures are not changing the evaluation of the solution.

3.4.2 Input diversity

Multiple testing sessions implies multiple pool of inputs. If the same cluster of classes is tested or if the universe of classes is not well-clustered, the diversity of the pool is reduced. Indeed if some routines of the first class use inputs with a type based on a second class, then it is interesting to gather the two classes in an identical cluster. The testing of the second class provides a diversity of inputs for the routines of the first class. A lack of diversity implies the discovery of repeatedly occurring faults.

To get round this decreasing in diversity, the island model [15] can be considered. It consists in exchanging some objects at regular intervals of time. This approach could solve the diversity issue. However it heavily complicates the architecture for maybe no real benefit.

3.5 Criteria of assessment

The prototype should determine if the proposed solution is scalable, in other words if the number of unique faults discovered increases with the number of running instances.

3.5.1 Identity of faults

AutoTest identifies a fault with the combination of the violated contract clause and test inputs. Taking or not in concern the inputs is a sensitive question. Different inputs can cover different paths of code and then produce different faults. However different inputs can also run the same path of code and then produce the same fault.

The optimist approach takes care of inputs and the pessimist one discard the inputs from the identity of the fault. The least damageable approach should be used. If the most pessimist measure reveals that the solution is scalable then it is also true in the optimist scenario.

For instance, let me take three inputs violating the same contract clause. The two first inputs take the same path of execution and the last one covers another code part. Since the three inputs are different the three violations are consisted as unique faults in the optimist approach. In contrast, the same contract clause is violated then it is considered as a single fault in the pessimist viewpoint. We will use the pessimist approach in the following.

3.5.2 origin of faults

A fault is characterised by the origin of its raising. If the fault appears in the routine under testing then it is a first-class fault, otherwise it is a second-class fault. Basically, all faults raised by a routine call in the routine under testing is a second-class fault.

When the fault is raised from a class out of the scope of testing then the fault is easily categorised as second-class fault. However a fault raised from a class under testing is either a first-class or second-class fault. The identity of the fault doesn't inform about it.

The following evaluation omits the second-class faults raised from a class out of the scope of testing.

3.6 Related work

The York Extensible Testing Infrastructure (YETI) is a language agnostic random testing tool. It is capable of generating random inputs and it uses exceptions and contracts if such exist as oracles.

YETI on the Cloud [10] is a Cloud-enabled version of YETI. This version uses a map-reduce algorithm [9] which basically map the command on a worker cluster. Indeed the command is divided such as each worker tests a class. The project under testing is uploaded on the distributed file system, making the project available for all the workers. After each session is finished the reduction step merges the log files and send it to the user.

The designed solution allows to reduce the time of execution, however no information is provided about fault discovery.

Chapter 4

Assessment

4.1 Implementation

4.1.1 Cloud platform

The choice of the Cloud provider is not so important in a prototype step, particularly if the architecture allows a simple implementation for adding support of other Cloud platforms.

The designing and the testing of the prototype should have a minimal cost. The Cloud platform choice was guided by the requirements for running EVE, the service cost and the comfort of development. EVE and by extension AutoTest has dependencies which depend on the OS used. In this case the adapted Cloud model is IaaS.

AWS was selected for the availability of free instances and of an educational grant.

4.1.2 Effective architecture

A library of reusable components was created to minimize the redundancy between the components. It gathers file helpers, communication facilities to improve object parsing and restoring, and common abstractions. Improvements were brought for the Eiffel JSON library[1].

In practice Storing and Mapper components are implemented together, in the same unit. They have a centralised interface with the Cloud services and share

[1] https://github.com/eiffelhub/json

the same code base. If the Mapper is on local then both Mapper and Storing components are the same running entity. In contrast, if the Mapper is on a separate server then they are two running entity of the same effective unit.

All components use objects representing requests and responses for communication. The visitor design pattern [3] is used in combination with a framework of Inversion of Control for decoupling totally the request and response domain of the Cloud platform details. If several Cloud platforms are supported then it is easily switchable.

The majority of the component parts take advantage of the Inversion of Control [4]. They can easily be replaced with others. Currently it needs a manual change in a centralized part. For example the current JSON parsing module could be replaced with a XML parsing module. These characteristics were not chosen only for better evolving but also for testing purpose.

The designed prototype is under GNU GPL licence.

Classes	144
Routines	670
Lines of code	6 979

Table 4.1: Project metrics

4.2 Benchmarks

4.2.1 Parameters

All testing sessions ran for the solution assessment used a dedicated AWS instance as a worker. The instance type used is m3.2xlarge. It has 8 virtual CPU, a memory of 30 Gio and a network access rated as high by AWS. Each worker is based on an image using Windows Server 2012 and the last revision of EVE relying on EiffelStudio[2] 14.05, the last release of the Eiffel Software IDE.

The solution assessment is based on a testing sessions launched for 60 minutes. Classes under testing came from EiffelBase 14.04[3], a production-level library of data-structures for Eiffel. The library has evolved over a number of years and today is the base of many Eiffel applications. Selected classes are commonly chosen for testing AutoTest efficiency. Their metrics and relations are respectively available in Table 4.2 and in Figure 4.1. Missing classes in Figure 4.1, have no relation with the others classes under testing.

[2] https://www.eiffel.com/eiffelstudio/

[3] https://docs.eiffel.com/book/solutions/eiffelbase

Class	Attributes	Routines	Contracts	Lines
ARRAY	4	103	180	891
LINKED_LIST_CURSOR	3	34	43	54
LINKED_LIST	6	111	161	753
ARRAYED_LIST_CURSOR	1	34	43	41
ARRAYED_LIST	3	135	209	853
HASH_TABLE_CURSOR	17	123	157	44
ARRAYED_QUEUE	4	73	84	464
Minimum	1	34	43	41
Average	5.43	87.57	125.29	442.86
Maximum	17	135	209	891

Table 4.2: Class metrics

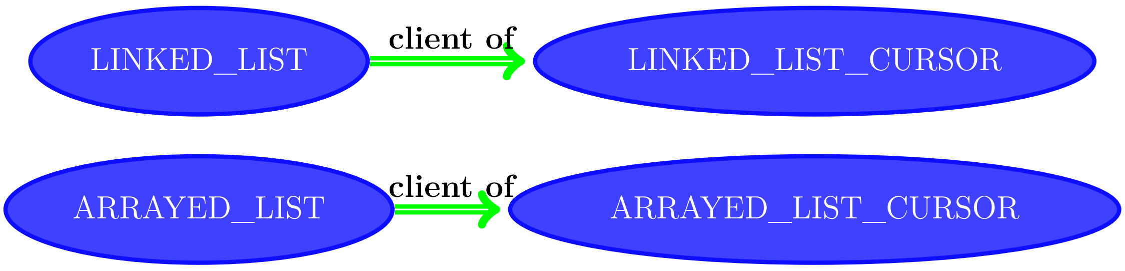

Figure 4.1: Class relations

4.2.2 Outcomes

Instances	Faults	Repeated faults	test-cases	Fault ratio
1	30	158.2	33 044.6	0.0908 %
5	38.2	897.2	15 9471.2	0.0240 %
10	40.4	1 827	319 030.8	0.0127 %
15	42.2	2 733.6	483 107	0.0087 %
20	44.2	3 672.8	641 841	0.0069 %
25	45.6	4 603.4	800 723.2	0.0057 %
30	46.5	5 640.25	486 642.25	0.0048 %

Table 4.3: Average of benchmarks

Table 4.3 presents 7 experiments. Each experiment has a different number of dedicated instances and was repeated 5 times. The results are the averages of its 5 repetitions. A fault denotes a unique fault in the pessimist approach (subsection 3.5.1). A repeated fault is a fault which already occurred. Test-cases denotes the count of passing, failing and invalid test-cases.

Figure 4.2 shows an increase in fault discovery. The number of discovered faults tends towards a plateau. This plateau is also observable on long-time sessions [6].

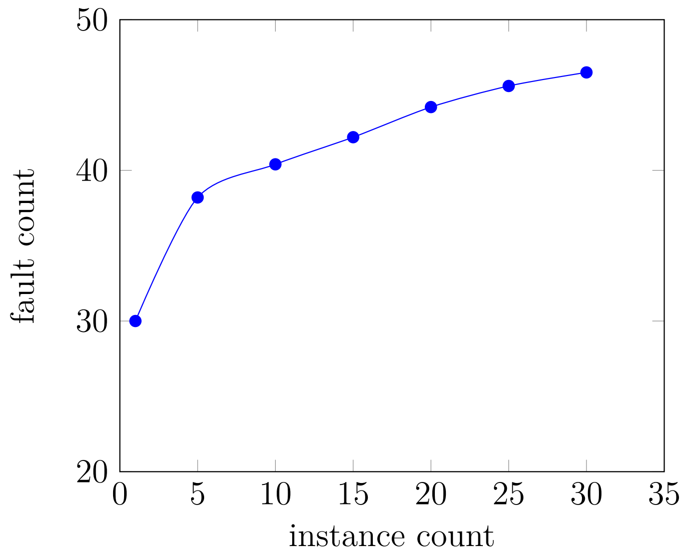

Figure 4.2: Average of faults found depending on the number of instances

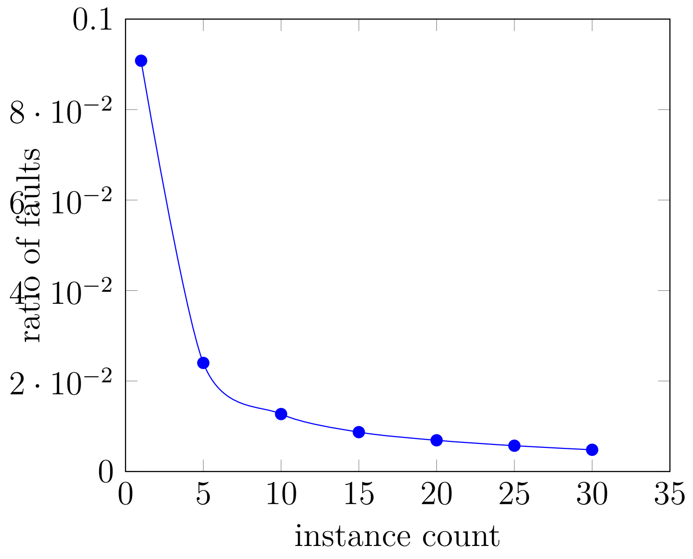

Figure 4.3: Proportion of faults found depending on the number of instances

However this increase in fault discovery is expensive in terms of resources. Indeed Figure 4.3 highlights the need of more and more resources for detecting a few new faults. For example, the efficiency between a single testing session and a distributed testing session of 10 instances is close of 7.4 %.

$$\frac{30}{(40.4*10)} = 7.4\%$$

4.2.3 Threats to validity

The tested classes come from a library of data-structures. Despite their diversity in terms of code metrics, their representativeness of software is limited.

Another threat to validity is the number of classes under testing. An ideal case would be the testing of entire libraries or softwares.

Chapter 5

Conclusion

The designed prototype enables a distributed testing session over the Cloud for AutoTest or other black-box testing frameworks. It is highly adaptable for future investigations. The evaluation of the adopted approach shows an increase in fault discovery but with an efficiency drawback.

However the testing of the same class cluster on several workers raises many repeatedly occurring faults. These occurrences of faults reduce the efficiency of the solution which could find out further faults with a smarter distribution.

5.1 Future work

How to divide the classes under testing in order to reduce the raising of repeatedly occurring faults remains an open research question. An automatic tool for class clustering could be added to the EVE ecosystem and delivers its services to the designed prototype.

Currently a distributed testing session of AutoTest delivers a feedback after testing. The application could be extended for real-time user feedbacks. This change will allow an integration in the current AutoTest tool.

The assessment could be extended to various testing times, in particular short times, in order to compare it with long-time and single testing sessions.

References

[1] Ilinca Ciupa Andreas Leitner Yi Wei Emmanuel Stapf Bertrand Meyer, Arno Fiva. Programs that test themselves, September 2009. IEEE Computer, Volume 42, Nunmber 9, pages 46-55.

[2] Ilinca Ciupa. Strategies for random contract-based testing, 2008. DISS. ETH NO. 18143.

[3] John Vlissides Ralph Johnson Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 31 octobre 1994. pages 113-135, chapter 6.

[4] Martin Fowler. Inversion of control containers and the dependency injection pattern, January 2004. http://www.martinfowler.com/articles/injection.html.

[5] Andreas Leitner Ilinca Ciupa. Automatic testing based on design by contract, 2005. 6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World.

[6] Andreas Leitner Manuel Oriol Bertrand Meyer Ilinca Ciupa, Alexander Pretschner. On the predictability of random tests for object-oriented software, 2008. First International Conference on Software Testing, Verification, and Validation, pages 72-91.

[7] Manuel Oriol Alexander Pretschner Ilinca Ciupa, Bertrand Meyer. Finding faults: Manual testing vs. random+ testing vs. user reports, 2008. 19th International Symposium on Software Reliability Engineering (ISSRE), pages 157-166.

[8] Manuel Oriol Bertrand Meyer Ilinca Ciupa, Andreas Leitner. Artoo: adaptive random testing for object-oriented software, 2008. Proc. Internatioanl Conference on Software Engineering (ICSE), pages 71-80.

[9] Sanjay Ghemawat Jeffrey Dean. Mapreduce: Simplified data processing on large clusters, 2008. Communications of the ACM - 50th anniversary issue: 1958 - 2008, Volume 51 Issue 1, pages 107-113.

[10] Faheem Ullah Manuel Oriol. Yeti on the cloud, 2010. IEEE Software Testing, Verification, and Validation Workshops (ICSTW), pages 434-437.

[11] Bertrand Meyer. *Object-Oriented Software Construction 2nd edition.* Prentice Hall, 2 edition, 1997.

[12] Bertrand Meyer. *Touch of Class, Learning to program well with Objects and Contracts.* Springer, 2009.

[13] Timothy Grance Petter Mell. The nist definition of cloud computing, 2009. National Institute of Standards and Technology (NIST), Special Publication 800-145.

[14] Ian Lumb Rimal Bhaskar Prasad, Eunmi Choi. A taxonomy and survey of cloud computing systems, 2009. NCM, pages 44-51.

[15] Tauhida Parveen Scott Tilley. *Software Testing in the Clloud.* IGI Global, 2012.

[16] Kuo-Chung Tai. Program testing complexity and test criteria, November 1980. IEEE Transactions on Software Engineering, Volume SE - 6, Number 6, pages 531-538.

[17] Yi Wei. Putting contracts to work for better automated testing and fixing, 2012. DISS. ETH NO.20861.

[18] Bertrand Meyer Manuel Oriol Yi Wei, Serge Gebhardt. Satisfying test preconditions through guided object selection, 6-10 April 2010. Proc. Internatioanl Conference on Software Engineering (ICSE), pages 303-312.