Guided Random-Based Testing Strategies
Diploma Thesis

Cosmin Mitran

Faculty of Automation and Computer Science
Technical University of Cluj-Napoca

Supervisors:
Ilinca Ciupa
Prof. Dr. Bertrand Meyer

Chair of Software Engineering
ETH Zurich

June 11, 2007
Acknowledgements

There are many people who had contributions to this project. First of all, I would like to thank Ilinca Ciupa for supervising my diploma thesis. From the first minute we discussed about this project I received valuable information, guidance and feedback from her. She was a great help in developing both the project and the current report, supporting me with very helpful ideas, suggestions and comments.

I want to thank Prof. Dr. Iosif Ignat for offering me the opportunity to develop my diploma thesis at ETH Zurich. My deep gratitude goes to Prof. Dr. Bertrand Meyer, for accepting me in his research group from ETH Zurich. Being able to work under his supervision and to attend his lectures and talks has been an honor for me.

My appreciation also goes to the developers of AutoTest and its predecessors, especially to Andreas Leitner. I would also like to thank all the members of the Chair of Software Engineering at ETH Zurich, for their support and for everything I have learnt from them. Working in this group has been a real pleasure for me. Special thanks go to my office colleagues Michele Croci and Stefan Sieber, for being not just colleagues, but also friends.

I am deeply thankful to my family and to Adela, for their love and for supporting my decisions.
Abstract

Software testing is no longer considered a peripheral activity during the software development process. Its importance is widely recognized and researchers invest a lot of efforts in order to make testing less tedious and more effective. Despite this, progresses in this field are not yet advanced enough to offer software developers satisfactory solutions for their testing needs.

Manual and automated testing are the two complementary ways in which software can be tested. While manual tests are built according to the testing engineer’s knowledge about the system under test, automated testing relies on the computation power of a machine for performing many more tests than a human could in a limited period of time. So, it is desirable to benefit from the advantages of both methods in order to test any software system effectively and intensively. In spite of this fact, manual and automatic testing tools are usually different, often making the testing process laborious and hard to manage.

*AutoTest* is a testing tool that reconciles the two approaches: it permits both manual and automated tests inside the same testing process; furthermore, it generates, compiles and runs tests on the push of a button. It relies on the principles of *Design by Contract* for assessing whether a test passes or not. *AutoTest* has found bugs in several contract equipped Eiffel libraries, proving itself to be a valid solution for the challenge of testing.

The main goal of the current project was to implement new strategies for making the automatic testing process more accurate and effective, trying to take advantage of the user’s knowledge contained in manual tests. The automatic tests attempt to reproduce as much as possible the distribution of inputs for the manual tests, while selecting representative test cases from the whole input space. Automatic tests need to be diverse in order to be effective. They also need to be close to the manual tests, which contain information about the system under test. Our strategies try to reconcile these two apparently opposite requirements.

Another goal of the current work was to provide the user with various statistics about the testing process: classifications of the causes that lead to failures; the number of failure triggering lines found in the tested code; the amount of time elapsed until the first failure is triggered. Gathered statistics are displayed in an easy to read format.
# Contents

1 Introduction ................................. 5

2 Main results ................................. 7

3 Theoretical background ..................... 8
   3.1 The Eiffel programming language .............. 8
   3.2 Design by Contract .......................... 10
      3.2.1 Overview ................................ 10
      3.2.2 Eiffel Contracts ......................... 11
      3.2.3 Exception Handling ....................... 18
   3.3 Software testing ........................... 19
      3.3.1 Terminology .............................. 19
      3.3.2 Testing in software engineering .......... 20
      3.3.3 The limits of testing .................... 22
      3.3.4 The testing process ...................... 22
      3.3.5 Testing strategies ....................... 23
      3.3.6 Testing based on Design by Contract \(\text{TM}\) .......... 24
      3.3.7 Related work ............................ 25

4 Random based testing strategies ............ 27
   4.1 Pure random testing ...................... 27
   4.2 Adaptive random testing ................. 28
   4.3 Object distance ........................... 30
      4.3.1 Elementary distance .................... 31
      4.3.2 Type distance .......................... 31
      4.3.3 Composite object distance ............... 32

5 AutoTest - a push-button testing tool ..... 34
   5.1 Overview ................................... 34
   5.2 Functionality .............................. 35
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Architecture</td>
<td>39</td>
</tr>
<tr>
<td>5.4</td>
<td>Structure</td>
<td>42</td>
</tr>
<tr>
<td>5.4.1</td>
<td>The root class of AutoTest</td>
<td>42</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Integrating testing strategies in AutoTest</td>
<td>44</td>
</tr>
<tr>
<td>5.4.3</td>
<td>The interpreter</td>
<td>46</td>
</tr>
<tr>
<td>5.4.4</td>
<td>The communication between AutoTest and the interpreter</td>
<td>47</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Generating the statistics</td>
<td>49</td>
</tr>
<tr>
<td>5.5</td>
<td>System usage</td>
<td>49</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Requirements</td>
<td>49</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Starting the application</td>
<td>50</td>
</tr>
<tr>
<td>5.5.3</td>
<td>ACE Files</td>
<td>51</td>
</tr>
<tr>
<td>6</td>
<td>Implementation of the testing strategies for AutoTest</td>
<td>54</td>
</tr>
<tr>
<td>6.1</td>
<td>The support for manual testing in AutoTest</td>
<td>55</td>
</tr>
<tr>
<td>6.2</td>
<td>Implementation of pure random testing in AutoTest</td>
<td>56</td>
</tr>
<tr>
<td>6.3</td>
<td>The previous implementation of ART in AutoTest</td>
<td>59</td>
</tr>
<tr>
<td>6.4</td>
<td>The new implementation of ART</td>
<td>59</td>
</tr>
<tr>
<td>6.5</td>
<td>The implementation of the new ART strategy when the manual objects</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>are considered</td>
<td></td>
</tr>
<tr>
<td>6.5.1</td>
<td>Serializing the manual objects into files</td>
<td>65</td>
</tr>
<tr>
<td>6.5.2</td>
<td>The algorithm implementation</td>
<td>66</td>
</tr>
<tr>
<td>7</td>
<td>Adding more statistics to AutoTest</td>
<td>71</td>
</tr>
<tr>
<td>7.1</td>
<td>Counting the number of faulty lines in the tested code</td>
<td>71</td>
</tr>
<tr>
<td>7.2</td>
<td>Showing the time elapsed until the first failure was triggered</td>
<td>73</td>
</tr>
<tr>
<td>7.3</td>
<td>Counting the number of faults of different categories</td>
<td>73</td>
</tr>
<tr>
<td>7.4</td>
<td>Creating a “comma-separated-values” file with the results</td>
<td>74</td>
</tr>
<tr>
<td>8</td>
<td>Results and conclusions</td>
<td>75</td>
</tr>
<tr>
<td>9</td>
<td>Directions for future work</td>
<td>79</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

As in other engineering fields, a software engineer’s main task is to produce quality products. When we talk about the quality of a software system we must talk about the external and internal quality factors that characterize the considered system. The external quality factors (like runtime performance, reliability, ease of use, compatibility, portability and so on) are those perceptible by the user, while the internal quality factors (like modularity, readability and so on) are those perceptible by designers and implementors who have access to the software code. Generally, the external factors are all that matter, but in achieving them, the internal factors definitely have the most important contribution.

It is often necessary to make various trade-offs between these quality factors, but one of them is intangible, and compromising it is not justified whatever the benefit. This special quality factor is correctness. It is the most important quality factor, and it is defined as “the ability of software products to perform their exact tasks, as defined by their specification” [11]. This is the most important requirement a customer has, and a software developer should never sacrifice this factor in favor of any other. As stated in the definition, correctness is a relative notion. It doesn’t characterize a software module by itself, but only according to its specification.

The main methods for checking the correctness of software are validation and verification. Validation is the process of checking that the system produces the desired results (are we developing the right product?), while verification checks that the task of the system is fulfilled in a correct way (are we developing the product right?). These two processes are often reffered to under a common name: the V&V process. Testing is one of the most important activities of this process.

Testing means providing the system with inputs and letting it operate on them in order to prove that it works correctly. If it doesn’t, the purpose of testing is to show the differences between the obtained results and the expected ones. Such a difference
is caused by a fault (often referred to as “bug”) which must be exactly identified.

For finding even the most subtle bugs, testing must be thorough. The test cases must be chosen very carefully, and their execution should cover most of the existing code. To test the code intensively, we need to generate many test cases. This would be a very unpleasant and time consuming task if it were done by a person. Hence, the process of generating test cases, running them and assessing the obtained results should be automated.

**AutoTest** is a tool that automatically generates test cases, runs them and uses Eiffel contracts as oracles for assessing the obtained results. Contracts (preconditions, postconditions, class invariants, loop variants and invariants and check instructions) are a built-in mechanism in Eiffel and they contain a lot of information related to the intended semantics of the system. This information is a valuable resource exploited by AutoTest for assessing the failure or the success of executing a test case. “The way in which software respects its contracts ascertains its validity”. Therefore, the contracts constitute a solid basis for the automation of the testing process.

The key aspect here is that, even if testing is automated, it should be *guided* in some way. For increasing the probability of uncovering bugs in a certain time, the chosen test cases should be evenly spread in the space of all possible test cases. In the same time, it is desirable to try those tests that are more likely to uncover a bug. But the desirability of a test case can not be assessed by a computer if the user doesn’t give a clue related to what “desirable” means. In AutoTest this is done by providing it a set of manual tests. We have developed two new strategies for selecting test cases in AutoTest by which we try to guide the selection of test cases to achieve the two previously stated goals: covering most of the possible input space and immitating as much as possible the distribution of manual tests.

After performing tests for a limited period of time on a given system, AutoTest is able to build statistics and save them into an HTML file. The user can see there various information about the testing process for every tested class and feature of the system under test: number of total run tests, how many of them were a success, number of faulty lines in the tested code, main causes for the failures, test-cases and their results etc. Some of these statistics were added as a part of the current project.
Chapter 2

Main results

The main purpose of the current project was to find a way of integrating manual and automatic testing and to guide the test case selection process in such a way to improve the effectiveness of the testing process. Another purpose was to extend the statistics AutoTest builds after running the tests by making more information available to the user. We have extended the existing functionality of AutoTest, which previously used two strategies for selecting test cases: a Random Testing strategy and an Adaptive Random Testing strategy based on the concept of “distance between objects”.

The contributions of the current project are:

- Developing a new Adaptive Random Testing (ART) strategy.
- Developing a strategy that combines ART with the knowledge gained by the system from the manual tests.
- Improving the statistics built by AutoTest by also:
  - Counting the number of failure-triggering lines in the tested code.
  - Measuring the time from the beginning of the testing process until the first failure is triggered.
  - Classifying the failures according to their originating source.
  - For a given feature, counting the number of failures caused by faults from other features.
  - Printing various information from the gathered statistics in a “comma-separated-values” file.
Chapter 3

Theoretical background

This section focuses on presenting some theoretical concepts used during the development of the current project. It first makes a short introduction into Eiffel, which was the development language for the project. After that, it presents the concept named Design by Contract\textsuperscript{TM}. Finally, some notions about testing are introduced.

3.1 The Eiffel programming language

Eiffel is a pure object-oriented programming language, used both in the academic world - because of its suitability with the object oriented principles - and in the industry, as a development platform.

Eiffel was designed for being applied during the entire lifecycle of a software system, so as to “minimize the gaps between successive activities” \textsuperscript{11}. “The aim of Eiffel is to help specify, design, implement and modify quality software” \textsuperscript{10}, where by quality in software we mean a combination of the quality factors mentioned in chapter\textsuperscript{1} Eiffel integrates the Design by Contract\textsuperscript{TM} mechanisms - described below - with some object-oriented mechanisms like inheritance, genericity, exception handling and many others.

Any Eiffel system is a collection of classes which are organized in a decentralized architecture. Such a system contains a root class and all classes which the root class needs directly or indirectly, through one of the two possible relations between classes: inheritance or client. Execution of a system consists in creating an instance of the root class and calling its creation procedure. This creation procedure can contain instructions for object creation and feature calls on these objects.

In Eiffel, a class is considered to be both a module - “a unit of software decomposition” \textsuperscript{11}, a syntactic concept that is not related to the functionality of the software -, and a type - a semantic concept, a static description of the structure of objects created and manipulated at run-time by the software system. In a consistent object-oriented
approach, there is no need for another level of structuring software above classes. For easy management, some related classes can be grouped to form a cluster. Tipically, every class will be written in a separate file and a cluster will be the directory containing those related files. Grouping some classes into a cluster is just an organizational convention, and it has no influence over the visibility of a class. So, the role of a cluster is different than the one a package has in other programming languages.

A feature can be an attribute, if it is implemented by using a memory location containing some information related to an object, or a routine, if it is implemented by computation. Routines are further separated in functions and procedures, depending whether they return a result or not. Features can be grouped into feature clauses according to their role. The features grouped under the same feature clause can be selectively exported to all, none or just some specified classes.

For a client, all the features look the same, hence the Uniform Access Principle is respected. The principle says that a feature is accessed in the same way if it is implemented by memory or by computation. The client doesn’t have to know the way in which a feature is implemented.

An attribute of a class can be modified only from one of the routines of the enclos-
ing class. Eiffel emphasizes information hiding by requiring formal interfaces to data modification. Modifying an attribute only through certified routines is a guarantee that the new value of the attribute will not bring the class in an inconsistent state - meaning that it will not violate the class invariant, a notion described below.

Once routines are executed only the first time they are called, and the result of the execution is stored. The following calls of such a routine require non additional computation or resource allocation, but simply return the previously computed result.

Eiffel is a statically typed language that supports dynamic binding. By using dynamic binding, the written code is very elegant and flexible. Static typing is a sine qua non condition for ensuring reliability, readability and efficiency of a software code. Syntactic overloading is not allowed.

Mechanisms like genericity and inheritance are available for achieving the goals of extendibility and reusability. Genericity - which can be constrained or unconstrained - permits writing classes having a formal generic parameter. A generic class can be used only after its client provides it with the corresponding actual generic parameter, in a process called generic derivation.

Eiffel supports multiple inheritance. Hence, a class D can inherit from two different classes B and C that both share a common ancestor A. This situation is called repeated inheritance. In the case of repeated inheritance, mechanisms like renaming, redefinition, undefined and selection are used for dissambiguating between name clashes that may
occur.

All Eiffel classes have a common ancestor, the class ANY, and a common descendant, existing only in theory, the class NONE, whose only instance is Void. Conformance, covariance and descendant hiding are examples of mechanisms tightly coupled with inheritance.

Deferred features, and hence deferred classes, can be used for achieving abstraction. A deferred feature has no implementation. It is known in the class where it appears, but it is implemented only in proper descendants. A feature that is not deferred is said to be effective. A class that has at least a deferred feature - inherited or not - is called a deferred class and can not be instantiated.

3.2 Design by Contract

3.2.1 Overview

Design by Contract \textsuperscript{TM} is a methodology for designing and building reliable software. The basic idea is to view “the relationship between a class and its clients as a formal agreement, expressing each party’s rights and obligations” [11]. This separation of responsibilities is crucial in order to build trustworthy large software systems. The analogy with real world is immediate: “a client” and “a supplier” agree on “a contract” in which the obligations and benefits of each are clearly specified. Only after a client fulfils his obligations, he can ask the supplier to perform a certain action for him and expect to get the correct result. The supplier is entitled not to perform anything if the client hasn’t fulfilled his obligations before.

It has been mentioned before that software correctness is a relative notion. A software element can not be catalogued as being correct or incorrect unless its desired functionality is also considered. “The question of correctness doesn’t apply to software elements; it applies to pairs made of a software element and a specification” [11]. A software element or system is said to be correct if it is consistent with its specification. Hence, writing a correct specification is a very important step in ensuring software correctness. In Eiffel, specifications are partly expressed through assertions which are contained inside the software itself. This means that specifications are written before or at the same time with the software that fulfills them. Some of the consequences, as expressed in [11], are:

- Designing the software to be correct;
- A better understanding of the problem and hence a better chance to find an optimal solution;
• Incorporating the documentation into the product;
• Providing help for the testing and debugging processes.

The notion of contract is derived from the “Hoare triple”. If A is an operation, the “Hoare triple”

\{P\} A \{Q\}

means that “any execution of A, starting in a state where P holds, will terminate in a state where Q holds” [11]. This property may or may not hold for any particular operation A. Assertions are just ways to express into software what P and Q express mathematically.

3.2.2 Eiffel Contracts

The assertions used in Eiffel programs can be routine pre- and postconditions, class invariants, loop variants and invariants and check instructions.

**Assertions** are boolean expressions involving various software entities and expressing various properties that these entities are expected to satisfy at certain moments during the software execution. An example of an assertion is

\texttt{Not\_void: x/=Void}

meaning that whenever the execution reaches this point, x should not be Void but attached to an object. Assertions can also be composed of several such simple expressions connected by boolean operators. We say that an assertion A is weaker than an assertion B if and only if B implies A and they are not equal.

The labels associated with the assertions - like Not\_void in the previous example - play an important role during debugging because, in case an assertion is violated at runtime, they help the programmer see which assertion has been violated. The violation of an assertion indicates an error in the source code. Also, the labels increase the readability and understandability of the code, improving the documentation.

**Routine Pre- and Postconditions** are the most often used types of assertions. A routine is a piece of code that performs a task. This task is captured by the precondition and the postcondition associated with the routine. The precondition states the properties that must hold whenever the routine is called; it is the equivalent of P in “Hoare triple”. The postcondition of a routine states the properties that are guaranteed to hold when the routine returns; this is the equivalent of Q in “Hoare triple”.
Preconditions and postconditions must be written carefully as they divide the responsibilities between the clients of a routine and the routine itself. If a routine \( r \) has a precondition - introduced in Eiffel by the `require` keyword -, this means that every client of \( r \) must satisfy that precondition before calling \( r \). If the routine is called and its precondition clause is not satisfied, the routine is not obliged to fulfil its postcondition. At the same time, \( r \) will assume that the precondition is fulfilled when its body will be executed, so its code will not contain too many branches. This is a consequence of the “Non-Redundancy Principle”, which states that “under no circumstances shall the body of a routine ever test for the routine’s precondition” \[11\]. The stronger a routine’s precondition, the easier the routine’s job. A routine postcondition - introduced in Eiffel by the `ensure` keyword - specifies what the routine guarantees will be true after its execution. The stronger a routine’s postcondition, the harder the routine’s job. For expressing a postcondition, the `old` notation can be used. The notation `old e` - where `e` is an expression - denotes the value `e` had on routine entry, which might have been modified in the routine’s body.

The “Precondition Availability” rule \[11\] states that “every feature appearing in the precondition of a routine must be available to every client to which the routine is available”. However, the postcondition’s clauses can refer to secret features, even if this implies that the routine’s full effect is not visible to its clients.

Preconditions must not be used as an input checking mechanism and will not be responsible for correcting the inputs. Also, “assertions are not control structures” \[11\] which means they shouldn’t be seen as techniques for handling special cases, for which we have dedicated conditional instructions.

In \[11\], two rules are defined that are of great interest in the testing process performed by AutoTest. The “Assertion Violation Rule (1)” says:

“A run-time assertion violations is the manifestation of a bug in the software.”

We will define more precisely the picturesque and widely used term “bug” in the next section. Furthermore, the “Assertion Violation Rule (2)” says:

“A precondition violation is a manifestation of a bug in the client.”

“A postcondition violation is a manifestation of a bug in the supplier.”

A precondition violation means that the routine’s caller, although obligated by the contract to satisfy a certain requirement, did not. This has nothing to do with the routine itself and can be expressed as a situation where “the customer is wrong”. A postcondition violation means that the routine, called under correct conditions, was not able to fulfil its part of the contract. The total responsibility for this situation belongs...
to the routine itself, and “the client is innocent”.

**Class invariants** are assertions that represent global properties of all instances of a class, while precondition and postcondition are used to describe the properties of individual routines. Using class invariants, a class can be equipped with deeper semantics and integrity constraints.

A class invariant is a set of assertions - appearing in the invariant clause of a class, after all features have been defined - that must be satisfied by every instance of the class at all times when instances are in an observable state:

- Right after instance creation
- Before and after every call of a routine on that instance

The invariant can be temporarily broken during the execution of a routine, but, in this case, that routine has the duty of restoring the invariant before returning.

The “Invariant rule”[11] defines the conditions an assertion must meet in order to be a correct invariant of a class:

> “An assertion I is a correct class invariant for class C if and only if it meets the following two conditions:

- Every creation procedure of C, when applied to arguments satisfying its precondition in a state where attributes have their default values, yields a state satisfying I

- Every exported routine of the class, when applied to arguments and a state satisfying both I and the routine’s precondition, yields a state satisfying I”

Invariants of a class are implicitly “anded” - in the sense of logical “and” operation - to both the precondition and postcondition of every exported routine of that class. So, the correctness requirement on a routine must be expressed as: whenever the routine is called in a state where its precondition and the invariant of the enclosing class are satisfied, the routine’s body is executed and the instance is left in a state in which the routine’s postcondition and the class invariant are both satisfied.

A class not only inherits the functionality but also inherits the contracts of its ancestors. Hence, its instances must satisfy the invariants of all the ancestors in addition to satisfying its own invariant. The precondition of a redeclared routine - by redefinition or effecting - must be equal or weaker, and its postcondition must be equal or stronger, compared to the original ones. This is due to polymorphism and dynamic binding.
Preconditions, postconditions and class invariants are the most used types of assertions. Besides them, Eiffel defines two other types of assertions: the check instruction and some constructs for ensuring loop correctness. The check instruction serves to express that a certain condition must hold at a certain point of the execution. It is used for documentation purposes and to express non-trivial assumptions that the programmer makes at various places in the source code. It is similar to the “assert” instruction used in other programming languages.

Loop variants and invariants are constructs used to help avoid some common problems that occur when using loops:

- Infinite looping
- Difficulties in handling extreme cases (such as empty structures)
- Performing one more or less iteration than required

A loop invariant is an assertion which must hold at the beginning and at the end of each loop iteration. The language used for assertions is restricted to boolean expressions, enriched with the old mechanism. These expressions can not be used to express some more complicated conditions that may be required at certain time. Here is an example of a loop that counts the maximum element of an array and is equipped with a variant and an informal invariant:

```plaintext
from
    i := t.lower; Result := t@lower
invariant
    --Result is the maximum of the elements of t at indices
    --t.lower to i
variant
    t.upper - i
until
    i = t.upper
loop
    i := i + 1
    Result := Result.max(t @ i)
end
```

When the power of assertions is not big enough to express a condition it is recommended to use a comment for specifying the desired condition in order to fulfill at least the purpose of documentation. This situation is often met in the case of using loop
invariants. A loop variant is an integer expression that must always be non-negative and which is decreased with every iteration of the loop.

Eiffel offers the user the possibility of choosing if assertions are evaluated at run-time for each of the used classes. By a compilation option, the user can switch between various types of assertion checking at run-time: “no”, “require”, “ensure”, “invariant”, “loop” and “check”. It is recommended that during debugging or testing a system, the assertion monitoring should be enabled at the highest level for classes - although not necessarily for highly trusted libraries, for efficiency purposes. If the system is fully trusted and it operates in a real time environment where every microsecond counts, the assertion monitoring may be switched off.

For a better understanding of Eiffel contracts, here is an example, taken from [11], of a class representing an array of various types of elements:
indexing description: "Sequences of values, all of the same type or of a conforming one, accessible through integer indices in a contiguous interval"

class ARRAY[G]

creation make

feature{NONE} -- Initialization

make (minindex, maxindex: INTEGER) is
  -- Allocate array with bounds minindex and maxindex
  -- (empty if minindex > maxindex)
  require
    meaningful_bounds: maxindex >= minindex - 1
  do
    ...
  ensure
    exact_bounds_if_non_empty: (maxindex >= minindex) implies
      ((lower=minindex) and (upper=maxindex))
    conventions_if_empty: (maxindex < minindex) implies ((lower=1) and
      (upper=0))
  end
  ...

feature -- Access

  lower, upper, count: INTEGER
  -- Minimum and maximum legal indices; array size
  ...

feature -- Element change

  put (v:G; i:INTEGER) is
    -- Assign v to the entry of index i
    require
      index_not_too_small: lower <= i
      index_not_too_large: i <= upper
    do
      ...
    end
In this example, everything preceded by “- -” is a comment. The indexing clause describes the purpose of the current class. It is defined for documentation purposes and for providing support for the development environment.

The presented class is generic. Arrays represented by this class can hold various types of elements, but all elements of an array-instance are of the same type, denoted by G. All features grouped under the “Initialization” feature clause - only make in our example - are usable just from inside of the class, because they are not exported to any other class beside NONE. But make is also the creation procedure of the class. When used as a creation procedure, it can be used by all the classes of the system. Its export status - as defined in the feature clause to which it belongs - is applicable only when it is used as a normal feature. The other features are visible from all classes of the system and their grouping under different feature clauses is just for increasing the readability of the code. Features grouped under the “Access” feature clause are attributes, while put is a routine. There are no functions in this example.

The precondition and postcondition of the given routines are quite straightforward. They state what the routines expect from their users before being called and what they guarantee to happen if they are called in a correct way. For example, any client of put must provide a correct index as the second actual argument of the routine. Otherwise he would violate the precondition.

The assertions that form the invariant of the class must hold in all stable states of the system and they are not related to particular routines of the class. The other types of contracts are not shown in this example. All the presented contracts are preceded by labels that clearly suggest their role.

The following table presents the basic keywords related to operating with assertions in Eiffel. “Assertion” and “Exit_condition” are boolean expressions. The “+” sign means that we have to deal with one or more entities of the type given by the preceding word.


CHAPTER 3. THEORETICAL BACKGROUND

<table>
<thead>
<tr>
<th>Contract</th>
<th>Eiffel construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precondition</td>
<td>require assertion+</td>
</tr>
<tr>
<td>Postcondition</td>
<td>ensure assertion+</td>
</tr>
<tr>
<td>Weakened precondition for a redefined</td>
<td>require else assertion+</td>
</tr>
<tr>
<td>routine (using inheritance)</td>
<td></td>
</tr>
<tr>
<td>Strenghtened postcondition for a redefined</td>
<td>ensure then assertion+</td>
</tr>
<tr>
<td>routine (using inheritance)</td>
<td></td>
</tr>
<tr>
<td>Class invariant</td>
<td>invariant assertion+</td>
</tr>
<tr>
<td>Check instruction</td>
<td>check assertion+ end</td>
</tr>
<tr>
<td>Loop contracts</td>
<td>from initialization_operation+</td>
</tr>
<tr>
<td></td>
<td>invariant assertion+</td>
</tr>
<tr>
<td></td>
<td>variant decreasing integer</td>
</tr>
<tr>
<td></td>
<td>until exit_condition</td>
</tr>
<tr>
<td></td>
<td>loop body</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

Tabel 3.1: Eiffel constructs for supporting Design by Contract™

3.2.3 Exception Handling

When assertion monitoring is switched on at a certain level, and one of the active assertions evaluates to “false” (which means that “the contract is broken”[11]) the execution will trigger an exception. If no measure is taken for catching the exception, the execution of the system will stop.

A routine call is a success if it finishes its execution in a state where its contract is satisfied, and it is a failure otherwise. In [11], an exception is defined as “a runtime event that may cause a routine call to fail”. An exception may occur if one of the contracts is violated or if an illegal operation - calling a feature on a Void target, making a division by 0 etc. - is executed. The source of the exception has a great importance in the testing process because we can assess what happened by examining the information provided by the exception. It is worth noting that: “A failure of a routine causes an exception in its caller ... A routine call will fail if and only if an exception occurs during its execution and the routine doesn’t recover from the exception”[11].
Eiffel has a disciplined way of dealing with exceptions. If an exception occurs during the execution of a routine, the routine’s response can be either of the following:

- Retrying to execute the routine from the start after changing the conditions that led to the exception being thrown

- Failing and reporting the failure to the caller

The way to deal with an exception occurring in a certain routine is specified in the text of the routine itself. The instructions to be executed by the routine in case of an exception are introduced by a clause rescue that appear right at the end of the routine code. One of the instructions that might appear in the rescue clause - and only here - is retry, which tells the routine to reexecute its body. “Execution of a rescue clause to its end, not leading to a retry causes the current routine call to fail”, says the “Failure principle”[11].

If a routine fails, it will interrupt the execution of its caller with a “Routine failed” exception, which says nothing about the original source of the exception. The caller has the same two alternatives: either it has a rescue clause that contains a retry instruction, or it fails too and passes the exception one level up in the call chain. If none of the routines from the call chain are able to recover from the exception, the whole execution will fail, and the environment provides the user an exception history table that shows exactly what happened. For obtaining more information about the last exception, there is a library class EXCEPTIONS which provides a number of queries like code of the original exception, code of the current exception, type of the exception, if it was triggered by the user and so on.

### 3.3 Software testing

Testing is the process of providing software various inputs, exercising it on those inputs and comparing the obtained results with the expected ones with the declared intention of finding errors in the software.

#### 3.3.1 Terminology

This section presents definitions for some terms used in this report. Some of these definitions are adapted from [11] and [1] while some are taken from [15].

“An error is a wrong decision made during the development of a software system”[11]. Errors lead to defects which are “properties of software systems that may cause the system to depart from its intended behavior”. If a software system departs from its
intended behavior during one of its executions, we say that a fault occurs. In this case, the execution of the software system fails. The relation between faults and failures is of “many-to-many” type: a fault can cause multiple failures and a failure can be caused by multiple faults. In informal discussions, the term “bug” is often used in the sense of fault and error, but it usually has the meaning of defect.

Testing is a dynamic technique - involving code execution, real or simulated - for software verification. To test a program means to try to identify bugs in it. To debug a program means to try to remove bugs from software.

Testing can be done at different levels during the software development process. According to this criterion, we can talk about:

- Unit testing: the subject of testing is a single component of the software, or a module - in our case a class;
- Integration testing: the subject of testing are the interactions between the modules of a system;
- System testing: the subject of testing is the software system as a whole in order to see if it meets its requirements;
- Acceptance testing: performed for determining whether the customer should accept the program or not;
- Regression testing: re-running tests after making changes to the software.

The units that can be tested in an object oriented context can be: an instruction, a feature, a class, a library, a program etc. So, we can talk about feature under test, class under test, library under test, system under test (SUT) - etc. The scope of a test is the collection of software elements that the test targets.

In order to perform a test, one or more test cases should be defined. A test case represents the pretest state of the tested unit and its environment, inputs and conditions on which the functionality of the system is checked and the expected result. A sequence of test cases forms a test suite. The program or tool used to execute the test suite on the tested system is called test driver. For assessing whether a certain test case has been successfully executed or not, a test oracle must be used. It defines the expected behavior of the program on running the considered test case.

3.3.2 Testing in software engineering

Testing is a problem in software systems engineering. “It is the design and implementation of a special kind of software system: one that exercises another software system
with the intent of finding bugs” [1]. Tests are designed only after analyzing the system under test and deciding how it can be buggy. After having designed the tests, one can design the test automation system that will apply and evaluate the tests. The test automation system “must be designed to work with the physical interfaces, structure and the runtime environment of the system under test” [1].

Typical software systems have a very high complexity. For increasing the efficiency of testing, abstraction must be used. Test design must integrate both general - which “offer a systematic and repeatable means to generate test suites” [1] - and application specific - which “represent the required behavior of the system under test” - test models. As an analogy, general test models can be viewed as general functions, and application specific test models can be viewed as their inputs. These relations are sketched in the figure 3.1.

According to [1], test design involves several steps:

- Identify, model and analyze the responsibilities of the system under test.
- Design test cases based on this external perspective.
- Add test cases based on code analysis, suspicions and heuristics.
- Develop expected results for each test case or choose an approach to evaluate the pass/no pass status of each test case.

After the design is complete, tests are applied to the system under test. For this, a test automation system - sometimes encapsulating manual tests - must be used. A test automation system must start the system under test, set up its environment properly,
bring it to the required state for executing tests, apply the test inputs and evaluate the resulting output and state.

3.3.3 The limits of testing

Any non-trivial software system has bugs. These bugs are uncovered by various combinations of execution sequences that can not be known in advance. We can not conclude that a system is bugless if it passed one or more tests. A famous quotation related to this problem is attributed to Dijkstra: “Program testing can be used to show the presence of bugs, not their absence!” Even passing many tests for many different test cases, a program will not necessarily perform correctly for the untested test cases.

The most important limits in the process of testing are:

- The input/state space, which is vast for typical applications. The number of input and output combinations is very high and the space can never be tested test exhaustively.

- A program has very large unique execution sequences especially because of branches and dynamic binding.

- Coincidental correctness, which denotes a situation where a buggy code can act well for some certain combination of inputs. An example taken from [1] is the following: suppose we have wrongly coded $X + X$ instead of $X * X$. The bug will not be uncovered if we use one of the values 0 or 2 as input.

- Fault sensitivity, which is the ability of code to hide faults from a test suite. Sometimes, a buggy line doesn’t produce a failure when executed.

- The testing system itself may not be correct.

3.3.4 The testing process

During the software development process, testing has the most important role in obtaining quality systems. According to [15], the testing process consists mainly of the following steps:

- Choosing the elements to be tested (the granularity can vary as previously shown);

- Building test cases and test oracles;

- Executing test cases;
• Comparing actual results with the expected ones (as given by the corresponding test oracle);

• Measuring execution characteristics (time, memory...).

3.3.5 Testing strategies

The second activity mentioned above is in particular very time consuming and effort intensive when performed by a human tester. Since exhaustive testing is in most cases impossible relevant test cases may be omitted.

Strategies for testing software can be grouped in two categories: manual and automated. With a manual strategy, testers are responsible for creating the test suites that they think will best exercise the program, and for providing test oracles for these test cases. When a fully automated strategy is used, the responsibility of creating test cases and test oracles is delegated to a software tool. For doing this, the software tool can rely only on the specification of the tested program - this is called black box testing - or also on its actual code - white box testing.

Ideally, all the possible executions of a program must be tested. Otherwise, many bugs can remain uncovered if one or more combinations of inputs remain untested. But exhaustive testing is an unattainable goal because most programs have a very large or even infinite set of possible inputs. Automated generation of test cases and oracles can be more efficient in solving this problem because it can rely on the computation power of a machine, but it still won’t do exhaustive testing. Anyhow, the generation process must be guided somehow in a way that the resulted test cases would be relevant.

Manual testing frameworks automate the activities of executing the test and gathering the testing results, but test cases and test oracles must be created by hand. To add a new test case, a user must generally create a new test class that inherits from an abstract predefined one. He can define as many testing routines as he wants in this new test class.

Tools generally use reflection to trigger the set of classes from the entire system that are involved in the manual testing process. A class is member of this set if it inherits from the predefined abstract test class. The member routines of these classes are then responsible, through their execution, for exercising the system by creating objects and invoking routines on them. They also verify whether the obtained result is or not identical with the intended one.

Some manual unit testing frameworks currently used are:

• jUnit, for Java; [16]
• sUnit for Smalltalk; [17]  
• pyUnit, for Python; [18]  
• Gobo Eiffel Test, for Eiffel; [19]

Such frameworks are typically small but they provide significant practical benefits [9].

Automated testing automates all the activities involved in the testing process. “A fully automated testing tool is able to test software as-is, without any user intervention” [9]. Even if the use of automated testing means less effort on the developer’s side, it might not perform so well in finding relevant test cases as a person who knows the software. On the other side, if a human tester doesn’t understand the software behavior properly, he may not think of certain borderline test cases, for instance testing a feature on a Void target, and in this way bugs may remain uncovered. Not trying to understand the intended semantics of the system can be an advantage for the automated strategies.

Automated and manual strategies are generally implemented in different tools. Actually, they are complementary as each addresses the weaknesses of the other one. “Manual tests are good in capturing deep or special cases, which automated tests might not guess; but they can not yield extensive coverage because of the sheer number of test cases this requires. Automated tests are good at breadth, and worse at depth” [9].

3.3.6 Testing based on Design by Contract \textsuperscript{TM}

Contracts are integrated into the Eiffel source code. They are executable specification and describe the intended purpose of the code they equip. This makes them perfect candidates for oracles to be used in the software testing process. The existence of executable contracts makes the intervention of human in the process of testing useless.

This was the starting point for the idea of “contract-based testing”: to use the contracts as automatic oracles. The information contracts provide is used to assess whether the software is correct or not. The information provided by thrown exceptions - in case of a broken contract - can be used to generate statistics about the bugs found in software. If we add the possibility of automatic generation of test cases, we have a complete general description of any tool that performs automatic contract-based testing.

“Excepting the case when a generated test case directly violates the precondition of the routine under test (and hence this is an invalid test case), any contract violation signals a mismatch between the implementation and the specification.” [9] Hence, whenever a contract violation is encountered - except for the case just mentioned -, AutoTest has uncovered a bug. The idea is illustrated in Figure 3.2.
3.3.7 Related work

The idea of contract-based testing is not new. Tools like Eclat [14], Jarteg [13], Directed Automated Random Testing - DART [7], Agitator [2] or Korat [3] already use contracts in the testing process. The Eclat tool generates unit tests for Java classes. Eclat’s input is a set of classes to test and examples of program successful execution. It infers the operational model of the tested software from the examples of successful executions. It implements a technique that selects from a large set of test inputs a small subset likely to reveal faults in the tested software. Eclat’s output is a set of JUnit test cases, each containing a potentially fault-revealing input and a set of assertions at least one of which fails.

The Jarteg tool permits a random generation of unit tests for Java classes that have JML specifications. The JML specifications are used both for supressing irrelevant test cases and as oracles. Jarteg randomly generates test cases, which consist of a sequence of constructor and method calls for the classes under test. Weights can be attached to classes and methods. Also, the number of instances created from each class under test can be controlled.

The DART tool combines three techniques: automated extraction of the interface of a program using static source code parsing, automatic generation of a test driver for this interface using a random testing strategy and a dynamic analysis of how program behaves on the generated inputs. The main strength of DART is that testing can be performed completely automatically on any program that compiles. There is no need to write any test driver or harness code.
The Agitator testing tool implements the idea of agitating the software: a unit-testing technique for automatically exercising individual units of source code in isolation from the rest of the system. After writing the code, the developer invokes the agitation process and, as a result, he gets a set of observations. These observations can be used for writing invariants for classes.

“The testing tool called Korat uses preconditions to generate valid and non-isomorphic inputs for each method under test and postconditions to evaluate the correctness of the methods.” Korat is used for Java programs enriched with JML assertions. The main disadvantage of this tool is that it is not fully automatic. It generates test cases based on parsing preconditions and a “finitization function” provided by the user for being applied on the inputs. Even if the skeleton for this function is generated by the tool, the user has to edit it in order to get the desired behavior.

Compared with other tools that implement this idea, AutoTest makes improvements on at least two important factors:

- It doesn’t require any user intervention, being completely automatic
- It has been tested on industrial size applications and libraries

AutoTest is a tool that permits completely automated unit testing of classes. It is able to compile manual tests, generate new ones and run them on the push of a button. It tests both automatically and manually, uncovering bugs that one strategy, used alone, may miss. It tries to reconcile both types of testing strategies in order to benefit from the advantages of each. AutoTest has proved its efficiency by detecting numerous unknown bugs in production libraries and software.
Chapter 4

Random based testing strategies

Test case selection is one of the main issues of software testing. Test cases can be selected according to different selection strategies, each having its advantages and disadvantages.

4.1 Pure random testing

The collocation “random testing” is used for denoting the use of a random strategy during the process of selecting test cases. In computer science, the attribute “random” has a number of bad connotations like “not well organized”, “hastily done” or “superficial”. Hence, “random testing” can be seen as being the opposite of “systematic” testing. From the mathematical, technical point of view, the word “random” refers to an intended lack of organization and systematization in the choice of testing inputs. By randomly creating inputs, we can be sure that there is no correlation between them. Two reasons for being desirable to be “unsystematic” on purpose in selecting test inputs could be

- The possibility to easy-define a vast number of tests, by using the efficient existing methods for selecting points randomly.
- The independence among test points allows statistical prediction of relevance in the observed results.

The first argument can be argued, because sometimes the expected results for a random input are hard to be generated. However, if a systematic strategy is used, the risk of not obtaining significant results is higher than when a random strategy is used.

A pure random strategy has the best efficiency when used for testing a software system as a whole. When used for testing a component of a library, random testing must select only test cases similar to the ones that will appear when the component is
part of a real system; if it selects irrelevant test cases, random testing looses its capacity of predicting the relevance for the obtained results.

Even when used for testing a system, the simplicity of random testing is influenced by the several ways in which a program can be used [8]:

- For a **batch program** - which treats one input at a time and the obtained result is a function of that input only - the sequence of inputs is irrelevant and hence a random strategy can be used: the tests are not related to each other at all.

- For an **interactive program** - which operates continuously by repeating a cycle of reading the inputs, processing them and writing the results – an input can have influence on the following ones. The program is driven in different “modes of execution” by different sequences of inputs and can behave differently in these different “modes”. When used for testing interactive programs, a random testing strategy must take into account all possible sequences of inputs.

- **Operating systems and realtime control programs** receive sequences of inputs in a chaotic way. In this case, a random strategy must take into account not only the input values and their sequence, but also the spacing and overlapping in time for the input sequences.

Random testing can successfully compete with systematic testing strategies. When systematic testing is used, some part of the program, which is suspected of being a possible source of failures, is always included in testing. However, systematic testing strategies can be very subjective in selecting test cases, because they rely on a human choice. But this choice is not always a good one. Purely random testing strategy replaces the human choice with chance selection, taking the risk of not selecting some failure triggering test cases at all. The chance selection is not necessarily inferior to a human choice if, for example, the person who makes the choice is not completely aware of the program functionality. Besides this, random testing is simple in concept, easy to be implemented and offers ways to estimate its reliability [8].

### 4.2 Adaptive random testing

Several methods that maintain the benefits of the random testing strategies and also increase their efficiency have been proposed lately. They try to guide the process of test case selection, hence loosing the property of being purely random.

In practice, testing with random inputs has proved to be efficient if the selected inputs are uniform distributed over the input domain. Adaptive random testing (ART)
strategies are based on the idea of selecting inputs in such a way that they are evenly spaced across the range of all possible input values. The intrinsic characteristic of ART is to have randomly generated test cases and select test cases one by one, according to a certain criterion.

Depending on the used criterion for selecting the next test case, there are various versions of ART. Distance based ART (D-ART) is a variant of ART that requires the existing of a way to measure distance between test cases. It basically consists of two activities:

- Generating random test cases candidates from the whole input domain
- Selecting the test cases one by one based on the criterion of maximizing the minimum distances between the test case candidates and all previously used test cases.

For implementation D-ART requires the existence of two disjoint sets: a candidate set and an executed set. Initially, the first set contains the random generated candidates test cases and the second one is empty. The first test case that is executed is selected randomly from the candidate set. It is removed from the candidate set, executed and put into the executed set; in the following steps, the furthest away candidate from the executed ones is selected from the candidate set, executed and moved to the executed set. So far, D-ART has been used only for inputs of primitive types equipped with a total order relation and for which it was easy to compute an Euclidean measure. The distance between two test cases \( a \) and \( b \) whose inputs are \( a_i \) and \( b_i \) respectively, for \( i \in \{1, \ldots, n\} \), is \( \text{dist}(a, b) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2} \).

Another variant of ART is called Restricted Random Testing (RRT). In RRT, some zones of exclusion are defined in the input space. If a candidate is generated in an exclusion zone, it is discarded and another random generation is attempted. After that, candidate tests are selected in a similar way as in D-ART.

For reducing the distance computation, Mirror ART (M-ART) can be used. In this version of ART, the whole input subdomain is first partitioned into disjoint subdomains, and D-ART is applied only in one subdomain. The selected test cases are then mapped into the other subdomains by using simple mirror functions. The number of distance calculations decreases for a higher number of subdomains. However, the number of the mirror subdomains must be kept small for maintaining the randomness characteristic.

Two ART methods which do not require distance computations at all are ART by Random Partition and ART by Bisection. The difference between these two methods and the other presented ones is that they locate the area of the input space where the next test case will be generated, instead of computing which is the best suitable test
case from the set of all generated ones. These two methods divide the input space into subdomains by using a partition scheme, and choose at each step a subdomain in which a test case will be generated. The chosen subdomain is called the test case generation region. The difference between these two algorithms is the partitioning scheme used:

- ART by Random Partition divides the input domain according to the last selected test case. The subdomain with the largest area is chosen as the new test case generation region. The way in which this algorithm works is sketched in 4.1 [4].

- ART by Bisection divides the input domain into subdomains of equal size and chooses as the new test case generation region the subdomain that contains the smallest number of executed test cases.

![Figure 4.1: ART by Random Partition](image)

### 4.3 Object distance

The idea of ART is an appealing one for testing today’s object-oriented programs. The problem is that for these programs the inputs are not elementary values but composite objects, with many fields, obtained by instantiating classes, for which we do not have an adequate measure of distance. The set of existing run-time objects is not equipped with an order relation, so we can not assess what means for objects to be “equally spaced” in a range. For addressing this issue, the notion of object distance was defined. This distance was used during the current project for developing new selection strategies for AutoTest, based on ART.

The distance between two objects is a function “$\leftrightarrow$” that takes as arguments two objects and returns a real value. More than this, it is a distance function in the mathematical sense of term, having the following properties:

- $p \leftrightarrow q > 0 \ (p \neq q)$ and $p \leftrightarrow p = 0$ (Positivity)
- $p \leftrightarrow q = q \leftrightarrow p$ (Symmetry)
• $\forall r : p \leftrightarrow q \leq (p \leftrightarrow r) + (r \leftrightarrow q)$ (The Triangle Inequality)

For computing object distance, some other notions must be first introduced.

### 4.3.1 Elementary distance

A composite object contains fields - attributes - which have at run-time various kind of elementary values. The distance between two elementary values $p$ and $q$ is defined, according to their type, as [5]:

- For numbers: $C(|p - q|)$, where $|p - q|$ denotes the absolute value of the distance between number values and $C$ is a monotonically non-decreasing function, which has a fixed point in 0. AutoTest uses the function $C(p - q) = (1 - \frac{1}{|p - q|}) \times Max_{elementary\_value}$, with a value of 1 given to $Max_{elementary\_value}$;

- For booleans: 0, if $p = q$, and a default value $B$, otherwise. AutoTest uses a default value of $B$ equal to $Max_{elementary\_value}$;

- For strings: The Levenshtein distance is used. The Levensthein distance - also known as the edit distance - between two strings is the minimum number of elementary operations (insertion, deletion, substitution of a character) needed for transforming a string into the other;

- For references: 0 if they are identical, a default value $V$ if only one reference is Void, or another default value $R$ if they are different and none of them is Void. AutoTest uses $V = Max_{distance\_value}$, which is set to 3, and $R = 0$.

### 4.3.2 Type distance

A polymorphic variable, statically declared as being of a certain type $T$ can be attached at run-time to objects of different dynamic types, which are descendants of $T$. We need to compute distances between two objects that have different dynamic types because, in the context of dynamic binding, objects of different types can be candidates for the same position in a routine call.

The type distance principle [5] states that the distance between two types must take into consider their path length to any closest common ancestor and the number of their non-shared features (attributes and methods). The type distance must be an increasing function of both these elements.

A closest common ancestor of classes $B$ and $C$ is a class $A$ that is an ancestor of both $B$ and $C$ and none of its descendants have this property. In languages where the inheritance hierarchy has an unique root class (ANY in Eiffel, Object in Java etc.), there
will always exist a common ancestor for any two classes. In other languages, the type
distance between two classes which don’t have a common ancestor can be considered
infinite. In languages that permit multiple inheritance (like Eiffel), we can have more
than one closest common ancestors for two given classes. In this case, type distance
must be a function of distances to all closest common ancestors.

The path length between a class B and an ancestor A is the minimum number of
edges on a path between B and A in the inheritance hierarchy. An edge is a link between
two classes.

Non shared features are those features that are not inherited from a common ances-
tor. More unshared features two types have, bigger the distance between them is.

4.3.3 Composite object distance

Having defined the concepts of elementary distance and type distance, we can step
further for defining what distance between composite objects means. The distance
between two distinct composite objects p and q should depend only on the following
elements [5]:

• A measure of the difference between the dynamic types of objects. The distance
  between the dynamic types of p and q is computed according to the principle
  stated in section 4.3.2.

• A measure of difference between the elementary values of the fields of p and q.
  Section 4.3.1 has shown how the elementary distance is computed. The elemen-
tary distance is computed between the matching fields of p and q - those that
correspond to a common attribute of their types. The difference caused by the
fields of an object that don’t have a pair among the fields of the other object is
already captured by the distance between the dynamic types of the objects.

• For elementary values of reference type there is no relevance in taking into consider
the values of references as memory addresses. However, the objects attached to
references should be considered when computing the distance. If references are
different, the distance between the corresponding objects is computed recursively.
In this way, the distance is influenced by the runtime object graph. AutoTest
goes two levels in depth for computing the reference distance recursively.

The distance between p and q should be an increasing function of each of the above
elements taken separately. Every attribute \( a \) of a class can have a non-negative weight
associated, \( weight_a \). The type, field and recursive distances must be increasing functions
of \( weight_a \times d \), for each \( d \) computed between \( a \) and a corresponding attribute of another
object, when $weight_a$ is positive, and they must not depend on $d$ if $w = 0$. AutoTest uses a weight of 1 associated with every attribute.

Having all these metrics defined, we can define the needed formulas for computing distance between two objects $p$ and $q$ [5]:

$$field\_distance(p, q) = \sum_a weight_a \times elementary\_distance(p.a, q.a)$$

where $a$ denotes all the matching attributes of $p$ and $q$.

$$recursive\_distance(p, q) = \sum_r weight_r \times (p.a \leftrightarrow q.a)$$

where $r$ denotes all the matching reference attributes of $p$ and $q$. Only the references that are not equal and neither of them is Void need to be considered.

$$type\_distance(p, q) = \lambda \times path\_length(p.\text{type}, q.\text{type}) + \nu \times \sum_{a \in non\_shared(p.\text{type}, q.\text{type})} weight_a$$

where $\lambda$ and $\nu$ are two non-negative constants. AutoTest uses the value 1 for these two constants.

The distance between objects $p$ and $q$ is a linear combination of the 3 above defined expressions:

$$p \leftrightarrow q = \tau \times type\_distance(p, q) + \phi \times field\_distance(p, q) + \alpha \times recursive\_distance(p, q)$$

where $\tau$ and $\phi$ are non-negative constants and AutoTest uses a 1 value for both of them. $\alpha \in (0, 1)$ is an attenuation factor.

The last formula is a recursive one because of its last right term. Hence, beside ensuring that $\leftrightarrow$ function is an increasing one on $field\_distance(p, q)$, $type\_distance(p, q)$ and $recursive\_distance(p, q)$ we must also ensure that it converges. This is the cause why the coefficient of the last term, $\alpha$, is an attenuation factor. This happens only in theory: AutoTest uses an $\alpha$ with a value of one, because we can not have an infinite recurrence here. As was said before, AutoTest goes only two levels in depth for computing the distance recursively, considering the further objects from the graph run-time object structure as being irrelevant for computing the distance.
Chapter 5

AutoTest - a push-button testing tool

For implementation purposes, we used AutoTest. This command-line tool was developed by Ilinca Ciupa and Andreas Leitner, PhD students at Chair of Software Engineering from ETH Zurich, Switzerland. AutoTest is the successor of TestStudio, and its source is available for free download [20].

This chapter presents some characteristics that AutoTest had before the start of my project. The added facilities will be described in chapter.

5.1 Overview

The term “push-button testing” really suits AutoTest: all the user must do is to provide the tool with a file describing the architecture of the system to be tested and to specify the set of classes he wants to be tested. Then, AutoTest will test those classes in an automatic way, generating and running tests without requiring any external human intervention. Optionally, the user can specify an amount of time allocated for testing, otherwise an implicit timeout will be used.

One of the novelties AutoTest brings is the possibility of integrating manual tests with automatically generated ones. The user can specify any test case he considers relevant in a very simple way: he has to create a test class that inherits from a specific library class, and he has to supply all the desired manual test cases as routines of this test class. AutoTest can detect the manual testing classes and it runs these manual tests first. The remaining time is dedicated to creating and executing as many automated tests as the time constraints permit.

Integrating manual and automated tests is facilitated by the use of a single mechanism as test oracle: Eiffel contracts. AutoTest interprets contract violations as bugs in
software. This means that “the quality of the test results is highly dependent on the quality (correctness and completeness) of the contracts” [20].

The possibility to require so little from the user (only the system and a list of classes to be tested) is related to two properties of the object technology as realized in Eiffel [12]:

- Each possible run-time object is an instance of a class. The class defines all the possible features applicable to its instances. AutoTest - relying on the computation power of today’s computers - can then generate a large number of sample objects and invoke features on them.

- Contracts that equip Eiffel classes specify the expected behavior which can be monitored during execution. Thus, contracts can act as oracles.

After the specified testing timeout has elapsed, AutoTest provides the user information about the process of testing. For all the classes and features that were tested, various statistics (how many tests were a success, how many failed, how many generated tests were invalid and so on) are displayed. Various information has been added as part of the current project.

5.2 Functionality

AutoTest permits completely automated unit testing of classes. It needs very little information from the user in order to perform tests. In particular, user has to provide a control file (an ACE/Xace file in our case) that contains all the information AutoTest should know about the tested system, and the names of the classes to be tested. The test scope for AutoTest contains all the classes the user lists. Any other classes will only be exercised indirectly, through the classes under test. Optionally, the user can specify the amount of time allocated to testing; if he doesn’t, the testing process will last for a default time of 30 minutes.

An AutoTest session will, in the given time, test the given classes and their features using various heuristics and strategies for increasing the effectiveness of the testing process. AutoTest retains a list of all the features that have to be tested. Features are given priorities and they are selected for being tested in the decreasing order of their priorities. After every execution, the priority of the executed feature is decreased.

A simplified description of what testing a routine means is provided in [20] and it consists mainly from the following steps:

1. Select a target object and one or more argument objects (if required) for the
currently selected feature, according to the type of the target and of the formal arguments for the feature. Try to call the feature using the selected objects.

2. If the precondition of the routine under test is satisfied by the currently selected arguments, the body of the routine is executed, otherwise we have an invalid test case.

3. If the precondition is not violated, the test case can be run. Any contract violation that occurs in the body of the routine or somewhere during the call chain, indicates the presence of a bug in the tested software. If no such bug is found, the test case has passed.

Objects to be used in the call can be newly created ones or can be selected from the objects previously used in the current session for testing the system. Initially, AutoTest used a completely random strategy for choosing the objects needed for calling the feature. Later, a strategy implementing the D-ART idea has been added.

It used a random strategy for creating new objects, too. Basic values and reference type objects are created differently:

- For primitive (basic) types, a predefined set of values is available and one element of the set is chosen randomly. For instance, for integers, this set contains the values: -1, 0, 1, -2, 2, 3, 4, 5, 6, 7, 8, 9, 10, -10, 100, -100, INTEGER.Min_value, INTEGER.Max_value. Whenever an integer is needed, one of these values is randomly selected. The authors have tried to put in these sets the most representative values for each primitive type.

- For non-primitive types, objects of the corresponding type are built by using a randomly selected creation procedure from all the available ones for that type. Arguments needed for that creation procedure are selected in the same way: either by creating a new object, or by selecting an already existing one.

During the testing process, a pool of objects is maintained. All newly created objects are added to this pool for possible later reuse. At each step, besides creating a new object, there is the alternative to select from the pool an object conforming to the desired type. Objects are never removed from the pool.

Objects from the pool will periodically be diversified - with a preset frequency - by selecting randomly one of their class’ available procedures and invoking the procedure on the object. This is in addition to the diversification already performed by running tests using pool objects. Objects used for running a test or a diversifying routine are returned to the pool in their new state, which may be different from their previous state.
As can be easily observed, much randomness is involved into the testing process. This fact has many benefits: ease of implementation, ease of understanding, speed of execution (which makes possible the execution of an increased number of tests) etc. The chapter 6 presents two new strategies used for selecting inputs, whose purpose will be to choose the objects in a more systematic way by assessing the available choices according to various criteria.

For a better understanding of how testing proceeds, here is a simple example. Let’s suppose we want to test a class named PERSON that has an attribute age of type INTEGER. For setting the age of a person, we need to define a routine because, as we have mentioned in the section 3.1, an attribute of a class can not be modified directly in Eiffel even if it is not hidden. Here is the corresponding part of the class:

```eiffel
class PERSON
  ... 
feature --Access
  ...
  age: INTEGER
  -- Age of the person
  ...
feature --Modifiers
  ...
set_age (an_age:INTEGER) is
  require 
    valid_age : an_age > 0
  do 
    age := an_age
  ensure 
    age_set: age = an_age
  end
  ...
end
```

In order to test the set_age routine, we must first create a target object (an instance of class PERSON, the enclosing class for the tested routine) and an INTEGER argument for the call. If, for example, the object generator gives us a negative value for the INTEGER argument, the precondition is violated and the routine can not be run. This is an example of an invalid test case. If the object generator gives us a positive INTEGER number as argument, the precondition is satisfied and the routine will be run. Supposing that our routine calls other routines in its body, if any contract is violated in these routines we have found a bug. Also, if the routine performs an illegal
operation or violates its postcondition or the invariant of the enclosing class, we have also found a bug.

Until now, we have presented the basic ideas used by AutoTest to automate the test generation and execution. But this framework also supports manual tests. This facility can be used by the user to indicate to the system that he wants a certain test case to be executed. AutoTest detects all the manual test cases (all the classes that inherit from an abstract class named \texttt{AUT\_TEST\_CASE}) and it is able to select only the relevant ones from them. The relevant manual test cases are those targeting one of the classes found in the test scope. The relevant test cases are detected by one of the two fundamental relations of object-oriented programming: the “client relation”. So, from all the existing manual test cases, only the ones that are client of a class in the scope are retained.

AutoTest runs all manual tests at the beginning of the session. All the remaining time after executing the manual tests is dedicated to executing automatic tests. The reason behind this scheduling is that the existence of manual tests is an indication that the user really wants those cases to be tested.

During test execution, no distinction is made between the two kinds of objects. Manual and automatic objects are treated in similar ways: both types contribute to the object pool and both can be chosen for diversification. An advantage emerging from treating manual and automated tests similarly is that their coverage will be computed together. Manual testing can be disabled by a specific command-line argument.

When the testing process is finished, the user is provided with statistics for the testing process. For every feature and every class involved in this process, test results are grouped in the following categories:

- **Pass**: A feature/class is classified in this category when all the test cases run on it have passed. A test case is classified as pass if it is valid and has run successfully.

- **Fail**: A feature/class is classified in this category when one of the test cases run on it has failed. A test case is classified as fail if it is valid but, during its execution, a bug in the system under test has been uncovered (i.e. a contract violation occurred or another exception was thrown).

- **Invalid**: A feature/class is classified in this category when all the tried test cases for it have been invalid. A test case is invalid when any of the required conditions for running it was not fulfilled. Such an example is a violation of a precondition of a routine by the automatically generated arguments for testing the routine.

- **Bad response**: A feature/class is classified in this category when all the valid test cases run on it have provoked an invalid result.
• **Untested**: A feature/class is classified in this category when there has been no test case run on it.

Figure 5.1 presents the way in which AutoTest displays the results for the user.

![AutoTest statistics for system 'unified_testing'](image)

**Figure 5.1**: Results provided by AutoTest

For every feature, the number of each type of test case is displayed. For each test case, AutoTest will provide a *witness* - a test scenario automatically generated by AutoTest - and the result of running the test case. During the current work, we have extended the statistics offered to the user in such a way that he is provided also, for failures, with the time - in seconds - that has elapsed from starting the testing process until the failure has been triggered, the cause of the failure and if the cause is an internal or an external one. Some statistics for the entire system and for every class have been also provided.

Failure witnesses will be displayed in a minimized form. This means that even if AutoTest has detected the failure after a longer sequence of calls, some of these calls are not really needed and the tool will compute a minimal sequence of calls equivalent to the one that led to failure. This helps the user to understand what happened and correct the bug.

### 5.3 Architecture

For executing tests, AutoTest uses a two process architecture. This allows test case execution to continue even if a particular test case execution has failed. Failures of test cases are not uncommon events, especially when the system under test or the time
devoted to testing are big enough. “Producing failures is indeed among the intended outcomes of the testing process, as a way to uncover bugs, but this shouldn’t stop the testing process itself” [12]. For automating the process of testing under these circumstances, at least two communicating processes are needed: one to drive the testing activity and another for actually performing tests. The first one is not allowed to fail. The second one may fail while executing one particular test, but must be restarted by the driver after the results are recorded. The two communicating processes used by AutoTest are called:

- **The driver**, responsible for coordinating the testing activity
- **The interpreter**, responsible for actually executing all the tests

The interpreter is an independent component that is able to execute various simple commands received from outside, returning the result of the execution as a text message. It is delivered with the Eiffel Reflection Library Generator (ERL-G) [21]. The interpreter uses ERL-G for dynamically calling routines on all the alive types of the system under test. ERL-G is responsible for generating all the alive types for the SUT. A type is said to be “alive” if it corresponds to a class under test (one of the classes specified by the user as a command line argument) or if it is part of the transitive closure of the set of types needed by the previous ones [20].

The interpreter is able to execute simple commands as creating an object, calling a feature on an object, assigning a result to a variable, returning the type for a specified variable and so on. All the operations done by the interpreter are written in its log file, that can be found in its directory. For communicating with the driver, another log file is used. This file contains all the messages exchanged between the driver and the interpreter and it can also be found in the directory of the interpreter.

The driver passes to the interpreter various commands that the latter understands, according to a testing strategy. In other words, the driver is the “master process” that knows the testing strategy, and the interpreter is the “slave process”, actually responsible for executing the tests.

The main components of the driver are [9]:

- **The testing strategy**: AutoTest allows arbitrary testing strategies to be plugged in. “The pluggable testing strategy is responsible for determining exactly on what inputs the system under test should be invoked” [9]. The testing strategy receives a description of the SUT - under the form of an abstract syntax tree (AST) - and the set of classes that should be tested. Having this information, it generates the test cases forwarded to the proxy. The test strategy provided by default is, as described previously, a totally random one, integrated with a strategy that
handles manual test cases. During the current project, we have developed two other new testing strategies. A comparison between various alternatives that can be used as testing strategies was provided in the previous chapter.

- **Proxy**: This is the part responsible for managing the communication between the driver and the interpreter. It receives commands from the testing strategy component and forwards them to the interpreter via the proxy log file. The interpreter reads the commands from the file, executes them and writes back its responses. The execution results are then sent by the proxy to the oracle component.

- **Oracle**: In the testing process, the oracle part is known as being hard to automate. As described before, this process is facilitated by the use of contracts, which are executable specification tightly integrated into the Eiffel programs. This component receives the execution results and determine the outcome of the execution. It then saves the results in HTML documents or CSV files.

The figure 5.2 taken from [9], presents the overview architecture of AutoTest.

Figure 5.2: Architecture overview for AutoTest

Besides all the described components - drawn with rounded corners -, the other components - drawn in rectangles - represent the inputs (SUT and test scope) and the outputs (results) of AutoTest.
5.4 Structure

It is easy to understand the internal structure of AutoTest, because classes are grouped in clusters and subclusters respecting the described architecture. AutoTest is very well structured and easy to be extended.

The classes are grouped in two main clusters. The cluster named *auto_test* contains just two classes: the root class of the application and the class responsible for parsing the command line. The other main cluster is called *auto_test_library* and it contains different classes responsible for implementing the strategies used by AutoTest, for communicating with the interpreter or for generating the results, respectively. This second main cluster also contains some other useful classes for the system: a class responsible for creating random numbers, classes responsible for accessing the file system, classes responsible for generating unique names, error handlers for various messages that can appear during execution etc.

5.4.1 The root class of AutoTest

The root class of the system is called *AUTO_TEST* and it inherits from *AUTO_TEST_COMMAND_LINE_PARSER*. The class diagram representing only these two classes, and omitting the rest of the inheritance hierarchy is presented in the figure [5.3].

The creation procedure of the root class is called execute. Its main steps are:

- Processing the input arguments using the process_arguments method inherited from *AUTO_TEST_COMMAND_LINE_PARSER*. The arguments are read in a predefined order, various flags being set for guiding the execution. The main arguments and their order are described in 5.5.2.

- Compiling the system using its description provided in an the .ace file. The structure of such a file is presented in 5.5.3. By compiling the system, an universe containing all classes used by the application is built.

- Using the universe just created, the method generate_interpreter is used for creating the proxy that will communicate with the interpreter. Also, the directories where the results will be stored are created.

- The routine update_remaining_time is used for computing how much time has elapsed since the system was started. The maximum time can be given as an input argument, as described in 5.5.2.

- If there still is some time after generating the universe and the interpreter, the tests are executed by using the method execute_tests. Depending on an input ar-
Figure 5.3: Root class of AutoTest
argument, various methods, implementing various testing strategies, are then called: `execute_random_tests` for using a random strategy, `execute_art_tests` for using the new ART strategy or `execute_man_art_tests` for using the new strategy that combines ART with manual testing. These last two strategies were developed during the current project.

- `build_result_repository` builds the structures containing the results, after the time allocated to testing has elapsed. The results are built by parsing the log file that contains all the messages exchanged between AutoTest and the interpreter during the testing process. The results are stored in a result repository that contains all the information about the testing process. If the minimization is enabled, the `minimize_witnesses` method is used for creating smaller equivalent witnesses for the detected failures. Using the result repository, `generate_html_statistics` builds the HTML files containing the results.

- `generate_csv_statistics` builds a file with main results obtained by testing the system. When the corresponding argument is turned on, the tests are not reexecuted again; only the result repository is built according to the log.

### 5.4.2 Integrating testing strategies in AutoTest

Tests can be executed using various testing strategies. The hierarchy containing the classes used by AutoTest for implementing the testing strategies is presented in the figure 5.4. For creating a new strategy, the corresponding class must be added to the hierarchy inheriting directly or indirectly from `AUT_STRATEGY`.

Manual tests are executed differently from automated ones. The difference comes from the fact that the order in which features are called by a manual strategy and their target and arguments are given by the user. For an automated strategy, they are created or selected according to an algorithm. As can be seen in the figure 5.4, the classes that implement our automatic testing strategies are all descendants of the class implementing the random strategy.

A feature and its type are stored as instances of `AUT_FEATURE_OF_TYPE`. All the features that have to be tested automatically have a static and a dynamic priority associated and are stored in an instance of the class `DYNAMIC_PRIORITY_QUEUE`. For being executed, the features are retrieved from the queue in the descending order of their dynamic priorities. Between two features that have the same dynamic priority, the choice is made random. After a feature is executed, its dynamic priority is decreased by 1. The static priority remains untouched. At every moment, the highest dynamic priority of all the features is known, and every time a feature with that dynamic priority
Figure 5.4: Tasks and strategies
is selected for execution. When all features have a dynamic priority equal with 0, their dynamic priority is reseted to the value of the static priority.

During an automated testing strategy, we need to call features using various inputs selected according to the strategy. We also need to create objects of certain types. All the operations done by a testing strategy are considered to be tasks, which can have subtasks. The most important operations performed on a task are starting it, testing if it has a next step or not, executing its next step or stopping it. At a certain moment, a task can have a single subtask. Executing a step of a task means creating a new subtask for it, if it doesn’t have one, or executing the next step of its subtask if possible. A task is executed until all its steps are finished or until the time for testing has elapsed.

The purpose of a strategy is to call features; for calling a feature, it selects a feature and creates a feature caller for calling the feature. The feature caller is the subtask of the strategy. After creating it, the strategy starts it and executes all its steps. The purpose of feature caller is to create inputs needed for actually calling the feature; hence, a feature caller creates an input creator as its subtask, starts it and executes all its steps. When it has all the needed inputs, it invokes the feature.

For creating all the inputs needed by a feature in order to be called, the input creator has an object creator. This is responsible for creating individual objects of a certain type. If that is not possible, it has to return a Void value. Objects of an expanded type are chosen from a predefined set of representative values, while for non-expanded types the object creator selects randomly one of the available creation procedures for the type of the object to be created and also creates an input creator for generating the inputs for this creation procedure. The input creator might in turn need an object creator for creating arguments for the creation procedure. This chain continues until all the necessary objects are created.

A most detailed presentation of the strategies used by AutoTest and how they are implemented will be presented in chapter 6.

5.4.3 The interpreter

As mentioned in 5.3, the interpreter is responsible for actually executing the tests. It is an independent software component that is able to execute simple commands. The requests received are parsed by an instance of the class ITP_REQUEST_PARSER. They are fulfilled and the responses are sent back to the driver. The interpreter writes all the operations it executes in its log file.

The class ITP_STORE holds the name and the value for every variable defined in the system at a certain time. It has methods for adding or deleting a variable, for assigning a value of an expression to a variable or for returning the value of a defined variable. The
The interpreter has classes for representing constants of primitive types, reference constants and variables.

The method `execute` from the class `ITP_INTERPRETER` is responsible for executing the interpreter. It calls the method `main_loop` that continuously reads input commands, executes them and reports the results of execution. The input lines representing commands are parsed using the `parse` method. Some commands that can be given to the interpreter are for:

- creating a new object;
- returning the type of an object;
- invoking a feature on an object - and assigning the result to a variable if the invoked feature is a function. The execution of a feature is protected: if they fail, they have a rescue clause where the exception trace is built and returned to the driver. The driver uses it for assessing what happened during the execution of the feature;
- storing the objects used for calling a feature by serializing them into files;
- retrieving the number of objects or the objects used for calling a feature from a file;
- computing the distance between two objects.

The class `ERL_OBJECT_DISTANCE_CALCULATOR` is responsible for computing the distance between two objects in the way described in the section 4.3. In our implementation, the maximum distance between two objects can be 3. All the obtained values for a distance are scaled into the interval \([0, 3]\). The functions and the constants mentioned in the section 4.3 are used for computing the distance between two objects.

### 5.4.4 The communication between AutoTest and the interpreter

AutoTest creates an interpreter and continuously gives commands to it for performing the tests. The order in which commands are given to the interpreter depends on the used strategy. AutoTest and the interpreter communicate by using a log file. AutoTest writes its requests, and after every request it gets a response from the interpreter. The content of the log file will be used eventually for reconstructing the execution sequence in order to build the result repository.
The classes of AutoTest that handle the communication with the interpreter are grouped under the cluster named `aut_proxy`. We have a class for each request that can be addressed to the interpreter. All classes used for implementing requests inherit from the deferred class `AUT_REQUEST`. The requests are processed by instances of classes that inherit from the deferred class `AUT_REQUEST_PROCESSOR`. For example, the class `AUT_REQUEST_PRINTER` is a descendant of `AUT_REQUEST_PROCESSOR` that serializes the requests for sending them to the interpreter.

The interpreter can respond to AutoTest’s request in a normal or in a bad way. A normal response from the interpreter means that the request has been successfully fulfilled. This does include the case when an exception is triggered by executing the request. In this case, an exception - instance of class `AUT_EXCEPTION` - is attached to the response. An exception object contains the name of the class and of the feature that triggered the exception, a code of the exception and the exception trace for the exception. If the execution of the request completed without throwing an exception, the exception associated with the response is set to Void. A bad response is received from the interpreter if the latter did an illegal operation while trying to fulfil the request.

The class that handles the communication with the interpreter is `AUT_INTERPRETER_PROXY`. It has methods for all the requests that can be sent to the interpreter. After writing the request into the log used for communication, the response from the interpreter is parsed and attached to the request. If the last response is a bad one, this means that the interpreter has performed an illegal operation and has been stopped. This class has methods for starting and stopping the process in which the interpreter is executed. By doing this the robustness of application is increased: even if the interpreter can not recover from the error produced while trying to execute something, the driver is not affected and the strategy to be executed can continue by restarting the interpreter.

For knowing at each time the name of the variables available in the interpreter and their type, the driver has a variable table implemented by the class `AUT_VARIABLE_TABLE`. It has methods for returning a random variable, a variable conforming to a certain type or all the conforming variables for a certain type. It also has a method for creating a new variable with a name generated by a generator of unique names. If the interpreter holds in its store a hash of all variables and their attached values, the driver holds in this class a hash of all variables and their types. It doesn’t need the values of variables as all the operations are done by the interpreter. It needs only their types and names in order to coordinate the testing process.
5.4.5 Generating the statistics

After the time allocated for testing has elapsed, the statistics showing the results are built. For this, the log in which the dialog between the interpreter and the driver was stored is parsed line by line and the testing process is reconstructed. The class AUT_RESULT_REPOSITORY_BUILDER is responsible for generating the result repository using the content of the log file. While parsing the log file, it creates objects for every request. The responses from the interpreter are parsed and the created response objects are attached to the corresponding requests. All the requests are stored in a request history table.

If the last request was about creating an object or invoking a feature, all the preceding requests from the history table starting from a last index are considered to form a witness for this sequence of execution. Initially, the last index is 0. After every creation or invoke request, the last index is reset to the number of elements currently in the history table when a witness is created. Depending on the response for the last request of the witness, a test case result is built for this witness. The test case can be a pass, a fail, an invalid one (if the last response has an exception indicating that the way in which arguments were created for the last called feature violated the precondition of the feature) or a bad one (if the interpreter responded badly). The test case result also contains the name of the class and of the called feature. If the last request represented a feature invocation and it was a fail, a test case number and the time for the failure are also stored in the test case. Every witness is added to the result repository.

The result repository has methods for returning all the test case results for a given class, or even for a given feature. Instances of classes AUT.HTML_STATISTICS_GENERATOR and AUT.CSV_STATISTICS_GENERATOR are used for creating the results to be displayed to the user according to the result repository.

5.5 System usage

This section presents the requirements for using AutoTest and the installation process. During the current project, we have used versions 1.1.1 and 1.2 of AutoTest.

5.5.1 Requirements

ISE Eiffel 5.6, or ISE Eiffel 5.7 - which can be downloaded from [22], are required for installing the binary distributions of AutoTest version 1.1.1, and respectively AutoTest 1.2. Any platform supported by the above can be used. AutoTest can be downloaded from [20]; in order to start the application, some environment variables must be set.
For compiling the AutoTest 1.1.1 from source, the requirements are:

- ISE Eiffel 5.6
- ERL-G 1.1.0 or higher - used for making an arbitrary Eiffel system reflectable.
- GOBO 3.4 or higher - a package containing Eiffel libraries and other utilities. The development version available on the CVS from Sourceforge must be used.
- E-POSIX 2.4.1 or higher - a complete Eiffel to Standard C and Posix binding.
- Any platform supported by the above

For compiling AutoTest 1.2 from source, the requirements are:

- ISE Eiffel 5.7 or higher
- ERL-G 1.2.0
- GOBO 3.5
- Any platform supported by the above

### 5.5.2 Starting the application

AutoTest has a command-line interface that is easy and intuitive to use. It can accept a number of command line arguments for tuning the testing process. Most of them are optional. A typical use of AutoTest from the command line is: `auto_test time_limit compilation_option set_of_classes_to_test`. `Time_limit` is the maximum time AutoTest has for exercising the software, while `compilation_options` is the name of the control file (generated by Eiffel Studio) governing the compilation of the system.

The order in which arguments are given is important. Here is a list of the most important arguments that can be provided to AutoTest:

- help : Displays all possible command-line options
- -version: Displays the current version of AutoTest.
- -verbose: Displays information about compilation progress and about the generation of the interpreter.
- **-just-test**: No new interpreter is created and a previously existing one is used. If none of the classes needed by the interpreter have changed, it is recommended to turn on this option because the generation of the interpreter is very time consuming.

- **-disable-manual**: No manual test will be taken into consideration. By default, the manual strategy is turned on.

- **-disable-auto**: Executes only manual tests. By default, the automated strategy is turned on.

- **-disable-minimize**: The minimization activity is time consuming, but important. By default, it is turned on.

- **output-dir**: Everything that AutoTest generates (the interpreter, log file, test results) is created in this directory.

- **time-out**: How long AutoTest runs, in minutes. If not provided, a default value is considered. This time includes the necessary time for creating the interpreter and the testing time, but doesn’t include the minimization time.

- **ace-file-name**: The name of the control file (ACE File, described next) for the system under test. This argument is mandatory.

- **classes+**: A list of one or several classes the user wants to test. This argument is mandatory.

There are also some other arguments representing probabilities for how often the diversification process is done or how often AutoTest should create new objects.

### 5.5.3 ACE Files

The information needed for putting together a number of Eiffel classes - one of which is the **ROOT_CLASS** - in order to produce an executable system - in a process called *assembly* - should not be included in the text of classes. For doing this, Eiffel uses ACE files (ACE stands for “Assembly of Classes in Eiffel”). Information from these files is written by using a simple control language, called Lace.

An ACE file is mainly a configuration file (a sort of makefile) that drives the whole compilation process. ACE files are automatically generated by EiffelStudio and are platform dependent. A small example of an ACE file is presented in figure 5.5.

The root class of the system is introduced by the keyword **root**. When the compiler starts to process the system described by this .ace file, it finds the text of the root
Figure 5.5: Example of an .ace file
class in the file `root.class.e`. From this text, the compiler will deduce the name of all classes needed and will search in the corresponding files. This process is repeated until all classes needed directly or indirectly by the root are processed. Various information, as enabling assertions or using optimizations is also provided.

Other types of files used are `.xace` files, that have the same role as `.ace` files but are platform independent. Recently, `.ecf` (Eiffel Configuration File) files have been introduced.
Chapter 6

Implementation of the testing strategies for AutoTest

This section presents the way in which the testing strategies used by AutoTest are implemented. AutoTest currently implements one strategy for manual testing and three strategies for automated testing. Before, AutoTest has been equipped with two automated testing strategies: the pure random strategy and the ART strategy; we have added two new strategies by which we are trying to guide the testing process in a way to uncover bugs after less number of tests than with the preceding strategies: a new version of ART that replaces the old one, and a version of ART that takes into account the objects provided by manual tests.

The testing strategy influences the way in which the objects - target and arguments - are selected for calling a feature, and doesn’t influence the order in which features are called. For implementing the two new strategies, we started from two basic ideas. The first idea was to try to implement a version of ART: guiding the process of selecting objects for calling a feature in such a way that the selected objects are in those regions of the space that have been less tested before for the respective argument position of the current feature. The classical version of ART needs to compute the distance to all the preceding used objects for a position of a feature - by a position of a feature we mean the target or one of the arguments for the feature - , and hence is not scalable; our ART version computes the distance to only a subset of all the preceding used objects. The second idea was to exploit as much as possible the information provided by the user in his manual tests and to try to select test cases that are close enough to the manual ones.

Basically, every automatic strategy implements a task-subtask model in the way described in the section 5.4.2. Every task is split into some small subtasks until the desired level of granularity is reached. The figure 6.1 illustrates what the process of splitting
tasks in subtasks means in our project. Every rectangle from the figure represent a
task of a higher or lower granularity, and the arrows represent the “has as subtask”
relationship between two tasks.

Figure 6.1: Tasks and subtasks

6.1 The support for manual testing in AutoTest

A *manual test case* is a class whose features represent tests for the system. Manual
tests are written by the user in such a way to exercise those parts of the system that
are suspected by the user that can be failure triggering. He feels that those parts of
the system must be tested in particular; hence, the manual tests are always executed
before performing any automatic test.
The classes from the universe that inherit from the class AUT_TEST_CASE represent manual test cases. Using the class AUT_MANUAL_TEST_CASE_LOCATOR, all the relevant manual tests for the current testing process are retrieved. A manual test case is relevant if it depends on a class from the testing scope. All the relevant manual test cases are executed prior to executing any automatic test.

The class AUT_MANUAL_STRATEGY implements the strategy for executing manual tests. It has an object of type AUT_MANUAL_TEST_CASE_CALLER as a subtask; this object is responsible for executing any manual test case. A step of the manual strategy consists in taking the next manual test case to be executed and creating a test case caller for it, if the execution of the preceding manual test case has finished, or in going to the next step of the execution for current manual test case. If the execution of the current test case causes the stopping of the interpreter, the interpreter is restarted and the next test case is executed.

The execution of a test case means the execution of all the test procedures of the class representing the test case. The non test procedures of the test case are skipped from execution. A test procedure is a procedure whose name starts with the word “test.”. At every step, a manual test case caller invokes a test procedure of the current test case. When all the test procedure of the current test case have been executed, the manual test case caller reports to the manual strategy the end of the execution for the current test case. When all the manual test cases have been executed, the execution of automated tests can start.

### 6.2 Implementation of pure random testing in AutoTest

The classes implementing the random strategy are: AUT_RANDOM_STRATEGY, AUT_RANDOM_FEATURE_CALLER, AUT_RANDOM_INPUT_CREATOR and AUT_RANDOM_OBJECT_CREATOR. They are grouped under the cluster aut_random. All these classes inherit from the deferred class AUT_TASK, and implement the operations all the tasks - in the sense described in 5.4.2 - must have: one for starting the task, one for executing its next step and one for canceling the task. Every of these classes is a client of the following one, using it as its subtask.

The class AUT_RANDOM_STRATEGY is responsible for coordinating the random strategy. One of its attributes represents the queue with all the features that need to be tested. After being created, the strategy is started. Starting a random strategy - with the procedure start inherited from the class AUT_STRATEGY - means just to start the interpreter. After being started, the strategy is executed step by step until
the time dedicated to testing has finished.

A step of a random strategy is represented by the calling of a feature. Basically, the step procedure has the following functionality:

- If the strategy doesn’t have a subtask, the next feature to be executed is selected according to its dynamic priority and an object of type `AUT_RANDOM_FEATURE_CALLER` that will be responsible for actually calling the feature is created. This feature caller is set as the current subtask of the strategy, and it is started. The name of the feature to be called and the type of that feature are attributes of the feature caller and they are set by the strategy.

- If the strategy has a subtask which has a next step to execute, and if the interpreter is started and ready to receive requests, which means that the execution of last selected feature is not finished yet, the next step of the subtask is executed. If the interpreter is stopped when the strategy tries to execute its next step, the interpreter is restarted, the subtask is set to `Void` and the strategy can continue by selecting the next feature to be called.

Objects of type `AUT_RANDOM_FEATURE_CALLER` are used for instructing the interpreter to call features on different targets and with different arguments. These objects call the feature indicated by the strategy, but also can call features with the purpose of diversification of the objects from the pool.

In order to call the feature indicated by the strategy, the feature caller needs a way to create a target, and eventual arguments, for the call. For doing this, the feature caller creates an input creator. After being created, the input creator is started and all its steps needed for creating all the objects are executed one at a time.

When all the inputs for creating the call are available, instead of doing the call directly, the feature caller object first tries to diversify the pool. The feature caller has an attribute, which is set by the strategy object when it creates the feature caller, and which gives the probability of doing a diversification operation after every call to a feature requested by the strategy. For doing the diversification, any random feature caller has another random feature caller as attribute. This second feature caller chooses randomly one of the objects existing in the pool, gets its type and also randomly chooses a feature available for that type. The object is then diversified by invoking the selected feature on it. The arguments needed for invoking the diversification feature are created by an input creator. When the feature for diversification is chosen, the priorities of features are not taken into consideration.

The final step of a feature caller is to invoke the feature using the objects received from the input creator. The feature is marked as being called, that means that its prior-
ity is decreased, and the communication proxy is invoked for instructing the interpreter
to execute the feature.

The responsibility of creating a list of objects needed in order to call a feature is del-
egated by the feature caller to an object of type AUT_RANDOM_INPUT_CREATOR. This object is given a list of types for all the objects needed by the feature caller and it will return a list of objects of the corresponding types. We will call these objects receivers. A receiver can be an existing object or a completely new created object of the desired type. The probability by which a new object is created is given by an attribute of the input creator object. When all the receivers are created, the input creator reports its last step and the feature caller takes the obtained receivers.

For creating a new object of a certain type, the input creator uses an instance object of the class AUT_RANDOM_OBJECT_CREATOR. If the type of the object is a primitive one, the value for the object is taken from a set of predefined values for that type or - for the types where it is possible - a new random value is generated, according to a preset probability.

For creating an object of a reference type, a creation procedure is chosen randomly from all the creation procedures available for the type and an input creator is delegated the responsibility of creating the list of arguments for the chosen creation procedure. After all the necessary arguments have been created, the proxy is used for instructing the interpreter to create the new object. The object creator reports that it has finished creating the object which is now available for the input creator.

The client relationships between the classes involved in implementing the random testing strategy are illustrated by the figure 6.2

Figure 6.2: Classes of the random testing strategy
6.3 The previous implementation of ART in Autotest

For implementing the D-ART strategy described in 4.2, we need to store all the objects used for calling every feature during the testing process. When a feature is to be called, for every of its position (target and arguments) a new object is created with a predefined probability, or an already existing one conforming to the desired type is used. When choosing one of the already existing objects, we select that object that is furthest away from all the objects that have been previously used for the current position of the feature. Hence, this version of the algorithm is not scalable; a scalable version will be presented in the following section. All the objects used for calling a feature are serialized in files with suggestive names; every time when the feature is reinvoked after restarting the interpreter the objects are deserialized from the corresponding file.

The classes used for implementing the ART strategy were organized in the same way as the classes used for implementing the random strategy. We had a class for the ART strategy, one representing the feature caller and one representing the input creator. The only major difference related with the preceding strategy occurred in the procedure step of the input creator, that used the procedure select_furthest_away for selecting one of the existing objects for the current position.

6.4 The new implementation of ART

For achieving the scalability objective for the ART algorithm, we have developed a new version of ART that implements a selection schema for choosing an object for a certain position in a feature call. For every such position, the idea is to create a number of candidates (new ones or by diversifying existing ones) and to choose for that position that candidate that better suits to an appropriate criterion.

In this version of ART, we don’t make any distinction between the manual existing objects and the automatic ones. All the objects existing in the system are treated in the same way. In particular, we allow the manual objects to be diversified. In the version of algorithm presented in the following section, we will make a clear distinction between objects coming from manual tests and the automated objects.

The basic idea of the algorithm is that, for each position of a feature to be called (target or argument), we do the following:

1. Generate N candidates for that position. With a probability of 50%, a candidate is a completely new object, the alternative being to create a candidate by diversifying an already existing object from the pool.
2. Order the candidates in an increasing order of the average distance between them and K randomly selected objects of all the previously used ones for the same position of the current feature. The object that is furthest away from the K used objects is selected as the winner of the current selection. The α closest candidates from the K selected used ones are removed from the pool.

The classes used for implementing the new ART, the relations between them and their relations with the classes used for implementing the pure random strategy are presented in the figure 6.3. The client relations are double lined while the inheritance relations are represented with a single line.

Figure 6.3: Classes of the new ART strategy and their integration in the system

The distance between any objects, once computed, is stored in a table of distances and every time it is needed again, it is taken from there. Any time a distance between two objects is different from the value stored in the distance table - because one of the two objects has changed its state through diversification -, the value from the table is updated. Even if its content is used and modified only by input creators for selecting for a position of a feature one of the candidates conforming to distances to the used objects, every class of the new strategy has a reference attribute to this table. This is because its
content must be maintained during the entire strategy execution. Every task, when creates a new subtask, sets a reference attribute of the subtask to point to this table. The only supplementary functionality the class `AUT_ADAPTIVE_RANDOM_STRATEGY` has compared to its ancestor is setting a reference attribute to this table for its subtasks after creating them.

The classes `AUT_ART_FEATURE_CALLER` and `AUT_ART_INPUT_CREATOR` contain the key functionality for implementing the new ART strategy. The class diagram involving only these classes and the client relation between them is presented in the figure 6.4

Figure 6.4: Core part of the new ART strategy

The class `AUT_ART_INPUT_CREATOR` has attributes which specify the number of candidate objects for a certain position of a feature - \( N \) -, how many of them will be retained in the pool - \( \alpha \) - and how many of the preceding already used objects will be
used for selecting the best candidate - K. The input creator will select a receiver object for a position of the feature to be executed only after having a list of all the candidates for that position - the attribute candidates -, and the way in which they were obtained, through creation or through diversification - the attribute candidate_types. We store the way in which a candidate was obtained because we will allow only the new created objects that are not among the alpha winners of the current selection to be removed from the pool. Candidates obtained through diversification are always retained in the pool.

The selection process is handled by the step method of this class. The first thing to be done is to retrieve all the previously used objects for the same position from the files where they were serialized. If there is no such object, and if less than alpha candidates were new created, we delete these candidates from the pool, otherwise we select randomly alpha of them to be deleted. After that, we choose one of the surviving candidates as the winner of the current selection and set it as the receiver for the current position.

If we do have some objects previously used for the current position, we first select randomly k of them. The average distance of every candidate to all these k objects is computed and the candidates are ordered increasingly according to this distance. For computing the average distance, we use the proxy to retrieve from the interpreter the distance between a candidate and every used object, if we don’t have this distance already computed in the distance table, or we take it from there, otherwise.

If one of the candidates is also among the k already used objects, case which can happen if the candidate is obtained through diversification, we set the average distance for that candidate to 0 for minimizing its chance for being chosen by the selection process. We use an alternative distance for deciding about the order between those candidates that are among the previously k used objects. The alternative distance is computed as being the average distance from a candidate to the other k - 1 used objects, excepting itself.

Those candidates that are among the least desirable alpha for a position are not directly removed from the pool. They are stored in a list of removable objects. The reason for this is that one new candidate that is not chosen by the selection process for a certain position can be diversified and hence be chosen by the selection process for a next position of the same feature. In this case, that object is removed from the list of removable objects. When the selection processes for all the positions of the current feature have finished, the selected feature will be invoked using the winners of the selection processes and all the objects that are still in the list of removable objects are removed from the pool.
The selection schema affects the way in which the feature caller works. If before, it used a single instance of the class `AUT_ART_INPUT_CREATOR`, now we need two such instances: one to provide the inputs for a feature called in the diversification mode - the diversify input creator - and another to provide the inputs for a feature calling as required by the testing strategy - the input creator. The need for two input creators comes from the fact that, in order to invoke a feature according to the testing strategy, more features are invoked for creating candidates through diversification.

For every position of the feature to be called according to the testing process, we will create a new candidate by diversification in 50% of cases and a complete new candidate in other cases. The choice is made in the `step` procedure of the class `AUT_ART_FEATURE_CALLER`. If we need a completely new candidate, we give the responsibility of creating it to the input creator and we execute all its steps until the new candidate is created. Otherwise, we create the diversify input creator by calling the procedure `create_input_creator_diversify_for_type` with the desired type for the candidate as an argument.

When the diversify input creator is created, a random variable of the desired type is chosen from the pool for being the target for diversification. If we don’t have such a variable, the process of diversification is canceled and, in the following steps, completely new candidates will be created. Otherwise, one of the available features for the type is randomly selected and the diversify input creator is indicated to create arguments of the types needed by the feature used for diversification. The diversify input creator will choose randomly the needed arguments, not implementing the selection schema. When the next step of the feature caller is executed by the strategy, if we have a diversify operation to finish, we do that first. When all the arguments needed for the diversifying operation are available, we invoke the diversifying feature and we set the diversify target as candidate for the current selection. The priority of a feature called for diversification purposes is also decreased. Only after finishing the diversification we step further for determining the next candidate.

We have set some conditions for creating candidates. The first one is that we don’t allow a Void candidate to compete for being the target of calling a feature. The second condition is that the list of candidates for a position of a feature doesn’t contain duplicate candidates. This could happen if a candidate is new created and it is also chosen for being diversified during the same selection process, or if a candidate is chosen for being diversified twice or more times during the same selection process. In these cases, we don’t allow the diversification and we try to select a new candidate. The third condition is that if, for a position we choose for diversifying an object that has been created as a candidate for a previous position of the same feature, but has not been
chosen by the selection process then - and hence it is now marked as being removable in the input creator -, we remove it from the list of removable objects.

When all the candidates for a position of the feature to be invoked are available, the input creator’s procedure step is invoked for choosing the winner for the current selection process in the way described previously. When the winners of all the the selection processes for the feature have been chosen, we invoke the feature and remove the candidates marked as removable from the pool.

When all the arguments are available, the procedure invoke_ is responsible for invoking features even if the feature is invoked for diversification or because it is needed by the testing strategy. Invoking a feature with the purpose of diversification is considered to be a test case too and the potential exception thrown by such a invocation is also taken into consider when the statistics are built. Also, the objects used for invoking such a feature are serialized in files for being used later if needed.

The class AUT_ART_OBJECT_CREATOR is responsible for creating new objects in this strategy. It inherits most of its functionality from its ancestors. The only modification done is in the procedure named choose_creation_procedure, which previously returned a Void object if there were no creation procedures for the specified type. In the new strategy, we mark this Void variable as not being really new even if it is created by the object creation. We need this information for not allowing the removal of it from the pool even if it is not chosen by the selection process for a certain position.

### 6.5 The implementation of the new ART strategy when the manual objects are considered

When this strategy is used, there is a clear separation between the objects involved in manual tests and the automated objects. In particular, the manual objects always remain in the pool and they are never allowed to take part at any selection process. The manual objects are never diversified, they always have the same state. Hence, we need to treat them in a different way. Some things to be done in order to do this separation are: to serialize these objects in files that have specific names - we used files whose name start with the word “manual_” -, to know at each time which objects from the pool are manual and to take them into account when assessing the desirability of the candidates for a selection.

The algorithm suffers some modifications when compared with the one presented in the preceding section. The basic idea is that, for each position of a feature to be called (target or arguments), we do the following:
1. Generate N candidates for that position. With a probability of 50%, a candidate is a completely new object, the alternative being to create a candidate by diversifying an already existing automated object from the pool. Manual objects cannot be diversified.

2. Select randomly K objects from all the previously used ones by the automatic tests for the same position of the current feature. For every candidate, compute the average distance AD to these K objects, and the average distance MD to all the objects used for the same position of the feature when the feature was tested manually. For being selected, a candidate must have a high value for AD and a small value for MD. The value with which a candidate takes part at the selection process is \( D = A \times AD + B \times MD \), where A and B are two constants for which we have chosen the values 1 and -1. The candidates are ordered increasingly according to D, and the one that has the highest value for D is chosen as the winner of the current selection. The \( \alpha \) candidates that have the smallest value for D are removed from the pool.

### 6.5.1 Serializing the manual objects into files

For finding the objects used by the manual tests, we have modified the `invoke_test_procedure` from the class `AUT_MANUAL_TEST_CASE_CALLER`; the role of this class is to call a manual test procedure of a test case, as was briefly presented in the section 6.1. We have modified the functionality of the named procedure in such a way that, for every qualified feature invocation or creation instruction the test case procedure uses, all the involved objects are serialized into a file.

The body of the test procedure might contain one or more real test calls: any qualified call or creation call that appears in the body of the test case procedure is a real test call. A real test call can have as arguments or as target any object that is a primitive constant - including strings - or any attribute of the class representing the test case. We serialize in files only the objects used for reachable test calls of the test case procedure. Those test calls of the current test case procedure that don’t get executed, because one of the preceding test call of the current test case procedure produces an exception, are considered to be unreachable and hence, the objects used for them are not saved.

The procedure `store_manual_objects` is responsible for executing a test case procedure and storing the objects used by its tests. For executing it, we first create a list of all the real test calls that appear in the body of the test case procedure. All the other instructions are ignored. After creating the list, we take every real test call, and we use...
the `store_manual_test_used_objects_for_feature` procedure of the proxy for instructing
the interpreter to store the objects used for this real test call into a file and to execute the
real test call. This procedure is given as arguments the name of the feature representing
the test call and a list of objects involved in the call. The first element of the list is the
object that represents the test case that contains the current test procedure, the second
element is the target for the current real test call and the following elements are the
arguments for the test call. If the execution succeeds, we can proceed to the following
real test call, otherwise we stop the execution of the test procedure.

The `store_manual_test_used_objects_for_feature` procedure of the proxy asks the inter-
preter to call the test and assesses, from the interpreter response, if the execution
was a success or not. If it was a failure, it indicates to the manual test case caller
that the current test procedure should be aborted. Even if the last execution was not
a success, the objects used for it are serialized in a file. A new command for storing
manual objects has been added to the interpreter.

### 6.5.2 The algorithm implementation

For making a clear distinction between the manual and the automated objects of the
system, the proxy has to retain in a list the names of all those variables that represent
manual objects. Because manual tests are always executed prior to any automatic
test, all the variables existing in the pool before starting to execute any automatic
test represent manual objects. Hence, we store them in the list of manual objects.
Besides them, when a feature that has also been tested manually is to be executed as
an automatic test, all the objects used for testing it manually are retrieved from the
corresponding file and are given new names. This happens because when an object is
serialized into file, only its value is stored, and not its name. These names are also stored
in the list with manual objects. Any time the proxy is asked to return the name of
one variable conforming to a type, it won’t return any of those variables that represent
manual objects because we don’t allow any operation to be performed on them.

The classes used for implementing the algorithm are `AUT_MANUAL_ART_STRATE-
GY`, `AUT_MANUAL_ART_FEATURE_CALLER`, `AUT_MANUAL_ART_INPUT_CRE-
ATOR` and `AUT_CLUSTER`. The relations between them and between their ancestors
are presented in the figure 6.5. We can see in the figure that for creating a new object,
the input creator will use the same object creator as the one used by the preceding
strategy.

In the same way we have a table that holds all the computed distances between
automatic objects from the pool, we also have a table that holds the distance between
all manual objects currently existing in the pool. The distance between two manual
objects never changes, hence the table is maintained during the entire execution. This is done by keeping references to the table in all the classes that implement this strategy.

The only supplementary functionality the class **AUT_MANUAL_ART_CALLER** has when compared with its ancestor is that it doesn’t allow manual objects to be used as a target or as an argument for a diversifying operation. When a diversify input creator is created for a diversifying operation needed in order to create a candidate for a certain position of a certain feature, the first thing to do is to retrieve the manual used objects for that position of the feature if they have not already been retrieved. After retrieving them, they will be known in the system and when we ask the proxy to give us a conforming variable of the desired type, as explained previously, we can be sure we have received a variable representing an automated and not a manual object.

The most important role in implementing the algorithm is played by the **AUT_MANUAL_ART_INPUT_CREATOR** class. Basically it has the same functionality as its ancestor: implementing a selection schema for choosing the best candidate for a certain position. In this strategy, the way in which the best candidate for a position is chosen is influenced by the manual objects. The class diagram containing only this class and its ancestor is presented in the figure 6.6.

The attribute **manual_testing_objects** indicates the number of the manual used objects for the current position of the current feature. A value of -1 for this attribute
Figure 6.6: The core part for implementing the new ART strategy when manual objects are taken into consider
means that we haven’t yet tried to retrieve the manual objects for the current position from the corresponding file. The first step to be done after creating the input creator is to retrieve the corresponding manual objects. This retrieval is done by calling the `retrieve_manual_objects` method of the class. It takes the manual objects from the corresponding file computes all the distances between them and put these values in the `manual_distances` table. The manual objects are also put in the table of the proxy that holds all the manual variables currently existing in the pool.

But, after retrieving them, we do something more with the manual objects that were used for the same position of the same feature: we group them into clusters of objects according to the distance between them. The clusters of manual objects are instances of the class `AUT_CLUSTER` and they have a center and one or more members. The center of a cluster is also considered a member of the cluster. In the same way we have tables for distances, we also retain a table with the available clusters for the entire duration of execution. The content of all these tables retained for the entire execution grows because the entries are permanently added and never removed from them. The strategy, the feature caller and the input creator all have references to the cluster table.

If we have a set of objects, S, we group them into clusters by using the maxi-minimum distance clustering algorithm:

1. Select the two furthest objects, create two new clusters having the two objects as centers; we use here the table that already contains all the distances between the manual objects. Remove the two objects from the initial set.

2. Select one object from the set S and compute the distance to the centers of all the existing clusters. Remove the object from the set and make it a member of that cluster that has the closest center from it.

3. Repeat step 2 until the set S becomes empty.

4. Compute the medium distance D between the centers of all existing clusters.

5. Take one of the existing clusters, C, and select that member object O that is furthest away from the center of the cluster. If the distance between that object and the center of the cluster is greater than D/2, create a new cluster with the center in O; remove all the members from the cluster C, excepting its center, and add them to S.

6. Repeat step 5 until all the clusters were processed.

7. If we have at least one new cluster, go to step 2, otherwise the algorithm can be stopped and the clusters are formed.
This algorithm is implemented by the `create_clusters` procedure. We also treat the case when we have a single object in the initial S set, and the case when we have more elements in S but the distance between every two of them is 0. In these cases, the result is a single cluster containing all the objects from the S set.

Having the manual objects used for a position of a feature grouped in clusters and a number of $N$ candidates for that position, the selection process is implemented by the `step` method of the class `AUT_MANUAL_ART_INPUT_CREATOR`. If no manual object and no automatic object have been used on that position before, we choose randomly the winner of the selection process and the candidates to be removed from the pool. If there is no automatic object but there are some manual objects used for that position, we order increasingly the list of candidates according to their minimum distance to the center of one cluster of manual objects. We remove the last $N - \alpha$ elements of the ordered list from the pool and we choose the first element of the ordered list as the winner of the selection process. We keep the rule that we remove only candidates that were new created.

If there are manual and also automatic objects used for the current position, we follow exactly the algorithm presented at the begining of this section. For every candidate we compute its average distance to every automatic object - $DA$ - and its average distance to the centers of all clusters with manual objects - $DM$. The candidate takes part to the selection with the value $D = DA - DM$. We order the candidates increasingly according to the value of $D$ and we choose the one with the maximum value as the winner of the selection.
Chapter 7

Adding more statistics to AutoTest

All the ideas presented in the previous two chapters were implemented using the version 1.1.1 of AutoTest. Before implementing the new strategies, we have used the version 1.2 of AutoTest for adding to the tool the possibility of displaying more detailed statistics for the user. There are no conceptual differences between the two versions of AutoTest, so the functionality, architecture and structure remain the same. The version 1.2 of AutoTest has been developed to work with Eiffel Studio 5.7.

The following sections present the enhancements we made to the statistics presented to the user. The modifications we have done are generally related to the part of AutoTest that generates the results and to the part of AutoTest that communicates with the interpreter. Most of the classes that were modified or added can be found in the clusters aut_statistics and aut_proxy.

7.1 Counting the number of faulty lines in the tested code

The first supplementary characteristic we have added to AutoTest was the possibility of counting how many lines from the tested code were failure triggering. This means that the corresponding lines contain one or more bugs. Two failures provoked by the same line of the same feature from the same class represent the same bug.

The features are executed by the interpreter in the procedure execute_protected. This procedure has a rescue clause: when the execution of the feature fails, the code of the rescue clause writes information about the exception that occurred in the log file used for communicating with the proxy. The interpreter uses the class EXCEPTIONS for finding the code of exception, the tag for the exception - a label associated with the place that triggered the exception - , the name of the feature and of the class whose
code has triggered the exception and the exception trace. We have modified this code in such a way that it also writes in the log file the same information about the original exception. The difference between exception and original exception is important in the case of “organized panic”: if a routine gets an exception of code A - caused for example by a Void target in a feature call - but has no rescue clause, its caller caller will get an exception whose code indicates “failure of the called routine”, that has nothing to do with the real source for exception. The attribute original_exception of the class EXCEPTIONS contains information about the real source of exception that we need when we want to count the number faulty lines.

On the driver side, exceptions are represented by the class AUT_EXCEPTION. This class has an attribute for every information we need about an exception. The original exceptions are also instances of this class. The class that parses the responses from the interpreter and creates objects representing the responses was modified in such a way to parse the original exception too. An object representing a normal response from the interpreter that has an exception will also contain the original exception. Then, the response is set to its corresponding request and the request is added to the history table that contains all the requests.

After the process of testing is finished the result repository is built by processing the list of all the requests and their attached responses. A witness is created from a sequence of requests. According to the response of the last request from the sequence of requests, the witness will have a list with classifications - test case result objects that contain the name of the class, of the feature, if the execution was a failure or not and so on - for the sequence of executions. We have also added to it a list of faults met during this witness. A fault is an instance of the class AUT_TEST_CASE_FAULT and contains as attributes only the original exception indicating the cause of failure and the witness that has uncovered the fault. The list of faults from a witness can contain only distinct faults. The class AUT_FAULT_EQUALITY_TESTER was written for comparing two faults. Two faults are equal if their exception attributes have the same class, feature, tag and code. After being created, the witness is added to the result repository.

The result repository contains the list of all the obtained test case results. We have also added to it a list of all the faults encountered during the testing process. The faults are grouped by the class that contain them. When a witness is added to the result repository, its list of results and its list of faults are appended to the corresponding list of the result repository. The result repository has methods for returning the set of all the results for a certain class or for a certain feature of a class. We have modified these methods to also return the list of all faults for the class or for the feature. Hence, the class responsible with generating the HTML will know for every class and feature the
number of faults and will display it.

### 7.2 Showing the time elapsed until the first failure was triggered

During this task we wanted to count, for every failure, the number of seconds that elapsed since the testing process was started until the failure was triggered. Also, the time to first failure for the entire system was needed.

The first thing we did was to write the date and the time when the testing process has started in the log file. The `execute_protected` procedure of the interpreter was modified in such a way that the interpreter writes in the log file the date and the time after every feature execution that triggered an exception.

The class that parses the responses from the interpreter will also parse the time and the date for a response that contains an exception. The time and date are stored in the object representing the response, and this one is attached to the corresponding request. The request is then added to the list containing the history of requests.

When a witness is created from a sequence of requests, it is also given the value for the date and time when the testing process has started. The witness computes the number of seconds between this starting time and the time when a failure has occurred. The number of seconds is stored in the classifications - test case results - done for the witness. When the witness is added to the result repository, its test case results are added to the list of results. All those test case results that represent failures have the number of seconds attached to them. The result repository will be able to return the minimum time in seconds for a set of results grouped by class or by class and feature. This time is displayed by the statistics generator.

Using the faults list created in the result repository as described in 7.1, the HTML statistics generator will display, for the whole system, the total number of faults, how many of them are in the classes under test and how many are in the other classes. Also, the time for the first failure caused by a fault from a class under test and the time to first failure caused by a fault in another class are displayed.

### 7.3 Counting the number of faults of different categories

A failure can have different causes: the breaking of an assertion, the calling of a feature on a `Void` target, trying to assign a `Void` value to an expanded variable and so on. We
counted the number of the faults of each category and printed them for every class,
every feature and the whole system.

The category for a fault can be obtained from the original exception that proves the
existence of the fault. When a witness creates a classification for a sequence of requests,
if the response of the last one is a fail we save an object representing the original
exception as an attribute of the classification. The name of the original exception
represents the category of the fault. So, for a set of test results we can count how many
of the test cases that triggered failures are of each category. Using this information,
the statistics generator prints the number of faults of every type. If we don’t have any
fault in a category, we don’t print anything for it.

In the case in which a precondition of a feature is violated, the caller of the feature
is the one that contains a fault, even if the original exception indicates the class and
the feature whose precondition was violated as being buggy. In this case, we determine
the real place of the fault by parsing the exception trace of the original exception.

7.4 Creating a “comma-separated-values” file with
the results

The class AUT_FILE_GENERATOR is responsible for creating the .CSV file that con-
tains the statistics. For every tested class from the result repository we add to the file
information like the time to first failure, number of faults, how many of the are in the
class itself and of many are in other classes number of preconditions, postconditions or
class invariants violations, number of floating point exceptions etc.
Chapter 8

Results and conclusions

The main results of testing are not only the number of bugs found in software, but also the time elapsed and the number of tests done until the first failure is triggered. All these aspects should be taken into consider when comparing various testing strategies.

The results obtained by AutoTest are highly dependent on the quality of contracts the tested code is equipped with. For making a comparison between the performance attained by AutoTest using each of the testing strategies, we have tested the system on some library classes like STRING, LINKED_LIST, BOUNDED_STACK, ARRAY or ARRAYED_LIST.

The number of found bugs also depends on the time devoted to testing. We have tested every of these classes for 10 minutes. Other factors that have an influence on the obtained results are the seed used by the random number generator and the probabilities used by the system for making diversifying operations or for creating new objects. We have used the default values for these parameters for all the tests we have done.

In particular for our two new strategies, the results are influenced by the parameters used by each strategy: the number of candidates that are needed for a selection process: \( N \), the number of already used automatic objects considered when assessing the candidates: \( K \), and how many candidates of a selection process are retained in the pool \( \alpha \). An example of the obtained results for class ARRAY when various values were used for these parameters are presented in the table 8.1. The last two columns give the number of test cases and the time in second until a bug is uncovered in a feature that is not a creation procedure.
The new strategies trigger the first failure after a less number of test cases, as expected, but they need a longer time for doing this. This is explainable because, when the process of testing is guided, more time is needed for building inputs for a test. Another thing to be mentioned is that we haven’t considered the creation operation as test procedures - because many creation instructions are needed by the new strategies for making candidates. In spite of this fact, the creation instructions may also contain bugs, which are counted when the total number of bugs is calculated; the bugs from the creation procedures don’t influence the time and the number of tests until a failure is triggered because the creation procedures are not considered to be test procedures.
The parameters that influence most the testing results are \( N \) and \( \alpha \). The parameter \( K \) has not so much influence in our tests because the time devoted to testing is too small, and the pool won’t contain too many objects for a position of a feature. For the classes we have tested, we obtain the best results with the new strategies for the values \( N = 3 \), \( \alpha = 2 \) and \( K=5 \).

The results obtained for all the tested classes with these values for the parameters are presented in the table 8.2.

<table>
<thead>
<tr>
<th>Class</th>
<th>Strategy</th>
<th>Bugs</th>
<th>Time to first failure</th>
<th>Tests to first failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>Random</td>
<td>7</td>
<td>23</td>
<td>208</td>
</tr>
<tr>
<td>STRING</td>
<td>ART</td>
<td>5</td>
<td>117</td>
<td>209</td>
</tr>
<tr>
<td>STRING</td>
<td>Manual ART</td>
<td>5</td>
<td>87</td>
<td>51</td>
</tr>
<tr>
<td>LINKED_LIST</td>
<td>Random</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>LINKED_LIST</td>
<td>ART</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LINKED_LIST</td>
<td>Manual ART</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>BOUNDED_STACK</td>
<td>Random</td>
<td>10</td>
<td>3</td>
<td>26</td>
</tr>
<tr>
<td>BOUNDED_STACK</td>
<td>ART</td>
<td>8</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>BOUNDED_STACK</td>
<td>Manual ART</td>
<td>8</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>ARRAY</td>
<td>Random</td>
<td>21</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>ARRAY</td>
<td>ART</td>
<td>10</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>ARRAY</td>
<td>Manual ART</td>
<td>12</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>ARRAYED_LIST</td>
<td>Random</td>
<td>16</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>ARRAYED_LIST</td>
<td>ART</td>
<td>10</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>ARRAYED_LIST</td>
<td>Manual ART</td>
<td>9</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 8.2: Results obtained for \( N = 3 \), \( \alpha = 2 \) and \( K=5 \)

Another major influence over the testing process is given by the quality of manual tests provided by the user. Our manual tests were simple ones: they contain just an instruction that create an object of the tested type. If we had written some more relevant manual tests, the results for the Manual ART strategy would have been better when compared with the other strategies.

The feature calls used for diversification are considered to be test procedures. The number of diversification influence the results because the inputs for a feature used for diversification are built in a random and not a guided manner.

An example of results obtained during the first part of the project, when the statistics provided by AutoTest were enriched, is shown in the figure 8.1. These results are for a simple system that contains only one class and that was developed just for testing.
purposes. These statistics can be compared with the statistics provided by AutoTest initially and shown in the figure 5.1.

**Figure 8.1:** Enriched results provided by AutoTest for an entire system

The results for the only buggy routine of the system are shown in the figure 8.2.

**Figure 8.2:** Enriched results provided by AutoTest for a buggy routine
Chapter 9

Directions for future work

There are numerous possibilities for further developing AutoTest. Among them are:

- Adding a graphical user interface. This would make AutoTest even easier to use.
- Integrating other automatic testing strategies. Some strategies that don’t use the distance between objects - like the ones presented in section 4.2 - can also be considered.
- Going from black-box testing to white box testing. By doing this, the precondition of a feature can be parsed before choosing input objects for that feature, avoiding the situation when the chosen objects don’t satisfy the precondition - as it was the case with invalid test cases. White box testing would allow us to see the real code coverage and create test cases that try to cover most of the source code.
- Integrating a regression testing mechanism. A database should be used for storing the test cases that have previously triggered failures. These test cases should be run first when the system is tested again.
- Making the order of the input parameters irrelevant.
- Allowing the test scope to be specified at the level of features too, not only classes.
Bibliography


