DISS. ETH NO. 18143

Strategies for Random Contract-Based Testing

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
ILINCA CIUPA
Dipl. Eng., Technical University of Cluj-Napoca, Romania

born
November 11th, 1980

citizen of
Romania

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Thomas Gross, co-examiner
Prof. Dr. Mark Harman, co-examiner

2008

Acknowledgements

First and foremost I thank Bertrand Meyer, for giving me the opportunity to do a PhD under his supervision at ETH Zurich. His support and guidance made the work presented here possible and had the decisive influence on this work's progress and finalization, as well as on my formation as a researcher in the past four years.

I also thank the co-examiners of this thesis, Thomas Gross and Mark Harman, who kindly made time in their busy schedules to co-referee this work.

I am very grateful to Alexander Pretschner, Patrick Eugster and Manuel Oriol, who have taught me a lot about what research is and whose advice shaped my work and its outcome. Special thanks also go to Andreas Leitner, who has been a great colleague and friend, and was always available for discussions on our common research interests.

The other present and past members and visitors of the Chair of Software Engineering at ETH Zurich have also been an important support for me, so my thanks also go to them: Karine Bezault, Arnaud Bailly, Stephanie Balzer, Till Bay, Claudia Günthart, Benjamin Morandi, Yann Müller, Marie-Helene Nienaltowski, Piotr Nienaltowski, Martin Nordio, Michela Pedroni, Marco Piccioni, Nadia Polikarpova, Bernd Schoeller, Wolfgang Schwedler, Vijay d'Silva, Stephan Van Staden, Jason (Yi) Wei. I also thank the students who worked with me during various semester, diploma, and master projects.

I thank Gustavo Alonso, for providing us the hardware infrastructure used in some of the experiments reported here.

My family and friends have been an important support for me. I especially thank Alexandra Ciupa, Radu Ciupa and Daniel Moser, without whose constant support, patience and understanding I could not have carried out this work.

CONTENTS

ABSTRACT

Testing plays a central role in software quality assurance. Although this is widely agreed upon, many software projects in industry suffer from improper testing. Inadequate testing tools and underestimation of the costs of thorough testing are among the reasons.

The research community has been trying to address some of the challenges of testing through automated solutions, which aim at taking most of the burden of devising, running, and evaluating tests off the shoulders of programmers and testers. This thesis is part of this effort: it proposes several strategies for automatically testing contracted object-oriented software, that is software built according to the principles of Design by Contract.

One of these strategies is based on the notion of "object distance": a measure of how different two objects are, taking into account the objects' types, their direct values, and a recursive application of the distance on their common attributes. Using this notion, the proposed strategy (called ARTOO) selects test inputs with the greatest distance between them and thus maximizes the diversity of the inputs.

Proposing a new testing strategy must be accompanied by a thorough evaluation of the performance of this strategy, both in absolute terms and compared to other testing strategies. This thesis hence also presents the results of such evaluations, for the newly introduced algorithms and for existing ones. These results offer insights into the performance of random testing for object-oriented software on such issues as the evolution of the number of found faults over time, the predictability of random testing, and the types of faults found through random testing, through manual testing and by users of the software. Experimental evaluation indicates that the newly introduced ARTOO strategy requires on average 5 times less tests to uncover faults than a random strategy combined with boundary-value testing.

The proposed and examined testing strategies are based on the presence of executable specification in the form of contracts in the software under test. A further contribution lies in evaluating how automatically inferred contracts can be used to improve manually-written ones. A case study we performed shows that the two approaches are complementary: an assertion inference tool generates more contracts than programmers write, but also does not find all manually-written contracts.

The contributions of this thesis hence include: proposing new automated testing algorithms, investigating ways of improving the executable specification embedded in software — a central component in automated testing — and evaluating the performance of existing and of the newly-introduced testing strategies.

ZUSAMMENFASSUNG

Testen spielt eine zentrale Role in der Qualitätssicherung von Software. Obwohl man sich dessen bewusst ist, leiden viele Softwareprojekte in der Industrie an mangelhaftem Testen. Unpassende Tools und die Unterschätzung der Kosten für gründliches Testen sind unter anderem Gründe dafür.

Die Forschungsgemeinschaft versucht, einige der Schwierigkeiten beim Testen durch automatisierte Lösungen anzusprechen. Das Ziel solcher Lösungen ist, die Programmierer und Tester vom grössten Teil des Entwerfens, der Ausführung und der Auswertung der Tests zu entlasten. Diese Dissertation ist Teil dieser Anstrengungen: Sie führt mehrere Strategien für das automatisierte Testen von objekt-orientierter und mit Verträgen versehener Software (d.h. Software die nach den Prinzipien von Design by Contract entwickelt wurde) ein.

Eine dieser Strategien basiert auf dem Begriff "Objektdistanz", einem Mass für die Verschiedenheit zweier Objekte, unter Berücksichtigung ihrer Typen, ihrer direkten Werte und einer rekursiven Anwendung der Distanz auf ihre gemeinsamen Attribute. Die vorgeschlagene Strategie (namens ARTOO) benutzt dieses Mass um maximal distanzierte Testinputs auszuwählen und auf diese Weise die Diversität der Inputs zu maximieren.

Das Vorschlagen einer neuen Teststrategie muss von einer gründlichen Analyse ihrer Leistung begleitet werden, sowohl in absoluter Hinsicht als auch im Vergleich zu anderen Teststrategien. Daher stellt diese Dissertation die Ergebnisse solcher Evaluierungen für die neu eingeführten und die bereits vorhandenen Algorithmen dar. Diese Ergebnisse geben Einblick in die Leistung zufallsbasierten Testens für objekt-orientierte Software in Fragen wie der Entwicklung der Anzahl gefundener Fehler pro Zeit, der Vorhersagbarkeit für zufallsbasiertes Testen und der Arten von Fehlern, welche durch zufallsbasiertes Testen, durch manuelles Testen und von den Benutzern der Software gefunden werden. Die experimentelle Evaluierung deutet darauf hin, dass die neu eingeführte ARTOO Strategie durchschnittlich fünfmal weniger Tests braucht um Fehler zu finden als das Testen mit einer Zufallsstrategie kombiniert mit Grenzwerten.

Die vorgeschlagenen und begutachteten Teststrategien basieren auf der Existenz von ausführbaren Spezifikationen in Form von Verträgen in der zu testenden Software. Ein weiterer Beitrag dieser Dissertation evaluiert, wie automatisch abgeleitete Verträge benutzt werden können um die manuell geschriebenen Verträge zu verbessern. Eine von uns durchgeführte Fallstudie deutet darauf hin, dass die beiden Ansätze komplementär sind: Ein Tool, das Verträge ableiten kann, generiert mehr Verträge als Programmierer schreiben, findet aber nicht alle manuell geschriebenen Verträge.

Die Beiträge dieser Dissertation umfassen folglich: Die Einführung neuer automatisierter Testalgorithmen, das Erforschen von Methoden, welche die ausführbaren und in der Software integrierten Spezifikationen — ein zentraler Bestandteil automatisierten Testens — verbessern und die Evaluierung der Leistung der bereits vorhandenen und neu eingeführten Teststrategien.

CHAPTER 1

TESTING AND QUALITY ASSURANCE

Software quality assurance is a complex process involving many activities and spanning the entire life cycle of a software product. The purpose of this process is to ensure that the resulting software meets a set of quality measures, relating both to the external, users' view of the product, and to the internal or developers' view of it.

In the effort to deliver products on time and within budget, industrial projects often end up purposely sacrificing one or several of these quality measures. A quality factor that should never be sacrificed is correctness: the property that the software conforms to its specification, or, in other words, implements the functionality specified in its requirements. Thus, ensuring software correctness is a central activity involved in quality assurance.

There are two main approaches to checking correctness: proving software correct through formal techniques, and proving it incorrect. Testing can do the latter: it can prove software incorrect by revealing the presence of faults in it. Testing is currently the most widespread method of achieving this. This makes testing combined with debugging, the subsequent process of identifying and removing the faults, a key element in improving software quality.

This chapter provides a brief overview of some of the current directions in testing research, focusing on the areas of contribution of this thesis: automated testing — in particular random contract-based testing — and evaluations of testing strategies.

1.1 Challenges of testing

Although the purpose of testing, finding faults, and the role testing plays in the quality assurance process are clear and well defined, testing itself is by no means a simple activity. Evidence of this are the numerous challenges that the software industry is facing in this field: inadequate testing methods and tools, difficulties in estimating the time and effort necessary for proper testing, difficulties in determining what “proper testing” is and how it should be performed, the inability of adapting the state-of-the-art in research on testing to their particular settings and needs.

Many research groups are trying to address these challenges. The results are a great variety of testing strategies, some fully and some partially automated, using different algorithms for generating inputs, such as purely random and directed random algorithms, symbolic execution, genetic algorithms, coverage-driven methods, combinations of static and dynamic analysis, etc. Some approaches generate test scripts from test specifications, others from models of the software, others from formal specifications. Others try to combine manual and automated testing.

Yet research on testing does not only involve developing new testing algorithms and tools implementing them, but also evaluating these tools both in absolute terms and in comparison to other tools, clearly defining their applicability and strong and weak points, defining testing best practices, evaluating tests (for instance through coverage measures or through mutation testing), studying testing behavior, investigating what can be automated and what not, classifying faults, performing static and dynamic analysis, performing theoretical and empirical case studies, defining domain-specific testing strategies, defining best practices in the context of test-first programming and test-driven development, etc. The result of recent research into software testing is thus a wide variety of testing strategies, tools, metrics, and studies.

1.2 Automated testing

Much of the recent research develops automated solutions: push-button testing tools that require no human intervention. Such a tool must automate all stages of the testing process: input generation, test script generation, test execution, evaluation of results (the oracle), and optionally also a component estimating the quality of the tests and a feedback loop from this component and from the result analyzer to the input generator. Out of all these, arguably the most difficult to automate are input generation and the oracle.

Random testing

As shown above, many strategies are possible for automatically generating inputs. One family of strategies is *random testing*, based on random generation of candidate inputs and then random selection from them of those to be used in tests. Variations of this, based on providing some guidance to the process, are also possible.

Random testing presents several advantages: wide practical applicability, ease of implementation and of understanding, execution speed, lack of bias. In the testing literature, these advantages of the random strategy are often considered to be overcome by its disadvantages. However, what stands behind this claim often seems to be intuition, rather than experimental evidence. Furthermore, several random testing tools developed in recent years (such as JCrasher [47], Eclat [124], RUTE-J [16], Jartege [122], Jtest [4]), both in academia and in industry, suggest that this view is currently changing.

Contract-based testing

Fully automated testing requires an automated oracle. This oracle can be as coarse as using any thrown exception as an indication of a failing test case or it can use executable specification as a means of checking the test case result. The latter is clearly the more precise method and is hence preferable if specification checkable at runtime is available.

The presence of executable specification embedded in the source code has several further advantages:

- It is a documentation aid: it provides clients of a software module with information about the services that the module offers and under what conditions it offers them.
- It supports developers in the analysis and design phases of software development by helping them identify the various software modules and define their responsibilities.
- It greatly aids testing (both manual and automated, as outlined throughout this thesis) and debugging, by providing information about the locations of faults in software.
- All the above-mentioned properties make it a key element in ensuring software quality.

Thus, software written according to the principles of Design by Contract lends itself very well to *contract-based testing*: automated testing that uses routine (method) preconditions as filters for invalid inputs and all other contracts as an automated oracle.

1.3 Evaluating testing strategies

Progress in testing requires evaluating the effectiveness of testing strategies on the basis of hard experimental evidence, not just intuition or a priori arguments. Thus, any newly introduced strategy must be evaluated, both in terms of its absolute performance and compared to other strategies, and guidelines must be provided to practitioners for choosing the testing strategy or tool best adapted to their settings.

Evaluations of testing tools can take into account several measures, but must focus on the purpose of testing: finding faults in software. Hence, key for any automated testing tool are:

- Its *effectiveness*: the number of faults it finds in a certain time or after generating and running a certain number of test cases

- Its *efficiency*: the time or the number of generated test cases necessary for it to find the first fault or a certain number of faults

We consider these to be the essential properties of a testing tool, but several other measures are also important: how does the number of faults found by the tool evolve over time and over the number of generated test cases; in what domain or under what assumptions is the tool most effective and efficient; what parameters influence the performance of the tool. Other measures are also often used in the literature, such as various versions of code and data coverage, but one must not forget that such measures first must be connected to the main purpose of testing, that of finding software faults, before they can be considered indicative of the quality of tests. Depending on the analyzed strategies, the answers to these questions can be investigated both theoretically and through empirical studies.

Once such measures are available, testing strategies can be objectively compared and testing practitioners can choose the tools that suit their needs best. Such data would also make evident the drawbacks of existing tools and thus the directions that future research into software testing must follow.

Roadmap

The rest of this thesis is organized as follows. Chapter 2 briefly outlines the main results and contributions. Chapter 3 explains the basic concepts of the Eiffel language and of the Design by Contract software development method. Chapter 4 presents an overview of the state of the art in software testing, evidencing the work that is the most closely related to the contributions of this thesis. Chapter 5 describes the automated testing tool that is used as a basis for most of the developments performed as part of this thesis. Chapter 6 presents the new strategies for test input generation and selection proposed in the thesis. In chapter 7 we describe the results of an investigation into improving contracts written by programmers with automatically-inferred assertions. The results of case studies examining the performance of random-based testing strategies are the object of chapter 8. Finally, chapter 9 lists directions for future work and chapter 10 draws conclusions.

CHAPTER 2

OVERVIEW AND MAIN RESULTS

This thesis is part of the research effort of improving existing testing strategies, proposing new ones and evaluating their performance. The basis for the work was AutoTest, a push-button testing tool applicable to contract-equipped Eiffel classes. AutoTest implements a random strategy for generating inputs, filters the inputs through the preconditions of the routines under test, and monitors contracts at runtime: any contract violation and any other exception triggered while running the tests are indicative of a fault in the software under test.

One of the contributions is the notion of *object distance*: a measure of how different two objects are, taking into account the objects' types, their direct values, and a recursive application of the distance on their common attributes. Based on this notion, this thesis proposes a strategy for selecting test inputs that tries to maximize the diversity of the inputs. This strategy is called *ARTOO* (Adaptive Random Testing for Object-Oriented software) and was implemented as a plug-in for AutoTest. Experimental evaluations of its performance show that ARTOO and the random strategy have different strengths: ARTOO requires on average 5 times less tests to find the first fault, but about 1.6 times more time, due to the overhead introduced by the distance calculations and other extra computations. The experiments also show that ARTOO and the random strategy do not find the same faults, so ideally they should be used in combination.

We also explored ways of *integrating manual and automated testing*, by using the information contained in manual tests to drive the automated input selection process. Experiments showed that such a combined strategy can reduce the number of tests to first fault by a factor of 2 compared to the basic implementation of ARTOO.

The quality of the results produced by any contract-based testing tool is inherently dependent on the quality of the contracts present in the code. We hence investigated ways of *improving these contracts by means of assertion inference tools*. A tool that has attracted considerable attention is Daikon [55]. We developed an Eiffel front-end for Daikon, which allowed us to compare the assertions generated by such a tool to the contracts written by programmers. The results of a case study we performed showed that assertion inference tools can be used to strengthen programmer-written contracts (on average, such a tool generates around 6 times more correct and interesting assertion clauses than programmers write), but the assertion inference tool only finds around half of the programmer-written contracts.

Another contribution of this thesis lies in *evaluations of various random-based testing strategies*: we performed large experimental studies to investigate several questions about the performance of random testing and to find the most effective algorithm for random testing. Among the results:

- The number of new faults found per time unit by random testing is inversely proportional to the elapsed time
- The number of found faults has an especially steep increase in the first few minutes of testing
- On average, random testing finds more faults through contract violations than through other exceptions
- Random testing is predictable in terms of the relative number of faults it finds in a certain time, but it is not predictable in the actual faults it finds
- Random contract-based testing and manual testing reveal different kinds of faults, and in turn these faults are different from the ones that users report; none of these three strategies for finding faults subsumes any of the others
- Random contract-based testing reveals more faults in the specification (contracts) than in the implementation, while for manual testing and user reports the situation is reversed

The contributions of this thesis hence span a variety of activities involved in advancing the state of the art in software testing: proposing new testing algorithms; investigating ways of improving executable specification embedded in software, a central component in automated testing; evaluating the performance of existing and of the newly-introduced testing strategies.

CHAPTER 3

EIFFEL AND DESIGN BY CONTRACT

The tools implemented as part of this thesis and the performed studies are all based on the Eiffel language [108]; applicability to other languages is discussed in the respective sections. Therefore, in this chapter we provide a brief overview of the basics of Eiffel and the Design by Contract software development method, essential to the results presented in the following chapters. Also, Eiffel uses slightly different vocabulary than other similar languages, hence we introduce here some terms which we use throughout the thesis.

3.1 The basics of the Eiffel language

Eiffel is a purely object-oriented (O-O) language. It uses static typing and dynamic binding and supports multiple inheritance and genericity. The type hierarchy has a common root: class *ANY* from which all other classes inherit by default.

Eiffel supports two kinds of types: *reference types* and *expanded types*. An entity declared of a reference type *C* represents a reference that may become attached to an instance of type *C*, while an entity declared of an expanded type *C* directly denotes an instance of *C*. A special case of expanded types are the *basic types* (also called "primitive" in other languages), such as *INTEGER*, *REAL*, *CHARACTER*, *BOOLEAN*, etc. The instances of these types are also objects, but they are implemented through special compiler support. Class *NONE*, which exists only in theory, inherits from all reference types, cannot be inherited from and has only one instance: the special value **`Void`**, denoting an unattached reference, the equivalent of `null` in other programming languages.

Eiffel does not support routine overloading, so all routines in a class must have different names.

Eiffel does not use keywords to express the export status of features; rather, it allows the specification of a list of classes to which features are available. For instance, features exported to classes *A* and *B* will be callable from *A*, *B* and their descendants. Hence, features exported to *ANY* are available to all classes (the equivalent of the `public` access modifier in Java/C#) and features exported to *NONE* are not callable from outside their class (the equivalent of the `private` access modifier in Java/C#).

Classes can be organized in *clusters*, which are simple administrative units and have no scoping effect. Two clusters in a software system should hence not contain two classes with the same name, or one of the classes must be excluded from the system.

3.2 Terminology

An Eiffel class has a set of *features* (operations), which can be either *routines* or *attributes*. In Java/C# terminology, features are called "members" and routines are called "methods". Eiffel makes a distinction between *functions*, that is routines returning a result, and *procedures* — routines that do not return a result. Objects are created through calls to *creation procedures*, known in Java/C# as "constructors". Creation procedures in Eiffel do not have to conform to any naming scheme; they are normal procedures, which acquire the special status by being declared in the **`create`** clause of a class.

Eiffel also distinguishes between *commands*, that is features that do not return a value (procedures), and *queries*, that is features that do return a value (attributes and functions).

Eiffel uses the notion of *supplier* to denote a routine or class providing a certain functionality and *client* for a routine or class using that functionality. Hence client-supplier and inheritance are the two fundamental relationships between classes.

Eiffel routines and classes can be *deferred* (or "abstract" in Java/C# terms). A class having one or several deferred routines must be deferred itself and cannot be instantiated. A class can be declared deferred even if it does not contain any deferred routines.

3.3 Design by Contract

Eiffel supports the Design by Contract software development method [107], through which classes can embed specification checkable at runtime. *Contracts* are the mechanism allowing this: *routine preconditions* specify conditions that must be fulfilled by any client upon calling the routine; *routine postconditions* specify conditions that must be fulfilled when the routine is done executing. Preconditions are thus an obligation for the client, who has to fulfill them, and a benefit for the supplier, who can count on their fulfillment; conversely, postconditions are an obligation for the supplier and a benefit for the client. Another type of contract are *class invariants* — conditions that must hold whenever an instance of the class is in a visible state, that is, after the execution of creation procedures and before and after the execution of exported routines. Satisfying the class invariant is the responsibility of the class itself.

Routine pre- and postconditions and class invariants should be written already in the design phase of the software system and are part of the interface of the class. Eiffel also supports assertions that are purely implementation-related. These are:

- Loop invariants — conditions that must hold before and after each execution of the loop body
- Loop variants — integer expressions that must always be non-negative and must be decreased by each execution of the loop body
- **`check`** assertions — conditions that can appear inside routine bodies, expressing properties that must hold when the execution reaches that point; they are similar to the `assert` statements of C/C++

For all these assertions, Eiffel uses simple boolean conditions, with the exception of postconditions, which can contain the **`old`** keyword. This keyword can be applied to any expression and denotes the value of the expression on routine entry. Calling functions from assertions is allowed and greatly increases the expressiveness of Eiffel contracts, but it also introduces the possibility of side effects of contract evaluations.

Any Eiffel assertion can be preceded by a tag, followed by a colon, as in `balance_positive: balance >= 0`, where `balance` is an integer variable in scope.

Runtime contract checking can be enabled or disabled. It is enabled typically during the development phases of a software system and disabled in production mode, due to its high performance penalty. Contract violations are signaled at runtime through exceptions.

The support for Design by Contract in Eiffel is essential to the use of executable specification by programmers in this language. This is made clear by a study [31] which shows that Eiffel classes contain more assertions than classes written in programming languages that don't support Design by Contract. In the classes examined in the study, 97% of assertions were located in contracts rather than in inline assertions.

Eiffel was the first language to support Design by Contract in this form. Since then, many other languages natively contain or have introduced features supporting executable contracts in the form of pre- and postconditions and class invariants; among them, some of the best known are JML [96] (the Java Modeling Language) and Spec# [20].

3.4 Example

To illustrate some characteristics of Eiffel code and its support for Design by Contract, listing 3.1 shows an example of an Eiffel class which implements the basic functionality of a bank account.

make is the creation procedure and it cannot be used as normal procedure too, because it is exported to *NONE* as normal procedure. The attributes of class *BANK_ACCOUNT* are *balance*, an integer representing the balance of the bank account, and *owner_name*, a string representing the name of the owner of the bank account. The invariant of the class states that the balance of the account must always be positive and that the name of the owner should not be void and empty. Class *BANK_ACCOUNT* implements the basic operations that can be performed on a bank account: depositing and withdrawing money, and transferring an amount between accounts. The routines implementing these operations all have corresponding pre- and postconditions.

```
class BANK_ACCOUNT

create
  make

feature {NONE} -- Initialization

  make (name: STRING)
      -- Create a new bank account with default balance and
      -- owner 'name'.
    require
      valid_name: name /= Void and then not name.is_empty
    do
      owner_name := name
    ensure
      name_set: owner_name = name
      default_balance: balance = 0
    end

feature -- Basic operations

  deposit (n: INTEGER)
      -- Deposit amount 'n'.
    require
      n_positive: n >= 0
    do
      balance := balance + n
    ensure
      balance_increased: balance = old balance + n
    end

  withdraw (n: INTEGER)
      -- Withdraw amount 'n'.
    require
      n_positive: n >= 0
      can_withdraw: balance >= n
    do
      balance := balance - n
    ensure
      balance_decreased: balance = old balance - n
    end

  transfer (n: INTEGER; other: BANK_ACCOUNT)
      -- Transfer amount 'n' from current account
      -- to 'other'.
    require
      n_positive: n >= 0
      can_withdraw: balance >= n
      other_not_void: other /= Void
      no_transfer_to_same_account: other /= Current
    do
      withdraw (n)
      other.deposit (n)
    ensure
      withdrawn_from_current: balance = old balance - n
```

```
            deposited_in_other: other.balance = old other.balance
                + n
        end

feature -- Status report

    owner_name: STRING
            -- Name of owner

    balance: INTEGER
            -- Sum of money in the account

invariant
    positive_balance: balance >= 0
    valid_owner_name: owner_name /= Void and then not
            owner_name.is_empty

end
```

Listing 3.1: Example of an Eiffel class implementing the basic functionality of a bank account.

CHAPTER 4

APPROACHES TO TESTING OBJECT-ORIENTED SOFTWARE

This chapter presents the background and related work for the contributions of this thesis. It starts by defining some testing-related terminology, continues by briefly discussing test selection, the central issue in software testing, and then examines existing solutions to the issues involved in automating the testing process. It then presents the results of studies evaluating various testing strategies and proposed fault classification schemes. The chapter ends with a discussion of static approaches for finding software faults.

4.1 Terminology

In this section we introduce some terms used throughout this thesis. Unless indicated otherwise, the definitions are taken from Robert Binder's reference work "Testing Object-Oriented Systems. Models, Patterns, and Tools" [25].

Types of tests

Various properties of software can be checked through testing. Thus, depending on the goal, there can be several types of testing, among which:

- *Functional testing* — checks if a software element fulfills its specification
- *Robustness testing* — exercises a system in cases not covered by its specification

- *Performance testing* — estimates various performance parameters of the system

This thesis only addresses functional testing. For briefness and simplicity, we hence use the term "testing" to refer to functional testing.

Another important distinction is between *white-box* and *black-box testing*: white-box testing uses information about the internals of the system under test, black-box testing does not. Hence the former only uses the specification of the system and needs only an executable, whereas the latter performs some analysis of the source code to design the test cases. Black-box testing is typically performed at the system level, while white-box testing is typically performed at the unit level.

Mutation testing addresses a different goal: estimating the quality of the tests, not of the system under test. Mutation testing consists of purposely introducing faults in a program (usually by making rather simple syntactic changes to the source code) and then running a test suite on the modified versions of the program (called "mutants") to see if it detects the modifications. Scores reflect the proportions of detected mutants and are considered indicative of the fault-revealing capability of the test suite.

Fault-related terminology

A *failure* is a run-time event showing that the system's behavior does not correspond to its specification. In other words, a failure is an observed difference between actual and intended behaviors. A *fault* is a problem in the code (incorrect or missing code), which may, but does not have to, lead to a failure when the system is executed. One fault can trigger arbitrarily many failures. Mapping failures to faults is part of the debugging process and is usually done by humans. According to the IEEE Standard Glossary of Software Engineering Terminology [10], the human action that leads to a fault in the code is a *mistake*. Binder [25] uses the term "error" for this, but since this is a very general term with several possible meanings, we only use it accompanied by an explanation of its meaning in context. A *bug* is a mistake or a fault. Due to this imprecision of meaning, we do not use this term in this thesis.

Test scope

A *unit test* typically exercises a relatively small executable. Hence, for O-O systems, the scope of a unit test is typically one or several routines of a class or group of related classes. An *integration test* exercises several components of a software system, with the purpose of checking if by interacting they can achieve a certain task. Integration testing is typically performed after the components have been unit tested. A *system test* exercises a complete integrated application; its purpose is to check if the application as a whole meets its requirements.

Components of a test

A *test case* contains the code necessary to bring the implementation under test (IUT) to a certain state, to create the test inputs, to call the IUT with these inputs, and to check if the actual result corresponds to the expected one. This result does not include only returned values, but also the state of the IUT after the test case execution, and any messages displayed and exceptions thrown during the execution. A *test suite* is a set of test cases, typically related through their test scope or goal.

A *test driver* is a program or part thereof that triggers the execution of the test cases on the IUT. A *stub* is a partial implementation of a component used as a temporary replacement for it. A *test harness* is a set of tools, including test drivers, that supports test execution.

Evaluation of the outcome of a test case

The *oracle* produces the expected results and can make a *pass/fail* evaluation of the test case: if the actual results match the expected ones, then the test case passes, otherwise it fails. In the context of testing contracted software, a further case arises: if the test case does not fulfill the precondition of a routine it calls, we say that the test case is *invalid*.

Debugging

Debugging is an activity separate from testing. It follows after the existence of a fault has been established, either through a failure during testing or through analysis of the source code, and encompasses the actions necessary for removing the fault: if a failure occurred, its cause must be identified and located in the source code and the code must be changed so that the fault is removed. After this, typically tests are run again to check that the fault was indeed removed. Therefore debugging, not testing, is the activity that actually improves software quality, but testing is a key step towards this goal.

4.2 Test selection

Since for a realistic program no practical strategy can include all possible test cases, the central question of testing is how to select a set of test cases most likely to uncover faults. This problem comes down to grouping data into equivalence classes, which should have the property that, if one value in the set causes a failure, then all other values in the set will cause the same failure and conversely, if a value in the set does not cause a failure, then none of the others should cause a failure. This property allows using only one value from each equivalence class as a representative for its set and is the basis of *partition testing*. If it could be proved that the chosen partition of the input domain indeed results in equivalence classes, then exhaustive testing could be performed. Through their work on the category-partition method [123], Ostrand and Balcer laid the basis for *systematic* approaches to partition testing.

Several strategies are possible for coming up with partitions of the input domain, such as:

- Based on the requirements and knowledge of the code and of defect likelihood — manually determining partition classes based on the developer's knowledge of the program and intuition of which inputs might be incorrectly handled
- Based on the control flow — determining partitions so that the control flow graph is covered
- Based on the data flow — determining partitions so that the data flow graph is covered

A completely different strategy does not use partitions, but simply picks test cases at random; this strategy is called *random testing*.

4.3 Automated testing

Any software testing activity involves several steps:

1. Establishing the test scope — this can be as small as one or several routines of a class (in the case of unit testing) or as large as an entire software system consisting of millions of lines of code (in the case of system testing)
2. Creating and choosing inputs for the test — in the case of testing O-O software, these inputs must be objects and possibly primitive values

3. Causing the execution of the elements in the test scope using the selected inputs

4. Examining the output of this execution and possibly also intermediate states reached by the program to decide if the test passed or failed. This decision naturally requires access to the specification of the software.

5. If resources permit, going back to step 2 in order to execute more tests. The information obtained through the previously run tests can guide the process of designing new tests.

Each of these steps can involve specific approaches and can be automated or performed by developers or testers, resulting in a very wide variety of testing strategies, with different levels of automation. In the following we examine how test execution, input selection, and the oracle can be automated and what the challenges are.

4.3.1 *Automated execution*

The automation of test execution through the xUnit family of tools had a strong impact on the regular practice of software testing. xUnit is a generic name for any unit test framework that automates test execution. xUnit tools supply all the mechanisms needed to run tests so that the test writer only needs to provide the test-specific logic: setting up system state, creating test inputs, and the oracle, typically in the form of an assertion comparing the actual result to the expected one. xUnit tools are available for all major programming languages: JUnit [5] for Java, CppUnit [1] for C++, sUnit [7] for Smalltalk, PyUnit [6] for Python, vbUnit [9] for Visual Basic, getest (Gobo Eiffel Test) [3] for Eiffel, etc.

Such tools typically require testers to group their test cases as routines in classes inheriting from a library class and/or following a certain naming convention, so that the tool can automatically detect the manual test cases. The tools also allow defining set-up and tear-down routines, which are executed before and respectively after any test routine. Typically xUnit tools further provide various `assert` predicates, allowing to check equality and inequality of variables, equality to certain constants, etc. Some tools allow specifying a timeout for the tests, or specifying that the test is expected to throw a certain exception.

4.3.2 *Automated input generation*

A wide variety of strategies exists for automatically generating test inputs, some black-box and some white-box, some guided and some unguided, some using a formal specification to select inputs, others using only the implementation, some relying on static analysis techniques, others purely dynamic. In the following we briefly describe some of the existing strategies for automated test input generation, grouped by their main characteristics.

Random-based input generation

Random testing is a strategy which picks values at random from the input domain. It thus purposely rejects any system in choosing the inputs and hence ensures that there is no correlation between tests [75]. Random testing presents several benefits in automated testing processes: ease of implementation, efficiency of test case generation, and the existence of ways to estimate its reliability.

Hamlet [75] provides a comprehensive overview of random testing. He stresses the point that it is exactly the lack of system in choosing inputs that makes random testing the only strategy that can offer any statistical prediction of significance of the results. Hence, if such a measure of reliability is necessary, random testing is the only option. Furthermore, random testing is also the only option in cases when information is lacking to make systematic choices [74].

Although random testing had until recently been applied mostly to numeric test data, interest has grown in applying it to object-oriented software. Tools like JCrasher [47], Eclat [124], Jartege [122], RUTE-J [16], or Jtest [4] are evidence of this interest. All these tools employ random strategies for input generation for object-oriented systems; Jartege also allows users to define inputs.

JCrasher [47] was implemented for testing Java programs and builds test inputs by calling sequences of creation procedures and other routines. It does this by first building a graph of routines that can be called to create instances of the needed types and then exploring this graph to generate test inputs. For primitive types it chooses values randomly from predefined sets. JCrasher thus generates test cases in JUnit format, uses the framework for running them and reports as failed any test cases that threw an exception.

Eclat [124] also targets Java classes and uses a constructive approach for generating test inputs: it either calls a creation procedure — potentially followed by calls to other routines of the class that return `null` — or a routine that returns the necessary type. Once created, such an input is stored in a pool, together with the corresponding code snippet. Eclat initializes the pool with a few values for primitive types and `null` for reference types. Eclat uses an operational profile (inferred from successful executions) of the system under test as oracle for the tests: any test violating this operational profile is either illegal (if it violated the precondition of the routine under test) or fault-revealing (in all other cases). Objects participating in illegal and fault-revealing tests are not stored in the pool, so that they cannot be used for building other inputs.

RANDOOP [125] builds inputs incrementally, by randomly selecting a routine or creation procedure to call using previously-computed values as arguments. Once it has generated a value through such a sequence of routine calls, it only retains the value if no exception was thrown during its creation and if the value is not redundant. RANDOOP determines redundant values by using the Java `equals` routine: any two values for which this routine returns **`True`** are considered redundant and one of them gets discarded. Thus, RANDOOP's approach is not purely random.

Jartege [122] is similar to JCrasher and Eclat in that it also builds test inputs by combining creation procedure and regular routine calls. It also targets Java classes equipped with JML [96] contracts, which it uses as filters for invalid inputs and as automated oracle. Jartege gives users some level of control over the input generation process, by allowing weights to be associated with classes and routines and also by providing users the possibility of specifying the number of instances of a certain type that should be generated.

RUTE-J [16] (Randomized Unit Testing Engine for Java) also uses a constructive approach towards input generation, but leaves much of the test fine-tuning work (such as specifying ranges for numeric types and initializing reference objects) to users, who have to write so-called test fragments — Java classes tightly coupled with some library classes, which set up the test cases. Test fragments can have associated weights. RUTE-J provides a GUI that allows users to specify the number of test cases to be run and their length. RUTE-J classifies any thrown exception as a failed test case.

Jtest [4] is a commercial product developed by Parasoft Inc. and there is little available technical documentation on it. Jtest generates sequences of creation procedure and routine calls up to a depth of 3 to create inputs and uses runtime exceptions as indications of faults.

Yet other tools combine random testing with other strategies. For example, DART [65] (Directed Automated Random Testing) implements a symbolic execution-based approach, which uses random input generation to overcome the limitations of symbolic execution. DART combines concrete and symbolic execution: it starts by executing the routine under test with randomly generated inputs and, as execution proceeds, calculates an input vector for the next execution, which ensures that a different path in the code will be followed. DART builds this vector by solving symbolic constraints gathered during the execution from predicates in branch statements. DART overcomes the limitations of constraint solvers by simply replacing constraints that it cannot solve with their actual value observed during the previous execution, and then continuing both the concrete and the symbolic execution.

The research community has split views on random testing. Many authors of reference texts are critical towards it. Glenford J. Myers deems it the poorest testing methodology and "at best, an inefficient and ad hoc approach to testing" [114]. Nevertheless, random testing has proved effective at uncovering faults in many different applications, such as Unix utilities [111], Windows GUI applications [60], Haskell programs [46], and Java programs [47, 124]. Furthermore, several studies [53, 73] disproved this assessment by showing that random testing can be more cost-effective than partition testing. Andrews et al. [14] show that, when specific recommended practices are followed, a testing strategy based on random input generation finds faults even in mature software, and does so efficiently. They also state that, in addition to lack of proper tool support, the main reason for the rejection of random testing is lack of information about best practices.

Adaptive random testing

Some approaches start from the idea of random testing and try to improve its performance by adding some guidance to the algorithm. Such guidance can mean pruning out invalid and duplicate inputs as RANDOOP [125] does, combining random and systematic techniques as DART [65] does, or trying to spread out the selected values over the input domain, as is the case for Adaptive Random Testing [35] (ART) and quasi-random testing [37].

ART is based on the intuition that an even distribution of test cases in the input space allows finding faults through fewer test cases than with purely random testing. The implementation of ART requires keeping two disjoint sets of test cases: a *candidate* set and an *executed* set. The test cases in the candidate set are generated randomly. The executed set is initially empty; then, as testing progresses, test cases that are executed are added to it and removed from the candidate set. The first test case that gets executed is selected at random from the candidate set and is added to the executed set; in the subsequent steps, the test case from the candidate set that is furthest away from the executed ones is selected from the candidate set. The distance between two test cases is computed using the Euclidean measure: for an n-dimensional input domain, the distance between two test cases a and b whose inputs are a_i and b_i respectively, for $i \in \{1, ..., n\}$, is *dist(a,b)* = $\sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$. To evaluate the efficiency of ART, the authors use the F-measure: the number of test cases required to reveal the first fault. Their experimental results show that, in terms of this measure, ART can be more efficient than random testing by more than 50%.

Based on the ART intuition, a series of related algorithms have been proposed. Mirror ART [34] (MART) and ART through dynamic partitioning [38] reduce the overhead of ART. In MART, the input space is partitioned into disjoint subdomains. Test cases are generated in only one of these subdomains, using the ART algorithm, and then these generated test cases are mirrored into the other subdomains. This reduces the number of distance calculations that ART must perform. Restricted Random Testing [32] (RRT) is also closely related to ART and is based on restricting the regions of the input space where test cases can be generated. As opposed to ART, which generates the elements of the candidate set randomly, RRT always generates test cases so that they are outside of the exclusion zones (a candidate is randomly generated, and, if it is inside an exclusion zone, it is disregarded and a new random generation is attempted). Lattice-based ART [106] and ART by bisection with restriction [105] bring further performance improvements to ART.

All the above algorithms need a measure of the distance between two test cases, which is calculated based on the distances between their integer (or real) inputs. To extend the applicability of these distance-based testing algorithms to object-oriented systems, we introduced the notion of *object distance*, which allows calculating distances between complex data structures. Chapter 6 describes this notion and new testing algorithms based on it.

Coverage-oriented approaches

Many testing strategies aim at achieving high coverage levels, either related to data or to code, and the algorithms they implement are especially targeted at increasing coverage.

Korel [92] developed a dynamic test data generation method that targets path coverage. For every path in the program, it starts by executing the program with a particular input and checks for every reached branch-ing point if the currently used inputs cause the desired branch to be taken. If this is not the case, then a hill climbing algorithm is used to find inputs that would cause execution to follow the desired branch. This algorithm takes each input value in turn and increases and decreases its value by a small amount, trying to optimize a fitness function which measures how close the input is to triggering the execution of the desired branch. The program is executed for the two new inputs (the increased and the decreased version), and the new value that improves the fitness function is kept. When the value for one input variable cannot be optimized further, the algorithm continues by changing the values for the other input variables in the same way. This approach of optimizing one variable at a time is known as the "alternating variable method".

Ferguson and Korel [58] developed the chaining approach, which also targets path coverage, but uses information from the data flow graph of the program too, not only from the control flow graph, to determine the statements that could influence whether a certain instruction gets executed or not. The sequence of statements identified in this way must be executed in order for the target instruction to get executed.

Offutt et al. [119] developed the dynamic domain reduction (DDR) procedure, which determines sets of values having the property that all values in a certain set will cause the same program path to be taken. To achieve this, DDR executes a path in the program control flow graph between a start node and an end node, and at every decision node reduces the domains of the variables involved in the decision so that the condition is either true or false (depending on the branch it must take next) for all remaining values. When DDR reaches the end node in the control flow graph, the domains of all input variables will be reduced to sets having the desired property. Thus, the DDR approach is similar to Korel's [92] but DDR does not use any initial values and the result of running DDR are sets of values that have the property of triggering the same path in the program.

Gupta et al. [70] developed an iterative relaxation method for generating test inputs that achieve path coverage. This method starts by running the program with arbitrary input. Then it iteratively refines this input so that all the branch predicates on the given path evaluate to the desired outcome. This refinement is achieved by monitoring constraints that cause the path to be taken, constraints which provide information on the amounts by which the input should be adjusted.

In later work, Gupta et al. [68] developed an approach to test data generation aimed at achieving branch coverage. They start with an arbitrary input and then modify it to get new inputs so as to force execution through a path containing the desired branch. They employ a concept of "path resistance", measuring how difficult it is to force execution through a certain path.

Visser et al. [139] use the Java PathFinder model checker [138] to generate test data that achieves branch coverage of code involving complex data structures. In particular, in this work Visser et al. show how preconditions can be used on structures that are not fully initialized to prune the search for inputs that increase coverage.

Concolic testing (standing for the combination of concrete and symbolic execution) and its flagship implementations in the CUTE [132] and jCUTE [131] tools aim at achieving path coverage. They specifically address testing of code taking memory graphs as inputs.

Other approaches focus on data coverage. The Korat tool [27] can generate all non-isomorphic inputs up to a given bound by using the precondition of the routine under test and pruning the search space. Korat monitors attribute reads in the precondition to determine the attributes whose values can influence the evaluation of the precondition and only looks for alternative values for these attributes for inputs which violate the precondition. Because it sets attribute values directly (rather than building test inputs through sequences of creation procedure and routine calls), Korat must check that every generated input object fulfills its class invariant. If this is not the case, the input is discarded.

The Symstra tool [146] employs a different technique for achieving data coverage: given a set of methods from the class under test and a bound on the length of sequences of method calls, Symstra uses symbolic execution and comparisons of symbolic states to exhaustively explore the state space of the class under test.

DeMillo and Offutt [51] target a completely different type of coverage: mutation operator coverage. Their approach, implemented in the Godzilla tool, is designed so that it uncovers particular types of faults in programs, essentially represented through mutation operators. Test cases are generated by solving constraints which describe these types of faults.

Search-based test data generation

An increasing body of research applies search methods for generating test inputs, using the test goal as the fitness function. Miller and Spooner [112] were the first ones to apply a search technique to the results of program executions. In his above-cited work, Korel [92] extended their approach to a Pascal-like language and in later work he and Ferguson developed the chaining approach [58].

The disadvantage of such local search methods as those employed by Korel is that they may determine a local minimum instead of a global one. Other research [137] hence employs simulated annealing instead of hill climbing for similar purposes.

Evolutionary testing, the application of evolutionary algorithms to test data generation, has also seen rapidly growing interest in the last few years. Coverage-oriented approaches [87, 130, 110, 136] use measures of code coverage as fitness criteria. Jones et al. [87] target branch coverage, hence they use the branch distance as fitness function. They also handle loop testing, for which the fitness function is the difference between the actual and the desired number of iterations of the loop.

Other research directions investigate how to provide some guidance to the search. Harman et al. [77] introduced the idea of analyzing variable dependence relationships to determine the variables that cannot influence the evaluation of a branch condition.

Tonella [136] applies evolutionary testing to O-O systems and represents test cases as chromosomes, containing information on how to create input objects and how to change their state. Any primitive input values are also represented in the chromosome. Test cases are then mutated to produce new populations, whose fitness function can be any coverage criterion.

Baudry et al. [22] use the mutation score (the proportion of mutants killed by a test suite out of all mutants) as fitness function and use a modified version of the classical genetic algorithm: they introduce a memorization function and do not perform crossover.

Evolutionary testing has not only been applied for checking the functionality of software systems but also for other types of testing activities, such as stress testing of real-time systems [29], verifying timing constraints on real-time systems [141], etc.

Harman and McMinn [79] analyzed theoretically when evolutionary testing should be used and compared this testing technique to hill climbing and to random testing. They found that under some circumstances hill climbing can perform better than evolutionary testing, but there are also cases in which evolutionary testing can explore cases unreachable by random testing and by hill climbing.

Specification-based test generation

Specifications can be used for testing in several ways: as filter for invalid inputs, as guidance for test generation, as coverage criterion, as an automated oracle, etc. Many testing strategies rely on these properties. Here we list some approaches that use specifications for generating inputs, and in section 4.3.3 we discuss approaches that use executable specifications as oracles.

Richardson et al. [129] extended implementation-based testing techniques existent at the time to specification-based test selection. Dick and Faivre [52] developed automated techniques for deriving partitions from state-based specifications. Chang [33] et al. present techniques to derive test conditions from ADL specifications. Hierons [84] presents algorithms for rewriting Z specifications so that partitions of the input domain and the states for a finite-state automaton model can be derived from the new form. Such a model can then be used to control an automated testing process. Offutt et al. [120] define coverage criteria for tests generated from state-based specifications. In later work [121] they developed a tool called SpecTest for the automatic generation of test inputs from formal state-based specifications. Weyuker et al. [142] present a method for generating tests from boolean specifications of the software.

Combinations of static and dynamic approaches

Several testing strategies combine static and dynamic analysis. A rich body of research work uses symbolic execution to drive the testing process. The first such system was EFFIGY [90] developed by King for a PL/I-style language. EFFIGY integrated symbolic execution, a program verifier, a test manager component, and features for debugging. The already-cited Symstra and Java PathFinder also integrate symbolic execution and testing. Khurshid et al. [89] combine Korat and Java PathFinder in a testing process that can handle dynamically allocated structures such as lists and trees, method preconditions, and concurrency. Beyer et al. [24] extended the BLAST model checker to determine the set of all program locations where a predicate can be true and to generate test cases which cause it to be true at all such program locations. Tillmann and Schulte [135] introduced the notion of parameterized unit tests (PUTs), which are essentially unit tests taking inputs that can change their behavior. Tillmann and Schulte employ symbolic execution to generate inputs for PUTs.

The Check'n'Crash tool [48] derives abstract error conditions using the ESC/Java static checker, uses a constraint solver to derive concrete error conditions from the abstract ones, and generates concrete test cases with JCrasher that should trigger the errors. The DSD-Crasher tool [49] augments Check'n'Crash with a dynamic analysis to filter out illegal input parameters.

Testability transformations

Many of the mentioned approaches for automated test data generation require a specific structure of the program or are impeded by the presence of particular patterns in the code. Harman et al. [78] introduced a general notion of *testability transformation*, which is a source-to-source transformation of a program with the aim of allowing a test generation method to more easily create test input data for the program. The notion of *testability* of a program was introduced by Voas and Miller [140] as the likelihood that a fault is executed and produces a failure. Voas and Miller thus defined the PIE (Propagation, Infection, and Execution) framework for measuring testability. Harman et al. use the notion of "testability" of a program in a more restricted sense: the ease of automatically generating test input data for the program.

Testability transformations differ from traditional program transformation methods in that the former do not need to ensure functional equivalence; testability transformations only ensure that the test data generated for the transformed program is adequate for the original program and for the original testing criterion. Testability transformations are applicable to any automated approach for generating test data.

4.3.3 Automated oracles

Fully automated testing requires the existence of an automated oracle. The expected output can be provided at different levels of abstraction. In an extreme case, it can be as general as "no exception thrown", approach taken by tools such as JCrasher [47], RUTE-J [16], and Jtest [4].

The expected output can be made more specific by the existence of specification checkable at runtime. Some tools assume the presence of such specification embedded in the source code: Cheon and Leavens [39] propose an approach for testing Java classes equipped with JML contracts, approach in which users have to provide the test inputs; the Jartege tool [122] also assumes the presence of JML specifications in the Java classes under test, but automates input generation, as explained in section 4.3.2; Edwards [54] addresses component testing and advocates the use of abstract mathematical models in specifications rather than of the internal representation of the component under test.

Other testing tools do not assume that the code is equipped with executable specification and hence use assertion inference tools for producing such specification.

Automated assertion inference

Arguably the best-known tool for dynamic assertion inference is Daikon [55], which infers specifications from successful executions of a system by checking, at various program points, a set of conditions derived from templates on the variables in scope. Section 7.1 contains detailed information about this tool.

DIDUCE [76] is another tool which infers assertions from program executions. DIDUCE is built on the same principles as Daikon, but can operate in two modes: the training mode and the checking mode. In the training mode, the tool infers assertions from executions of the system, by starting out with the most restrictive conditions and relaxing them as if finds states that violate them. The checking mode is an extension of the training mode, in the sense that in the checking mode, when an assertion violation occurs, DIDUCE also reports the violation, in addition to relaxing the assertion in question. DIDUCE works for Java code, but there exists also an implementation of it for C programs called C-DIDUCE [57].

Pytlik et al. [128] developed the Carrot assertion detector which uses the same principles as Daikon, but has a different implementation. Further work [81, 82] investigates dynamic inference techniques for algebraic specifications.

Several tools aim at determining legal sequences of routine calls, rather than specifications. Ammons et al. [13] developed a machine learning approach: sequences of routine calls observed in system executions are fed to a machine learner, which generates a grammar of permitted call sequences. Whaley et al. [144] propose a system with the same goal of inferring constraints on sequences of legal routine calls; their system combines static and dynamic approaches to achieve this goal and represents the routine sequences through finite state machines.

Some of the ideas developed in academic research on assertion inference were also adopted by industry. AgitatorOne [26], previously called Agitator, developed at Agitar Software, implements a Daikon-like approach for inferring assertions. Users have the option of promoting these inferred assertions to contracts included in the program or discarding them. The Axiom Meister tool [134] developed at Microsoft Research uses symbolic execution for finding routine assertions for .NET programs.

Testing based on automated assertion inference

Eclat [124], described above, uses assertions inferred by Daikon as filters for invalid inputs and as automated oracles. Xie and Notkin [147] developed the operational violation approach, which uses Daikon to infer likely assertions and automatically generates tests, verifying the inferred assertions. Tests violating these assertions are presented to users for examination, since they exercise behavior that the tool has not seen before. The DSD-Crasher tool [49], mentioned above, employs Daikon for inferring assertions, exports the assertions that Daikon generates as JML contracts, and uses these to guide the input generation of the Check'n'Crash tool [48]. Substra [149] generates integration tests based on Daikon-inferred constraints on component interfaces.

Test extraction for reproducing failures

Other tools do not create test cases themselves, but try to *extract* test cases from failing runs of a system, where a failing run is defined as an execution of the system triggering an exception or contract violation. Extracted test cases are saved to disk, typically in xUnit format, and ideally can reproduce the observed failure. This is the idea behind the CDD tool developed by Leitner et al [100]. CDD was created starting from the observation that software developers typically prefer testing a system in an informal manner, usually by triggering its execution through a GUI and assessing the visual output that they get from it, rather than by writing tests. If a failure (exception or contract violation) occurs during this informal testing process, it is hard for the developers to reconstruct the system state that led to it so that they can identify the fault, and this is where CDD helps them.

A related tool developed by Artzi et al. [17] is ReCrash, which addresses the same problem of capturing states that lead to failures, but for a production setting, in which the software is already deployed and in use. ReCrash employs a concept called "second chance", due to which it can capture a failure-reproducing state the second time a particular failure occurs.

4.4 Evaluations of testing strategies

Given this very wide variety of tools and approaches to software testing, it is essential for any proposed tool to be evaluated throughly, so that the tools' applicability and strong and weak points are clear to potential users. Indeed, many such evaluations exist; in the following we concentrate on these evaluations relevant to the contributions of this thesis.

The evaluations of the random testing tools mentioned above (JCrasher [47], Eclat [124], Jtest [4], Jartege [122], RUTE-J [16]) are focused on various quality estimation methods for the tools themselves: finding real errors in existing software (JCrasher, Eclat, RUTE-J), in code created by the authors (Jartege), in code written by students (JCrasher), the number of false positives reported (JCrasher), mutation testing (RUTE-J), code coverage (RUTE-J). As such, the studies of the behaviors of these tools stand witness for the ability of random testing to find defects in mature and widely used software and to detect up to 100% of generated mutants for a class. These studies do not, however, employ any statistical analysis which would allow drawing more general conclusions from them about the nature of random testing.

The evaluations of tools that combine random testing with systematic approaches (RANDOOP [125], DART [65], ART [35], Agitator [26]) use purely random testing as a basis for comparison: RANDOOP and DART are shown to uncover defects that random testing does not find, DART achieves higher code coverage than random testing, ART finds defects with up to 50% less tests than random testing. These results, although highly interesting in terms of comparing *different* testing strategies, do not provide much information about the performance of random testing itself or about the predictability of its performance.

Mankefors et al. [104] also investigate random testing and introduce a new method for estimating the quality of random tests. As opposed to the study presented in section 8.1, their focus is on random testing of numerical routines and on quality estimations for tests that do not reveal bugs.

Andrews et al. [14] state that the main reasons behind the so far poor adoption of random generation for object-oriented unit tests is the lack of tools and of a set of recognized best practices. The authors provide such a set of best practices and also compare the performance of random testing to that of model checking. They show that random testing produces good results both when used on its own and when used as preparation for model checking.

There are several studies which empirically compare the performance of various testing strategies against that of random testing. Some such studies [53, 73, 143, 36] compare random testing and partition testing. Their results are centered around the conditions under which partition testing (with its several flavors such as data-flow-oriented testing, path testing, etc.) can perform better than random testing. Their empirical investigations (or, in the case of Hamlet and Taylor [73], theoretical studies) also show that, outside of these restraining conditions, random testing outperforms partition testing. Gutjahr [71] shows that, for equal expected failure rates of all blocks in a partition, random testing is outperformed by partition testing. However, *not knowing* expected failure rates does not necessarily mean they *are* indeed equal, and we hence consider this assumption to be very strong.

Pretschner et al. [127] found that random tests perform worse than both model-based and manually derived tests. D'Amorim et al [50] compare the performance of two input generation strategies (random generation and symbolic execution) combined with two strategies for test result classification (the use of operational models and of uncaught exceptions). The results of the study show much lower applicability of the symbolic-execution-based strategy than of the random one: the authors could only run the symbolic-execution-based tool on about 10% of the subjects used for the random strategy and, even for these 10% of subjects, the tool could only partly explore the code. Although, as the study shows, the symbolic-execution-based strategy does find faults that the random one does not, the tool has extremely restricted practical applicability.

Numerous other studies have compared structural and functional testing strategies as well as code reading (among others, [63, 21, 88, 145, 61]). Most of them have used small programs with seeded faults and compared results of two or three strategies. The five cited studies compare the automated selection of test cases using the control flow or the all-uses – respectively mutation or the all-uses – criteria and their outcome in terms of faults uncovered by each strategy. In the case of functional vs control flow vs code reading, human testers applied successively these 3 strategies to several programs. In each case they wrote the test cases if needed (for control flow and functional testing) following the given approach. None of these studies compares manual testing to automated techniques.

A different and unfortunately not thoroughly explored avenue of work investigates the ways in which testing strategies are evaluated. Since the purpose of testing is to find faults, all other measures of its effectiveness and efficiency (such as code/data coverage, mutation testing, etc.) should directly relate to its fault-revealing capability. Surprisingly, only very few researchers have so far investigated these relationships.

Offutt [118] showed empirically that detection of simple faults (such as those introduced by mutation) correlates with detection of complex faults, i.e. combinations of several simple faults, finding which validates the premise of mutation testing. Andrews et al. [15] performed an empirical study of how mutation testing and some measures of code (block and decision) and data (C-use and P-use) coverage relate to real fault finding. They found that the mutation score is a good predictor for the detection effectiveness of real faults and that achieving coverage levels close to 100% is effective in terms of fault detection. According to their results, the higher part of the coverage range brings a significant increase in fault detection. Furthermore, for comparable test suite sizes, all the mentioned coverage criteria detected a similar percentage of faults. In short, in their empirical study, code and data coverage proved useful in increasing test suite effectiveness. Previous empirical studies [86, 61] also reported similar results.

4.5 Fault classifications

Although this is naturally not their only applicability, fault classifications are also essential to evaluations of testing strategies, because they allow these strategies to be compared not only in terms of the *number* of faults they uncover, but also in terms of the *nature* of these faults.

Knuth [91] pioneered the work on classification of defects by defining 9 categories reflecting the faults that occurred most often during the development of TeX. Many fault classification models have been proposed since then [66, 23, 40, 103, 11]. This includes the Orthogonal Defect Classification (ODC) [40], which combines defect types and defect triggers. In a sense our classification presented in section 8.4.1 is an ODC in itself, but our classification of defect types is finer while the defect location is simpler than defect triggers.

The IEEE classification [11] aims at building a framework for a complete characterization of the defect. It defines 79 hierarchically organized types that are sufficient to understand any defect, but do not address the particular constructs of contract-enabled languages. Lutz [103] describes a safety checklist that defines categories of possible errors in safety-critical embedded systems. The classification probably most similar to ours is the one used by Basili et al. [21], organized in two dimensions: whether the fault is an omission or a commission fault, and to which of 6 possible types it belongs. Our classification takes into account specifications (contracts) and is more fine-grained.

Bug patterns (e.g., [12, 85, 56]) are also related to our fault classification. Allen [12] defined 14 types of defects in Java programs, and hence did not consider contracts and multiple inheritance. The FindBugs approach [85, 18] relies on the definition of bug patterns that are syntactically automatically recognizable. This results in a classification that comprises hundreds of fault types that are not grouped into coarser categories.

4.6 Static approaches for automated fault detection

Testing, the execution of a system in order to find faults in it, is not the only way in which program faults can be identified. Especially in recent years, the research community has developed a variety of static methods for fault detection.

Static analysis tools such as PREfix [30], the Extended Static Checker for Java [59] (ESC/Java), or SLAM [19] typically use a combination of predicate abstraction, model checking, symbolic reasoning, and iterative refinement to uncover faults in programs.

A different category of tools are the so-called "bug pattern detectors". These are tools that look for predefined templates in the code, templates considered indicative of the presence of faults. Arguably the best known such tool is FindBugs [85], which currently checks more than 300 bug patterns, classified in several categories and with various degrees of seriousness: correctness, multithreaded correctness, security, malicious code vulnerability, performance, bad practice, and dodgy. Users can also add their own patterns.

Hallem et al. [72] also propose a method for static detection of faults, based on user-specified analyses. Xie and Engler [148] statically look for redundancies in source code, considered indicative of the presence of faults. They show that the presence of harmless redundancies indeed correlates with that of serious faults.

The major weakness of bug pattern matchers consists in the spurious warnings (also called false positives) that they generate; FindBugs, for instance, reportedly generates around 50% false positives. Hence, significant effort has been put into reducing the number of such false positives [94, 93].

CHAPTER 5

AUTOTEST: A FRAMEWORK FOR CONTRACT-BASED TESTING

AutoTest, the tool used as vehicle for implementing the testing strategies described in the following chapters and for performing evaluations of their performance, was developed together with Andreas Leitner, who had a decisive contribution in creating and shaping the tool; this chapter hence describes joint work. AutoTest is available in open source [97].

This chapter starts with an overview of the tool and then describes in detail each of the steps involved in its automated testing strategy: finding the test scope, creating inputs, executing generated tests, using contracts as automated oracle, and minimizing failure-reproducing test cases. The chapter ends with a discussion of the integration of manual and automated testing in AutoTest.

5.1 Overview

AutoTest is a fully functional tool, but has an extensible architecture, so that other testing strategies than the basic one, described in this chapter, can be seamlessly plugged in.

AutoTest implements a fully automated testing process, which allows it to be a truly push-button testing tool. Through the tool's command line interface, a user can specify the classes to be tested and the time interval for which AutoTest should test them, as in the following example:

```
   auto_test --time-out=15 banking_system.ace --class
BANK_ACCOUNT
```

AutoTest is thus instructed to test class *BANK_ACCOUNT* for 15 minutes. `banking_system.ace` is a configuration file describing the project to which the class under test belongs, much like a makefile in C[1]. Hence, in the given 15 minutes, AutoTest:

- Generates a set of wrapper classes (necessary as support for reflection as explained in section 5.4) for the classes under test and all classes that they directly or indirectly depend on
- Generates an interpreter using these wrapper classes and compiles it
- Identifies and runs any existing manual tests which apply to the classes under test
- Starts the actual automated testing process consisting of calling all exported routines of the classes under test, with inputs generated randomly, and checking that contracts are fulfilled during these calls. Any contract violation or other thrown exception signals a fault in the classes under test.

When the 15 minutes have elapsed, AutoTest stops testing the given class and minimizes all failure-reproducing test cases that it generated, to support programmers in the debugging process and to reduce the size of the test cases it stores for regression testing. The rest of this chapter explains each of these steps in detail.

Figure 5.1 presents a high-level overview of the architecture of AutoTest. As shown in this figure, AutoTest takes as input the system under test (actually, as explained above, a make-like configuration file describing it) and the test scope within this system (one or several classes). AutoTest uses a two-process model to generate and run tests, as explained in detail in section 5.4: a master process implements the test generation strategy and uses a proxy component to communicate with the other process — an interpreter responsible for executing the tests. The proxy receives responses from the interpreter showing the status and outcome of the execution. The proxy parses these responses and passes them to the oracle component of AutoTest, which decides on the outcome (pass/fail) of running the test case and generates test result files, the output of AutoTest. Both the component implementing the test generation strategy and the one implementing the oracle can easily be replaced, allowing the seamless integration of different testing strategies in the tool.

AutoTest targets Eiffel code but can be applied with minor modifications to any other O-O language supporting static typing and dynamic binding, and having a common root for the class inheritance hierarchy. In the absence of embedded executable specification, AutoTest can still use thrown exceptions as indications of failures, but this is naturally a less precise automated oracle.

[1]LACE is the language in which such files were written for versions of EiffelStudio up to and including 5.6. Version 5.7 used a an XML-based format called "acex" and later versions use the "ecf" format, which is also XML-based. Therefore, the extension associated with such files differs between different versions of EiffelStudio.

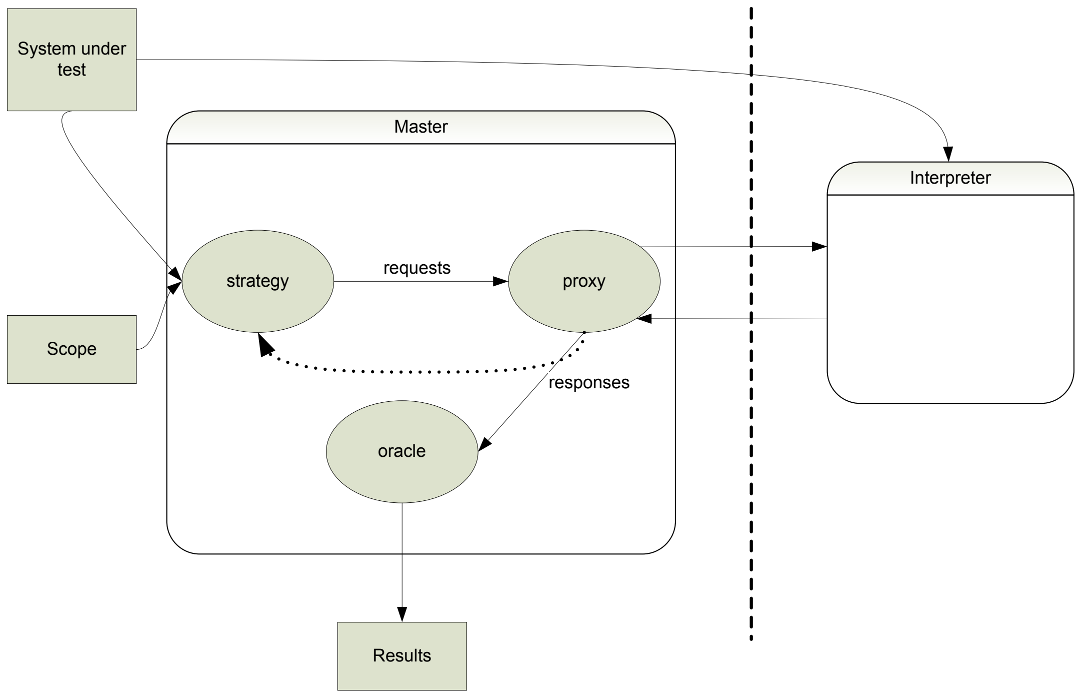

Figure 5.1: Overview of the architecture of AutoTest

5.2 Test scope

AutoTest was designed as a unit testing tool and, as such, its test scope consists of one or several classes, or more precisely of these classes' routines exported to *ANY*[2].

AutoTest tries to test all the routines in the test scope in a fair manner. To achieve this, it uses a system of priorities as follows. Each routine has a dynamic and a static priority. The static priorities are set at the start of the testing session and do not change anymore afterwards. These static priorities reflect how intensively each routine should be tested. For instance, setting the static priorities of routines inherited from *ANY* to 0 and the static priorities of all other routines to the same strictly positive integer would insure that routines inherited from *ANY* are not tested at all and all other routines are tested in a fair manner. Assigning to some routines higher static priorities than to others ensures that the former routines are tested proportionally more intensively than the latter.

Initially AutoTest sets all routines' dynamic priorities to their static priorities. At every step when it needs to select the next routine to test, AutoTest selects the one with the highest dynamic priority. Whenever it calls one of the routines in a test case, it decreases its dynamic priority by 1. When the dynamic priorities of all routines under test are equal to 0, AutoTest resets them all to the static priorities. AutoTest thus ensures that the routines under test are exercised in a fair manner in the time given for the testing session.

5.3 Contracts as oracles

As discussed in chapter 4, when specification is embedded in the software itself and is executable, it can be used as an automated oracle for testing. AutoTest thus exploits the presence of contracts in software written in Eiffel to gain a fully automatic and freely available oracle. But contracts play a double role in software testing: the *preconditions of the routines under test* specify constraints on the range of acceptable inputs; *all other contracts* specify conditions that must be fulfilled at various points in the execution. Thus, if AutoTest directly violates the precondition of a routine under test, this just means that the test engine has generated an *invalid test case* and the inputs are discarded, without executing the routine under test. All other cases of contract violations signal faults in the software under test, hence AutoTest has produced a *failing test case*.

When contracts are used as oracle, a failing test case can signal a fault either in the contract or in the implementation. As discussed in more detail in section 8.4.1, both cases are possible and do occur in practice, and a fault should be reported in both situations. It may not be obvious at first glance why reporting

[2]Since *ANY* is the root of the class inheritance hierarchy in Eiffel and all classes inherit by default from it, a routine exported to *ANY* can be called from any class.

faults in contracts is interesting at all, since in most cases contract checking is disabled in released software. The reason is that most often a developer writes both the contract and the body of a routine. A fault in the contract signals a mistake in the developer's thinking just as a fault in the routine body does. Once the routine has been implemented, client programmers who want to use its functionality from other classes look at its contract to understand under what conditions the routine can be called (conditions expressed by its precondition) and what the routine does (the postcondition expresses the effect of calling the routine on the state). Hence, if the routine's contract is incorrect, the routine will most likely be used incorrectly by its callers, which will produce a chain of faulty routines. The validity of the contract is thus as important as the correctness of the implementation.

Because in Eiffel it is allowed to call functions from contracts, evaluating a contract at runtime can have side effects. Hence, when AutoTest monitors contracts during test execution, it may actually be influencing the run of the system under test. This is not a flaw in the testing strategy itself; it is the programmer's responsibility to ensure that there are no side effects in contracts.

AutoTest also uses exceptions as part of the automated oracle, so any test case that raises an exception is classified as failing. This is possible due to Eiffel's special treatment of exceptions: unlike other languages, which use exceptions for control flow, in Eiffel exceptions are evidence of anomalous program states that should not occur in the execution of a system. In other words, in Eiffel a correct program should never throw an exception. AutoTest uses this property to filter fault-revealing test cases. Technically, contract violations also trigger exceptions in Eiffel, but, for increased precision, we refer to contracts and exceptions as separate components of the automated oracle.

5.4 Test execution

AutoTest uses a two-process model for executing the tests: the *master process* implements the actual testing strategy; the *slave process* is responsible for the test execution. The slave, an interpreter, gets simple commands (such as object creation, routine call, etc.) from the master and can only execute such instructions and return the results. This separation of the testing activity in two processes has the advantage of robustness: if test execution triggers a failure in the slave from which the process cannot recover, the driver will shut it down and then restart it where testing was interrupted. The entire testing process does not have to restart from the beginning and, if the same failure keeps occurring, the driver can decide to abort testing that routine so the rest of the test scope can still be explored.

The driver and the interpreter communicate through standard I/O: the driver outputs commands to the interpreter as strings, which the latter parses and executes. These commands are of two types:

1. Instructions that use a simplified and dynamically typed version of Eiffel. Once the interpreter has executed such an instruction, it will output a status message indicating if the execution was carried out successfully. Any output produced during execution, including stack traces for thrown exceptions, is also recorded. These instructions can have one of the following forms:

   - `create <TYPE> <Variable>[.<Creation_procedure_name> [<Arguments>]]` — creates an instance of `TYPE` associated to `Variable`
   - `<Variable> := Constant|<Variable>.<Query_name> [<Arguments>]` — assigns to a variable a constant or the return value of a query
   - `<Variable>.<Feature_name> [<Arguments>]` — calls a feature on the object associated to a variable

2. Meta-commands, unrelated to the actual test execution, but necessary for the driver to keep track of the testing activity:

   - `:type <Variable>` — instructs the interpreter to output the name of the type of the object that a certain variable denotes
   - `:quit` — stops the interpreter

Both the commands that the driver sends to the interpreter and the latter's responses are recorded in a log file. The interpreter's responses are recorded as comments, because this way the entire file can be re-fed to the interpreter as it is, and hence the testing session can be replayed. Figure 5.1 shows a fragment from such a log file.

```
v_1 := Void
  -- > done:
:type v_1
  -- > NONE
  -- > done:
v_2 := 9
  -- > done:
:type v_2
  -- > INTEGER
  -- > done:
create {FIXED_LIST [ANY]} v_3.make (v_2)
  -- > ---multi-line-value-start---
  -- > ---multi-line-value-end---
  -- > status: success
  -- > done:
-- Test case number: 1
create {CURSOR} v_4
  -- > ---multi-line-value-start---
  -- > ---multi-line-value-end---
  -- > status: success
  -- > done:
v_5 := 1
  -- > done:
:type v_5
  -- > INTEGER
  -- > done:
create {FIXED_LIST [ANY]} v_6.make_filled (v_2)
  -- > ---multi-line-value-start---
  -- > ---multi-line-value-end---
  -- > status: success
  -- > done:
-- Test case number: 2
v_6.copy (v_3)
  -- > ---multi-line-value-start---
  -- > ---multi-line-value-end---
  -- > status: success
  -- > done:
-- Test case number: 3
v_7 := v_6.is_inserted (v_3)
  -- > ---multi-line-value-start---
  -- > ---multi-line-value-end---
  -- > status: success
  -- > done:
-- Test case number: 4
```

Listing 5.1: Fragment from a log file recording the interactions between the driver and the interpreter.

AutoTest needs reflection support in order for the interpreter to be able to execute feature calls that it receives as strings from the driver. This particular ability to call routines via introspection was not present in Eiffel's reflection support, so it had to be implemented for AutoTest. The resulting tool was developed by Andreas Leitner and is called ERL-G (Eiffel Reflection Library Generator) [101]. ERL-G first parses the classes under test and determines the transitive closure of the classes that they depend on, thus determining the set of alive classes in the test session. For each class in this set, it generates a meta-class, which the interpreter uses to exercise the routines under test.

This separation into 2 processes also has the consequence that the objects used in tests only exist on the interpreter side. The driver only knows the variable identifiers of the objects in the pool and their types. (As explained in the next section, AutoTest keeps a pool of objects from which it selects inputs to use in tests.) Hence, when the interpreter dies, the object pool is emptied and must be rebuilt anew. Thus, interpreter restarts put a limit to the complexity of the object structures that can be reached in the testing session.

5.5 The random strategy for generating inputs

Random input generation for object-oriented applications can be performed in one of two ways:

- In a *constructive* manner, by calling a creation procedure of the targeted class and then, optionally, other routines of the class in order to change the state of the newly created object.
- In a *brute force* manner, by allocating a new object and then setting its fields directly (technique applied by the Korat tool [27], for instance).

The second approach has the disadvantage that objects created in this way can violate their class invariant. If this is the case, the effort for creating the object has been a waste and a new attempt must be made. (Invariant violations can also be triggered through the constructive approach, but, if that is the case, a fault has been found in the system under test.) Also, it can be the case that objects created in this way could never be created by executions of the software itself. For these reasons, AutoTest implements the first approach.

Any such constructive approach to generating objects (test inputs) must answer the following questions:

- Are objects built simply by creation procedure calls or are other routines of the class called after the creation procedure, in an attempt to bring the object to a more interesting state?
- How often are such diversification operations performed?
- Can objects used in tests be kept and re-used in later tests?
- How are values for basic types generated?

With AutoTest, answers to these questions take the form of explicit parameters provided to the input generation algorithm.

AutoTest keeps a pool of objects available for testing. All objects created as test inputs are stored in this pool, then returned to the pool once they have been used in tests. The algorithm for input generation proceeds in the following manner, given a routine *r* of a class *C* currently under test. To test *r*, a target object and arguments (if *r* takes any) are needed. With some probability P_{GenNew}, the algorithm either creates new instances for the target object and arguments or uses existing instances, which it takes from the pool. If the decision is to create new instances, then AutoTest calls a randomly chosen creation procedure of the corresponding class (or, if the class is deferred, of its closest non-deferred descendant). If this creation procedure takes arguments, the same algorithm is applied recursively. The input generation algorithm treats basic types differently: for an argument declared of a basic type, with some probability $P_{GenBasicRand}$, a value will be chosen randomly either out of the set of all values possible for that type or out of a set of predefined, special values. These predefined values are assumed to have a high fault-revealing rate when used as inputs. For instance, for type *INTEGER*, these values include the minimum and maximum possible values, 0, 1, -1, etc. Figure 5.2 depicts an overview of the use of these probabilities in the input generation algorithm.

Keeping a pool of objects which can be reused in tests raises the question of whether an attempt should be made to bring these existing objects to more interesting states as would occur during the actual execution of a program (for example with a list class, not just a state occurring after creation but one resulting from many insertions and deletions). To provide for this, we introduce the probability P_{Div} which indicates how often, after running a test case, a *diversification* operation is performed. Such a diversification operation consists of calling a procedure (a routine with no return value, which most likely changes the state of the object on which it is called) on an object selected randomly from the pool. Figure 5.3 shows where diversification occurs in the testing process.

Supplying different values for these three probabilities (P_{GenNew}, $P_{GenBasicRand}$, P_{Div}) changes the behavior of the input generation algorithm.

Other tools that use a constructive approach to random input generation also rely on calling sequences of creation procedures and other routines to create input data. Eclat [124], very much like AutoTest, stores objects in a pool and uses existing values to call creation procedures and routines which create new values. After this initial generation phase, it applies heuristics to classify and select which of the available inputs it will use in tests. AutoTest performs no such selection, because it implements a purely random strategy. JCrasher [47] builds sequences of creation procedure and routine calls starting from a routine under test and systematically building required input objects, by calling either creation procedures or queries returning objects of the desired type. Section 4.3.2 presents more examples of tools implementing constructive approaches to random input generation.

Example

The following describes how AutoTest's generation algorithm produces a test case via an example, which assumes that the class `BANK_ACCOUNT` shown in listing 3.1 is being tested. Figure 5.4 shows an extract from the generated tests. In this example, variables `v1`, `v2`, `v3`, and `v4` represent the object pool.

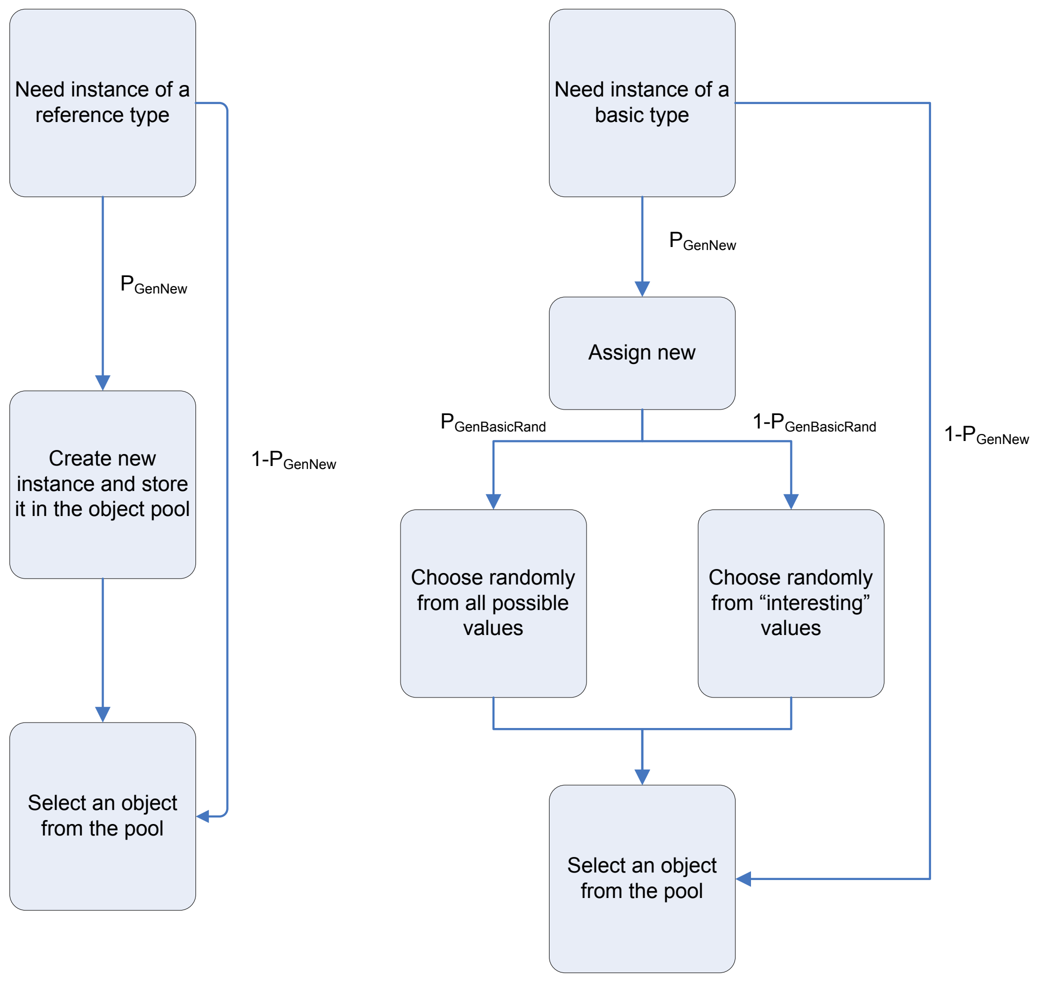
Need instance of a reference type
P_{GenNew}
1-P_{GenNew}
Create new instance and store it in the object pool
Select an object from the pool
Need instance of a basic type
P_{GenNew}
Assign new
$P_{GenBasicRand}$
1-$P_{GenBasicRand}$
1-P_{GenNew}
Choose randomly from all possible values
Choose randomly from “interesting” values
Select an object from the pool

Figure 5.2: Input generation. AutoTest proceeds differently if it needs to create an instance of a reference type or of a basic type.

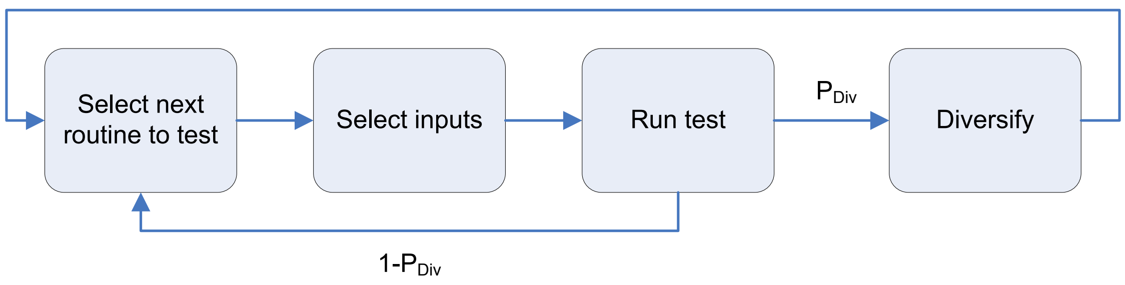

Figure 5.3: A high-level view of a step in the testing process. Diversification can be performed after every call to a routine under test.

```
create {STRING} v1.make_empty
create {BANK_ACCOUNT} v2.make (v1)
v3 := 452719
v2.deposit (v3)
v4 := Void
v2.transfer (v3, v4)
...
```

Figure 5.4: Example test case generated by AutoTest. Routines `deposit` and `transfer` of class *BANK_ACCOUNT* are being tested.

The class under test is *BANK_ACCOUNT*, so all its routines must be tested. At every step, AutoTest chooses one of the routines which have been tested the least up to that point. When the testing session starts, no routines have been tested, so one of the routines of class *BANK_ACCOUNT* is chosen at random. Assume routine `deposit` is chosen. In order to execute this routine, AutoTest needs an object of type *BANK_ACCOUNT* and an integer representing the amount of money to deposit.

The test generator randomly chooses inputs with the required types from the object pool. However, before it makes this choice, with probability P_{GenNew} it can also create new instances for the required types and add them to the pool. In the example test case, the generator decides at this point to create a new object of type *BANK_ACCOUNT*. Therefore it chooses a creation procedure (`make`) and now works on the sub-task of acquiring objects serving as parameters for this creation procedure. `make` requires only one argument which is of type *STRING*, so a string object must be created. The type of string objects in Eiffel is a regular reference type. The algorithm decides again to create a new object, and uses the creation procedure `make_empty` which does not take any arguments (line 1). After calling this creation procedure, the object pool is:

```
v1: STRING
```

AutoTest now synthesizes the creation instruction for the bank account object (line 2) using the newly created string object. This updates the object pool to:

```
v1: STRING
v2: BANK_ACCOUNT
```

Now AutoTest needs an integer as argument to `deposit`. Integers are basic objects and in the example, a random integer (452719) is chosen and assigned to a fresh pool variable (line 3). This changes the object pool to:

```
v1: STRING
v2: BANK_ACCOUNT
v3: INTEGER
```

AutoTest can now synthesize the call to *BANK_ACCOUNT*.`deposit`, using the newly created bank account and the randomly chosen integer (line 4).

At this point, AutoTest selects another routine for testing. Assume *BANK_ACCOUNT*.`transfer` is chosen. This routine transfers an amount from the bank account object on which it is called to the bank account that is provided as argument. One target object and two arguments are necessary. For each of these three inputs, an object of the corresponding type might be created and added to the pool. In the case of the call to *BANK_ACCOUNT*.`transfer`, the decision is to create a new instance of *BANK_ACCOUNT*. Whenever AutoTest has to create a new object, it may also choose to add **`Void`** to the pool instead of a new object.

This happens in the example, so the pool now consists of:

```
v1: STRING
v2: BANK_ACCOUNT
v3: INTEGER
v4: NONE
```

Now, two instances of *BANK_ACCOUNT* and an integer are chosen randomly from the pool and the result is the call to `transfer` as shown on line 6. (Since class *NONE*, the type of **`Void`**, conforms to all other types, **`Void`** can be used whenever an instance of a reference type is required.) Test case generation continues after this point, but for brevity the example stops here.

5.6 Test case minimization

AutoTest tries to minimize all failure-triggering test cases that it generates. Minimization is particularly important for randomly generated test cases exactly because of their lack of system, which leads to many parts of the test case not being necessary for reproducing the failure. Eliminating such parts is essential:

- As support for the developers who use the test case to debug the software, since this way it is easier for them to locate the fault in the code

- As support for regression testing: the reduced size of the test case means less necessary storage space and shorter execution time.

AutoTest employs a static slicing technique for minimizing test cases, as described in [98]. This technique does not guarantee the absolute minimal version of the test case, but it does guarantee to produce a version of the test case that still triggers the failure.

The technique proceeds as follows. It maintains a set of variables that are necessary for reproducing the failure. It initializes this set with the variables involved in the routine call that triggered the failure and examines each of the preceding routine and creation procedure calls to determine if they could have an effect on the variables. An instruction is considered to have an effect on a variable if it uses the variable as target for a routine or creation procedure call or if it assigns to it. When such instructions are found, the other variables used in it are also added to the set. If a variable is assigned to or used as target for a creation procedure call, then it is eliminated from the set of variables, since such instructions assign it new values and are completely independent from any values that the variable had previously.

The minimizer proceeds this way, examining each instruction from the test session, until it reaches the start of the testing session. The minimizer then checks that the thus determined set of instructions indeed reproduces the same failure as the original test case by simply running the minimized version. If the failure is not triggered again, then minimization has failed and AutoTest retains the original test case. Otherwise minimization was successful, so AutoTest stores the minimized test case and displays it at the end of the testing session together with the other results.

Experiments [98] have shown that this minimization technique reduces the size of randomly generated test cases on average by a factor of 96% and is around 50 times faster than delta debugging [150].

In AutoTest minimization takes place after testing is completed, hence after the timeout given by the user has occurred, so minimization takes extra time. It is also possible to disable minimization through a command line argument.

5.7 Combining manual and automated testing

AutoTest seamlessly integrates manual tests with automated ones, as described in [99]. Classes representing manual tests must inherit from a library class called *AUT_TEST_CASE*. This is a deferred class containing only two empty routines, *set_up* and *tear_down*, which descendant classes can redefine to contain actions to be performed before and after executing every test case. Test cases are themselves routines of the classes inheriting from *AUT_TEST_CASE* with names starting with "test". This practice of encapsulating manual test cases in classes inheriting from a certain library class is common with frameworks automating test execution, such as the xUnit family of tools mentioned in section 4.3.1.

By checking this inheritance relation AutoTest can determine the classes containing manual test cases. To determine which classes they test, AutoTest uses the transitive closure of the client relation between classes: it considers that those classes are tested, whose functionality the manual test classes use, directly or indirectly.

Figure 5.5 shows an example illustrating these ideas. Suppose AutoTest is instructed to test class *STUDENT*. Because class *TEST_UNIVERSITY* inherits from class *AUT_TEST_CASE*, AutoTest identifies it as containing manual tests. Now AutoTest must determine if *TEST_UNIVERSITY* can exercise the functionality of *STUDENT*, the class under test. This is the case, since *TEST_UNIVERSITY* is a client of *UNIVERSITY*, which is a client of *STUDENT*. Hence AutoTest executes *TEST_UNIVERSITY*'s routines whose names start with "test" as part of the manual testing phase for class *STUDENT*.

AutoTest runs manual tests before it starts the automated test generation. Running manual tests can be disabled through a command-line option.

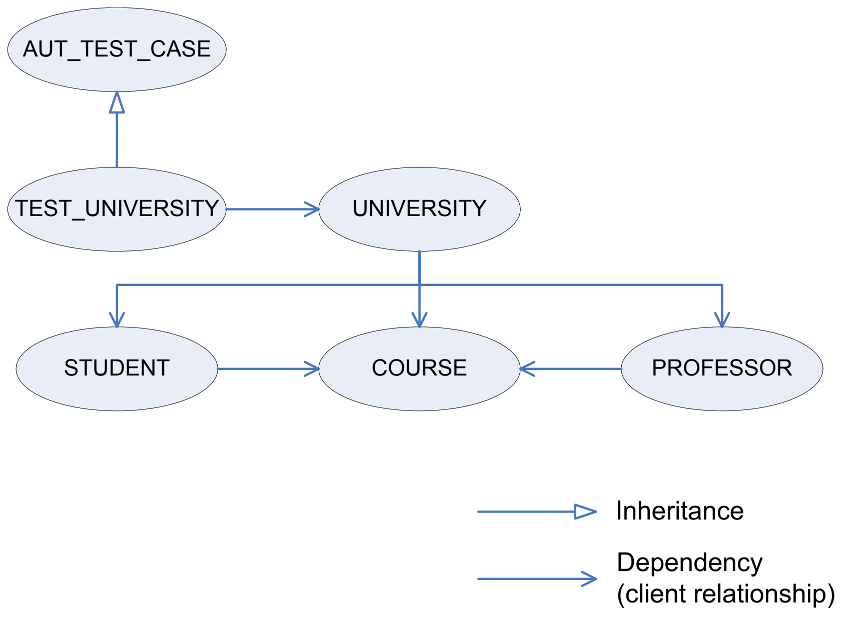

Figure 5.5: Example illustrating manual test case selection in AutoTest. AutoTest uses the inheritance relation to identify classes containing manual test cases and the client relation to identify the tested classes. Hence, in this example AutoTest determines that *TEST_UNIVERSITY* contains manual test cases and can exercise the functionality of class *STUDENT*, for instance (through class *UNIVERSITY*, which is in turn a client of class *STUDENT*).

CHAPTER 6

EXTENDING AUTOTEST WITH OTHER STRATEGIES FOR INPUT GENERATION

As explained in chapter 5, AutoTest is designed as a framework in which various strategies for generating inputs can be plugged. The input generators can thus easily be changed, while the rest of the support for the testing process stays the same: a driver process orchestrates the testing activity, while another process, an interpreter, carries out the actual test execution (5.4); contracts and exceptions provide for an automated oracle (5.3); fault-reproducing test cases are minimized when testing is finished (5.6). This design also has the advantage of making thorough comparisons between input generation strategies possible.

This chapter presents another strategy for generating inputs which we developed and implemented in AutoTest, other than the random one presented in section 5.5. It then describes how this strategy, the random one, and manual testing can all be combined to deliver the best results.

6.1 The object distance

Adaptive Random Testing (ART) [35], as explained in section 4.3.2, is based on the intuition that an even distribution of test cases in the input space allows finding faults through fewer test cases than with purely random testing. ART generates candidate inputs randomly, and at every step selects from them the one that is furthest away from the already used inputs. Work on ART has so far only considered inputs of numeric types, for which the notion of “even spacedness” immediately makes sense: any such input belongs to a known interval on which there exists a total order relation.

The ideas behind ART are attractive for testing O-O applications too. The challenge is to define what it means to "spread out" instances of arbitrarily complex types as ART does for numeric values. We have developed a method for automatically calculating an "object distance" [41] and applying it to adaptive random testing of object-oriented applications [43]. This section presents the object distance and the next section its application to adaptive random testing of O-O software.

There are many ways of defining a notion of distance between two objects. It is important to specify a framework for acceptable definitions, then make explicit any choices behind a specific proposal within that framework, and justify them. This discussion starts with a very general framework and makes a number of such choices until it arrives at a directly implementable notion, with an associated algorithm.

In general objects are characterized by:

- Their direct values
- Their dynamic types
- Their fields[1]

Hence, the object distance must take into account these three dimensions. Thus, the distance between two composite objects p and q should be a monotonically non-decreasing function of each of the following three components:

- **Elementary distance**: a measure of the difference between the *direct values* of the objects (the values of the references in the case of reference types and the embedded values in the case of basic types).
- **Type distance**: a measure of the difference between the objects' *types*, completely independent of the values of the objects themselves.
- **Field distance**: a measure of the difference between the objects' individual *fields*. This will be the same notion of object distance, applied recursively. The fields should be compared one by one, considering only "matching" fields corresponding to the same attributes in both objects; non-matching fields also cause a difference, but this difference is captured by the type distance.

Thus, we may express the formula for the distance $p \leftrightarrow q$:

$$\begin{aligned}
&p \leftrightarrow q = combination\ (\\
&\quad elementary\_distance\ (p, q), \\
&\quad type\_distance\ (type\ (p), type\ (q)), \\
&\quad field\_distance\ (\{[p.a \leftrightarrow q.a] \\
&\qquad |a \in Attributes\ (type\ (p), type\ (q))\}))
\end{aligned} \tag{6.1}$$

[1]We use the term *field* as a dynamic notion corresponding to the static notion of *attribute*. In other words, classes have attributes, while objects have fields.

where $Attributes(t1, t2)$ is the set of attributes applicable to both objects of type $t1$ and objects of type $t2$. We look below at possible choices for the functions $combination$, $elementary\_distance$, $type\_distance$, and $field\_distance$.

For the *elementary distance* we define fixed functions for each possible type (basic or reference) of the compared values p and q:

1. For numbers: $\mathbf{F}(|p - q|)$ (where **F** is a monotonically non-decreasing function with $\mathbf{F}(0) = 0$).

2. For characters: 0 if identical, **C** otherwise.

3. For booleans: 0 if identical, **B** otherwise.

4. For strings: the Levenshtein distance [102].

5. For references: 0 if identical, **R** if different but none is void, **V** if only one of them is void.

In this definition, **C**, **B**, **R**, and **V** are positive values chosen conventionally.

The *distance between two types* is a monotonically increasing function of the sum of their path lengths to any closest common ancestor, and of the number of their non-shared features. In languages where all classes have a common ancestor (*ANY* in Eiffel, `Object` in Java), any two classes have a closest common ancestor. If the types of the two compared objects do not have a closest common ancestor, then the object distance is not defined for them, since the objects must always be compared with respect to a common type of which they are direct or indirect instances. Non-shared features are features not inherited from a common ancestor.

We thus obtain the following formula for the type distance between two types t and u:

$$\begin{gathered} type\_distance\ (t, u) = \\ \lambda * path\_length\ (t, u) + \nu * \textstyle\sum_{a \in non\_shared\ (t,u)} weight_a \end{gathered} \tag{6.2}$$

where $path\_length$ denotes the minimum path length to a closest common ancestor, and non_shared the set of non-shared features. λ and ν are two non-negative constants. $weight_a$ denotes the weight associated with attribute a. It allows for increased flexibility in the distance definition, since thus some attributes can be excluded from the distance (by an associated weight of 0) or can be given increased weight relative to others.

The *field distance* is obtained by recursively applying the distance calculation to all pairs of matching fields of the compared objects:

$$field\_distance\ (p, q) = \overline{\sum_a} weight_a * (p.a \leftrightarrow q.a) \tag{6.3}$$

where a iterates over all matching attributes of p and q. We take the arithmetic mean (the sum divided by the number of its elements) to avoid giving too much weight to objects that have large numbers of fields.

The *combination* function is a weighted sum of its three components. The weights (ϵ for the elementary distance, τ for the type distance and α for the field distance) allow for more flexibility in the distance calculation. Furthermore, each of the three components of the distance must be normalized to a bounded interval with the lower limit 0, so that the distances remain comparable. A normalization function $norm(x)$ should be monotonically increasing and fulfill the property $norm(0) = 0$. By convention, we take 1 as the upper bound of the normalization interval.

The following formula gives the full distance definition combining the previous definitions:

$$\begin{aligned}
&p \leftrightarrow q = \tfrac{1}{3} * (\\
&norm\ (\epsilon * elementary\_distance\ (p, q))\\
&+norm\ (\tau * \lambda * path\_length\ (type\ (p), type\ (q))\\
&+\tau * \nu * \textstyle\sum_{a \in non\_shared\ (type(p), type(q))} weight_a)\\
&+norm\ (\alpha * \overline{\sum_a} weight_a * (p.a \leftrightarrow q.a)))
\end{aligned} \tag{6.4}$$

where in the last term a ranges over all matching fields.

This definition uses several constants and a function, for which any application must choose values. The implementation described in the next section uses the following values:

- 1 for all constants except $\alpha = \frac{1}{2}$ and $R = 0.1$
- the normalization function applied to each component of the distance: $(1 - \frac{1}{1+x}) * max\_distance$, where $max\_distance$ is the upper limit of the interval to which the distance must be normalized. As mentioned above, we used a value of 1 for this limit.

While our experience with the model indicates that these simple values suffice, more experiments are needed to determine correlations between the nature of the application under test and the choice of parameters.

```
used_objects: SET [ANY]
candidate_objects: SET [ANY]
current_best_distance: DOUBLE
current_best_object: ANY
v0, v1: ANY
current_accumulation: DOUBLE
...

current_best_distance := 0.0
foreach v0 in candidate_objects
do
   current_accumulation := 0.0
   foreach v1 in used_objects
   do
      current_accumulation :=
         current_accumulation + distance(v0, v1)
   end
   if (current_accumulation > current_best_distance)
   then
      current_best_distance := current_accumulation
      current_best_object := v0
   end
end
candidate_objects.remove(current_best_object)
used_objects.add(current_best_object)
run_test(current_best_object)
```

Figure 6.1: Algorithm for selecting a test input. The object that has the highest average distance to those already used as test inputs is selected.

6.2 Adaptive Random Testing for object-oriented software (ARTOO)

6.2.1 *Algorithm*

The object distance allows the development of several testing algorithms. We have proposed an algorithm [41] which keeps track of the already used and the available objects and always chooses as input from the available set the object that has the highest average of distances to the already used objects. The algorithm is shown in Figure 6.1.

This algorithm uses pseudo-code but borrows some notations and conventions from Eiffel. In particular, *ANY* is the root of the class hierarchy: all classes inherit from it by default. The *distance* function is implemented as described in the previous section. For simplicity, the algorithm only computes the sum of the distances and not their average; this is a valid approximation since, to get the average distance for every object in the available set, this sum of distances would have to be divided by the number of objects in the already used set, which is constant for a run of the algorithm.

This algorithm is applied every time a new test input is required. For example, for testing a routine *r* of a class *C* with the signature `r (o1: A; o2: B)`, 3 inputs are necessary: an instance of *C* as the target of the routine call and instances of *A* and *B* as arguments for the call. Hence, ARTOO maintains a list of the objects used for all calls to *r*, and applies the algorithm described above every time a new input is required. In other words, when an instance of *C* is necessary, ARTOO compares all instances of *C* available in the pool of objects to all the instances of *C* already used as targets in test calls to *r*. It selects the one that has the highest average distance to the already used ones, and then repeats the algorithm for picking an instance of *A* to use as first argument in the call, and then does the same for *B*.

This strategy is similar to the one originally proposed for ART [35], the differences being the selection criterion (average distance rather than maximum minimum distance) and the computation of the distance measure.

6.2.2 Implementation

We implemented ARTOO as a plug-in strategy for input generation in AutoTest. ARTOO is available in open source at `http://se.inf.ethz.ch/people/ciupa/artoo.html`.

ARTOO only affects the algorithm used for *selecting* inputs. The random *creation* of inputs stays the same, as described in section 5.5. Figure 6.2 represents this graphically, by reproducing figure 5.2 from section 5.5 and pointing out the level at which ARTOO works.

The other parts of the testing process (execution in the two processes, using contracts as an oracle) also remain in AutoTest as described in chapter 5. This is particularly important if one wants to compare the performance of the two strategies (random testing and ARTOO): the conditions under which the experiments are run must be the same.

ARTOO creates new objects and applies diversification operations with the same probabilities as the random strategy. It proceeds differently from the latter only with respect to the selection of the objects (composite and basic) to be used in tests. Its implementation is similar to the algorithm presented above. The main difference is that, while the latter algorithm does not consider the creation of new objects as it proceeds (in other words, no new objects are added to the set of available inputs), in the implementation new instances are created constantly and then considered for selection as test inputs.

The implementation of ARTOO solves infinite recursion in the field distance

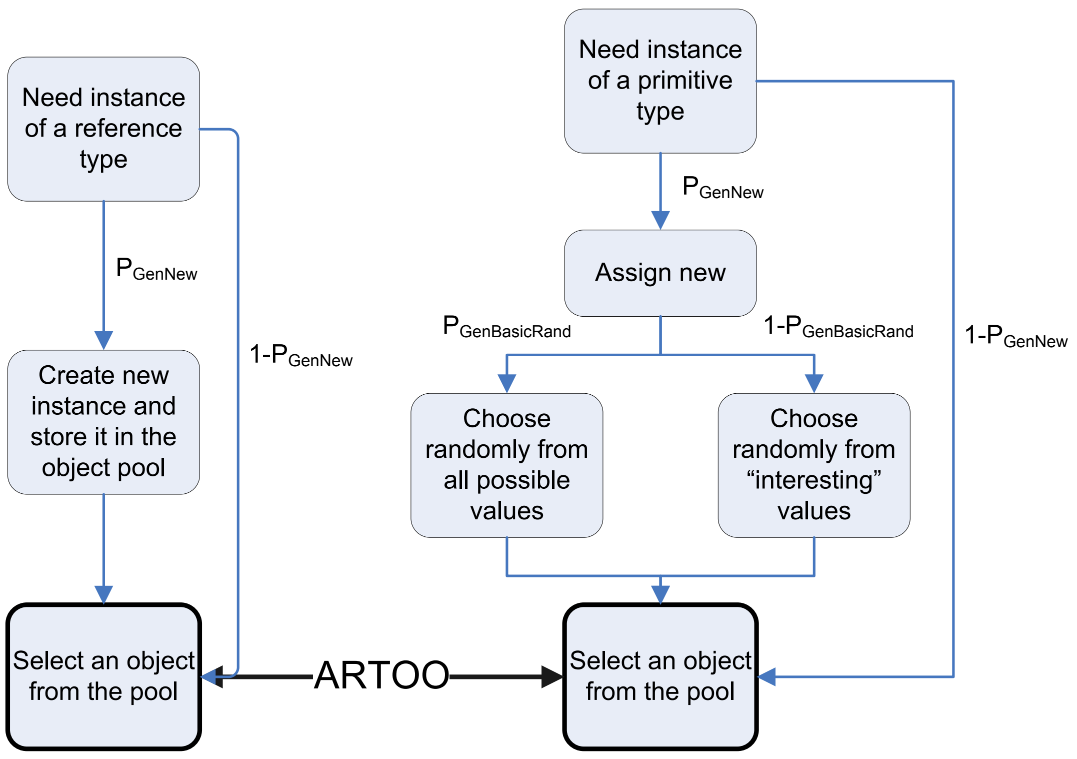
Need instance of a reference type
P_{GenNew}
Create new instance and store it in the object pool
$1-P_{GenNew}$
Select an object from the pool
ARTOO
Need instance of a primitive type
P_{GenNew}
Assign new
$P_{GenBasicRand}$
$1-P_{GenBasicRand}$
$1-P_{GenNew}$
Choose randomly from all possible values
Choose randomly from "interesting" values
Select an object from the pool

Figure 6.2: ARTOO integrated in the input generation process in AutoTest. ARTOO only affects selection of inputs, which are created with the random strategy of AutoTest.

by cutting the recursive calculation after a fixed number of steps (2 in the case of the experiment results presented in section 8.3). Also, the calculation of the object distance is slightly different in the implementation of ARTOO than in formula 6.4, in that no normalization is applied to the elementary distances as a whole: for characters, booleans, and reference values the given constants are directly used, and for numbers and strings the normalization function given in section 6.1 is applied to the absolute value of the difference (for numbers) and to the Levenshtein distance respectively (for strings). For the field distance, no normalization is necessary, since the averaged distances between the objects referred by the attributes are themselves bounded to the same interval.

ARTOO needs to store inputs used in calls to routines under test before these calls are performed, so that, in case the interpreter encounters an error it cannot recover from during the execution of the call, the information about the used inputs is not lost. ARTOO hence serializes test inputs before every call to a routine under test.

The results of an experimental evaluation of ARTOO's performance are presented in section 8.3.

6.2.3 *Example*

The following example shows how the implementation of ARTOO in AutoTest proceeds. Suppose ARTOO tests the class *BANK_ACCOUNT*, given in Listing 6.1.

```
class BANK_ACCOUNT
create
 make

feature -- Bank account data
 owner: STRING
 balance: INTEGER

feature -- Initialization
 make (s: STRING; init_bal: INTEGER) is
    -- Create a new bank account.
   require
    positive_initial_balance: init_bal >= 0
    owner_not_void: s /= Void
   do
    owner := s
    balance := init_bal
   ensure
    owner_set: owner = s
    balance_set: balance = init_bal
    end

feature -- Operation

  withdraw (sum: INTEGER) is ...

  deposit (sum: INTEGER) is ...

  transfer (other_account: BANK_ACCOUNT; sum: INTEGER) is
      -- Transfer 'sum' to 'other_account'.
    require
      can_withdraw: sum <= balance
    do
      other_account.deposit (sum)
      balance := balance - sum
    ensure
      balance_decreased: balance < old balance
      sum_deposited_to_other_account: other_account.balance
          > old other_account.balance
    end
invariant
  owner_not_void: owner /= Void
  positive_balance: balance >= 0
end
```

Listing 6.1: Part of the code of class *BANK_ACCOUNT*, which ARTOO must test.

For testing routine *transfer*, ARTOO needs an instance of *BANK_ACCOUNT* as the target of the call, another instance of *BANK_ACCOUNT* as the first argument, and an integer as the second argument. For the first test call to this routine, there are no inputs previously used, so ARTOO will pick objects with corresponding types at random from the pool. Suppose at this point the pool contains the following objects:

```
ba1: BANK_ACCOUNT, ba1.owner="A", ba1.balance=675234
ba2: BANK_ACCOUNT, ba2.owner="B", ba2.balance=10
ba3: BANK_ACCOUNT, ba3.owner="O", ba3.balance=99
ba4 = Void
i1: INTEGER, i1 = 100
i2: INTEGER, i2 = 284749
i3: INTEGER, i3 = 0
i4: INTEGER, i4 = -36452
i5: INTEGER, i5 = 1
```

Suppose ARTOO picks *ba3* as target, *ba1* as first argument and *i5* as second argument for the first call to *transfer*. These 3 values are saved to disk and then the call is executed:

```
ba3.transfer(ba1, i5)
```

The state of the object pool is now as follows:

```
ba1: BANK_ACCOUNT, ba1.owner="A", ba1.balance=675235
ba2: BANK_ACCOUNT, ba2.owner="B", ba2.balance=10
ba3: BANK_ACCOUNT, ba3.owner="O", ba3.balance=98
ba4 = Void
i1: INTEGER, i1 = 100
i2: INTEGER, i2 = 284749
i3: INTEGER, i3 = 0
i4: INTEGER, i4 = -36452
i5: INTEGER, i5 = 1
```

For the next call to *transfer*, ARTOO chooses a target by picking the non-void object of type *BANK_ACCOUNT* that is furthest from the already used target. For the first argument it picks the instance of *BANK_ACCOUNT* that is furthest from the first argument previously used, and likewise for the integer argument. It thus chooses *ba1* as target, *ba4* as first argument (void references always have the maximum possible distance to non-void references), and *i2* as second argument. These values are saved and the call is executed:

```
ba1.transfer(ba4, i2)
```

This triggers an attempt to call a routine on a Void target (through the instruction `other_account.deposit (sum)` in the body of routine *transfer*), which results in an exception, so ARTOO has found a fault in routine *transfer*. The precondition of this routine should state that *other_account* must be non-void.

The pool is not changed and ARTOO picks again the objects with the highest average distances to the already used ones. So *ba2* is chosen as both target and first argument, and *i4* as second argument. The values are saved and the call is executed:

```
ba2.transfer(ba2, i4)
```

This triggers a postcondition violation in *transfer* since trying to transfer a negative amount from the current account does not reduce the balance of the current account. ARTOO has thus found another fault, since transferring negative amounts should not be allowed. It is interesting to note that this call actually exposes another fault, namely that transferring money from an account to itself should not be allowed.

For simplicity this example did not consider the creation of objects between test calls. In practice, objects are created with a certain probability and added to the pool between calls to routines under test, so ARTOO also considers these new instances when selecting the inputs.

6.2.4 *Optimization*

The ARTOO algorithm described above computes, before each call to a routine under test, distances between all pairs of already used and available objects. As testing proceeds, both the number of already used and the number of available objects increases[2], and so does the time required to run the ARTOO algorithm.

To remove this performance penalty, we have developed a version of the ARTOO algorithm that considers a constant number of objects for every input selection. In the following we refer to this version of the ARTOO algorithm as ARTOO*. ARTOO* was implemented by Cosmin Mitran in his Diploma Thesis [113] as an input generation plug-in for AutoTest.

ARTOO* proceeds in the following way for every input:

1. It generates N candidate inputs. A candidate input can either be a newly created object or an object selected randomly from the pool and diversified. If no objects of the required types are yet available in the pool, ARTOO* skips the diversification step. Otherwise, it selects randomly one of the objects of the required type available in the pool and calls on it one of its procedures. If this procedure needs arguments, they are created with the basic random strategy.

2. It randomly selects K of the already used objects.

3. It computes the distances between the N candidate inputs and the K already used objects.

4. It selects the candidate that is the furthest away from the K already used to be used as input for the test.

5. Out of the α objects that are closest to the K already used ones, the newly created ones are removed from the object pool.

In this algorithm N, K, and α are constants. Experimental results detailed in [113] indicate that the values chosen for N and α have the biggest influence on testing results. Overall, $N = 3$, $\alpha = 2$ and $K = 5$ delivered the best results.

[2]It must be noted that, due to the interpreter restarts during the testing session emptying the object pool, the number of available inputs does not grow constantly during the testing session.

6.3 Combining random testing, ARTOO, and manual testing

Manual tests generally reflect a tester's intuition of which inputs could uncover faults in the software under test. Starting from this assumption and based on the ideas of ARTOO, we developed a testing strategy that selects inputs based on two criteria:

- Minimal distance to inputs used in manual tests and
- Maximal distance to inputs already used in automated tests.

We first present the details of this testing algorithm and then its implementation in AutoTest.

6.3.1 *Algorithm*

To combine the two criteria for input selection, this testing strategy computes, for every automatically generated candidate input, a combination of its average distance to already used inputs (AD) and its average distance to manual inputs (MD). Combining these two measures into a formula yields: $D = A * AD + B * MD$, where A and B are constants.

To be selected, a candidate should have a high AD and a low MD. For the implementation of this testing strategy we chose A = 1 and B = -1, hence $D = AD - MD$. The candidate with the highest value for D is chosen to be used in the test.

As an optimization, we further group the manual objects into *clusters* and then compute the distances of candidate objects only to the cluster centers, rather than to individual manual objects. We used the maxi-minimum clustering algorithm, which proceeds as follows, given a set S of objects that must be grouped into clusters:

1. Select the 2 objects that are furthest away and create 2 clusters having them as centers. Remove these 2 objects from set S.
2. Select an object from the set S and compute its distance to the centers of all clusters. Assign the object to the cluster to whose center it is closest and remove it from set S.
3. Repeat step 2 until set S is empty.
4. Compute the average distance D between the centers of all clusters.
5. Take one of the existing clusters, C, and select its member object O that is furthest away from the center of the cluster. If the distance between O and the center of C is greater than D/2, then create a new cluster with the center

in O and remove all members from cluster C except its center, and add them to S.

6. Repeats step 5 until all clusters are processed.
7. If there is at least one new cluster, go to step 2, otherwise the algorithm is finished.

6.3.2 *Implementation*

In the spirit of the optimization for ARTOO described in section 6.2.4, the implementation for this testing strategy also uses constant numbers of objects for calculating distances. It proceeds very similarly to ARTOO*, with the exception of the selection criterion:

1. It generates N candidate inputs, by creating new objects and by diversifying existing ones from the pool. Manual objects are never diversified.
2. It randomly selects K of the already used objects.
3. It computes the average distance AD between each of the N candidate inputs and the K already used objects.
4. It computes the average distance MD between each of the N candidate inputs and the manual inputs used for testing the feature.
5. It selects the candidate with the highest $D = AD - MD$ to be used as input for the test.
6. Out of the α objects with the lowest D, the newly created ones are removed from the object pool.

Experiments using a prototype implementation [113] showed that such a combined strategy can reduce the number of tests to first fault by an average factor of 2 compared to the basic implementation of ARTOO.

CHAPTER 7

TOWARDS BETTER CONTRACTS: CONTRACT INFERENCE WITH CITADEL

As shown in chapter 5, the presence of contracts embedded in software source code is essential for automated testing solutions. Contracts are also important in debugging, because they help locate faults: a precondition violation signals a fault in the caller of a routine, while a postcondition or an invariant violation signals a fault in the routine itself. Thus the location of the contract violation is typically very close to the location of the actual fault in the software. This property of contracts is important for all types of tests, from the unit level to the system level: because classes in O-O programs are typically closely coupled and need to cooperate for achieving tasks, in the event of a software failure it is hard, in the absence of contracts, to accurately lay blame on the software module containing the fault. Briand et al. [28] show that the effort of isolating a fault once a failure occurred is reduced approximately 8 times in the presence of contracts.

Given these benefits of contracted code, Eiffel developers do indeed include assertions in the programs they write, as shown in Chalin's study [31]. However, these assertions are most often incomplete and sometimes incorrect, in the sense that they do not reflect the *intended* specification of the software, as discussed in detail in section 8.4.1.

One way to improve these existing contracts is by inferring additional ones from the implementation. This is the approach that the Daikon tool [55] takes: given a set of passing test cases that exercise the code, Daikon infers assertions that hold at various program points (e.g. routine entry and exit) for the executions of the system through the test cases. Generalizing from these observations, one concludes that these assertions may hold for all program runs.

Daikon has already been used by several testing tools for automatically inferring the program specification, as detailed in section 4.3.3, but to our knowledge no studies have yet been carried out about how programmer-written assertions compare to inferred ones. A clear characterization of how programmer-written and inferred assertions compare is necessary in order to understand if and how assertions inference tools can assist humans in writing assertions and if they play distinct roles in producing high-quality executable specification.

To investigate these questions, we developed an Eiffel front-end for Daikon which allows us to infer contracts for Eiffel programs using the Daikon engine. The resulting tool is called CITADEL (Contract Inference Tool Applying Daikon to the Eiffel Language) and was implemented as a Master's project by Nadezda Polikarpova [126].

This chapter first gives an overview of the Daikon tool (section 7.1) and describes the Eiffel front-end (section 7.2). It then presents the experiments we ran to compare inferred to programmer-written contracts, and analyzes and interpretes the experimental results (section 7.3).

7.1 Daikon

7.1.1 Overview

Daikon was created by Michael Ernst in the late 1990's and is still being further developed and improved today. It has been used both unchanged and with various modifications by numerous other research projects[1] and has also been included in industrial tools, such as Agitar's AgitarOne, previously called Agitator [26].

Daikon is a contract inference tool. Its contract detection process is based on checking a set of assertion templates over all variables and combinations thereof in scope at a certain program point. More exactly, Daikon maintains a list of assertion templates which it instantiates using various program variables at routine entry and exit and checks if they hold for the executions of the program through a given set of test cases. As soon as an assertion does not hold for an execution, it is eliminated and not checked again for further executions.

Figure 7.1 shows an overview of the architecture of the tool. The main steps involved in the assertion inference process are the following:

- The program is instrumented so that, at certain program points, it saves the values of the variables in scope to a data trace file.
- The instrumented program is exercised through a test suite. Each run of the program results in a data trace file.

[1]A continuously updated list is available at `http://groups.csail.mit.edu/pag/daikon/pubs/#daikon-methodology`.

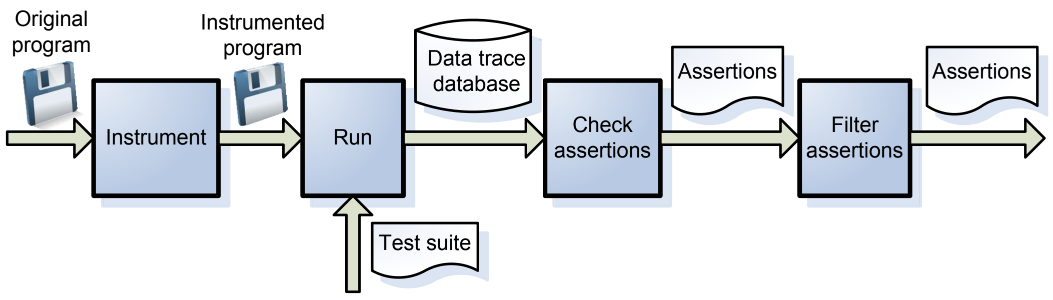

Figure 7.1: Architecture of the Daikon tool (adapted from [55]).

- Daikon creates the list of potential assertions and checks them against the variable values recorded in the data trace files.
- During assertion checking, Daikon also performs two other actions: it filters out those assertions that are likely not relevant to programmers and it introduces derived variables.

The following subsections present some of these steps in more detail.

7.1.2 *Contract detection process*

As explained above, Daikon detects assertions by checking if they hold for the variable values recorded in the data trace file. For simplicity and language independence, Daikon supports only a small set of data formats: integers, floating point numbers (only minimally supported), booleans, strings, hashcode (a special type for representing memory addresses) and sequences of these types. Hence all values recorded in the data trace files must have one of these formats.

The assertion templates that Daikon checks fall into the following categories, where x, y and z are variables and a, b and c are constants:

- Assertions over any variable: constant value ($x = a$), small value set ($x \in \{a, b, c\}$), uninitialized
- Assertions over a numeric variable: range limits ($x \geq a$, $x \leq b$), modulus ($x \bmod a = b$), etc.
- Assertions over 2 numeric variables: linear relationship ($y = a * x + b$), ordering ($x < y$, etc.), functions ($y = fn(x)$, where fn is a built-in unary

function), etc.

- Assertions over 3 numeric variables: linear relationship ($z = a*x+b*y+c$, etc.), functions ($z = fn(x, y)$, where fn is a built-in binary function)
- Assertions over a sequence variable: minimum and maximum sequence values, element ordering, invariants over all sequence elements, etc.
- Assertions over 2 sequence variables: linear relationship elementwise, lexicographic comparison ($x < y$, etc.), subsequence relationships, etc.
- Assertions over a sequence and a numeric variable: membership ($x \in y$)

The complete list of assertion templates that Daikon checks can be found in [55]. Users can also add their own assertion templates and variable derivation rules. In the experiments described in section 7.3, we used a subset of Daikon's standard template library — we did not check for a few very seldom occurring templates such as those involving bit-wise operations, the "OneOf" template for references (stating that the address of an object attached to the reference is always the same or has only very few values), the "NonEqual" template (which results in many irrelevant inequalities) and for most classes also the "Numeric" template (which yields assertions with multiplication, division, power, modulo).

Daikon checks assertions at routine entry and exit. This way it can infer routine pre- and postconditions. Daikon also identifies assertions that hold at both entry and exit of all public routines, promoting them to class invariants. Daikon can also detect assertions involving static class variables, but, as Eiffel does not support such variables, this property is not relevant to our study.

7.1.3 Selection of reported contracts

Daikon implements several strategies for selecting, out of the complete list of assertions that held during the program executions, those that are most likely to be relevant to programmers. In the Daikon terminology, an assertion is *relevant* "if it assists a programmer in the programming activity" [55].

Additionally, to achieve the same purpose of generating assertions relevant to programmers, Daikon can derive variables from the ones recorded in the data trace file. These variables also participate in the invariant inference process just like the normal ones. Here are some examples of variables that Daikon derives:

- Variables derived from sequences: length of the sequence, extremal elements (first, second, last, last but one)
- Variables derived from numeric sequences: sum of the sequence elements, minimum and maximum element
- Variables derived from numeric sequences and numeric variables: sequence element at a certain index, subsequences based on an index

- Variables derived from function invocations: number of calls

Daikon can also derive variables from derived variables. The derivation process stops at depth 2.

A further optimization that Daikon performs relates to variables that it determines to be equal: it chooses one variable from the set of equal variables and removes the others from the set of variables that it analyzes and derives from.

Daikon only reports assertions that it considers to not hold by chance, only in the executed tests. To determine this, for each inferred assertion it calculates the probability that the assertion would hold for random input. Only if this probability is lower than a user-specified confidence threshold does Daikon report the assertion. This is called a confidence test. Every assertion template has its own confidence test.

These confidence tests are dependent on the number of times an assertion is observed to hold at a certain program point. However, if the program point is executed several times without a variable in the assertion being assigned to between the executions of the program point, the statistical tests for that variable can be unjustly influenced. Therefore Daikon implements several strategies for deciding if an observation of a variable value should contribute to the confidence level in an assertion or not. We do not present all these strategies here; details can be found in [55]. Out of these strategies, experiments performed by the Daikon developers showed the "changed value" strategy to work best. In this approach, a variable value observed in a sample only contributes to the assertion confidence level if the variable value has changed since the last time the assertion was evaluated at that program point.

So overall Daikon applies 3 tests to decide if it should report an assertion:

- There should be sufficiently many observations of the variable values
- There should be sufficiently many samples contributing to the confidence level
- The statistical confidence level of the assertion should be above the user-provided limit

Daikon also tries to not report assertions that are already implied by others. It achieves this by:

- Not introducing redundant derived variables (ones that are equal to existing variables)
- Not checking assertions whose evaluation it already knows
- Pruning more redundant assertions before displaying the results

The first two steps have the highest impact on Daikon's performance.

Daikon does not use a general-purpose theorem prover for deriving implication relationships between assertions. Rather, it uses a set of hard coded checks for ways in which such implications can occur between the known assertions templates.

Daikon eliminates further assertions by determining that they involve unrelated variables. As an example, as assertion of the form `person.age > person.number_of_children` stating that a person's age is always greater than the number of children that person has is true, but very likely uninteresting for programmers. Several approaches for deciding if two variables are comparable (related) or not are possible: variables declared of the same type are always comparable; variables whose types are coercible are comparable; variables that occur in the program in the same expression are comparable (this is the basis of the Lackwit typing mechanism [117]); all variables are comparable. The experiments reported in [55] suggest that the Lackwit approach is the most appropriate in terms of performance and detection of relevant assertions.

7.1.4 Test suite selection

The test suite used to exercise the program has a clear influence on the quality of the inferred assertions. Several questions arise:

- How does test suite size influence the inferred assertions?
- How do other measures of test suite quality (such as code coverage) influence the inferred assertions?
- Can automatically generated tests be used for assertion inference and, if so, which generation method produces the best results?

Increasing test suite size affects inferred assertions through (1) increased confidence levels (due to the higher number of executions of the program points) and (2) potentially more falsified invariants. Increasing a quality measure of the test suite, such as coverage, is intuitively also likely to produce better inferred assertions by falsifying more irrelevant assertions.

Several studies of the effect of the test suite on Daikon-inferred assertions have been carried out both by the Daikon developers and by other researchers. Nimmer and Ernst [115] showed that Daikon produces, even from relatively small test suites, assertions that are consistent and sufficient for automatic verification (proving the absence of runtime errors) with very little change. Nimmer and Ernst [116] also showed that test cases exercising mainly corner cases are not suited for assertion inference. Gupta et al. [69] showed that existing code coverage criteria (branch coverage, definition-use pair coverage) do not provide test suites that are good enough for invariant detection (they produce many false, test-suite-dependent invariants); however, test suites that satisfy these traditional criteria produce more relevant assertions than random.

7.1.5 Performance and usability

The most important factor affecting Daikon's performance is the number of variables it has to examine at every program point. In the worst case, assertion inference is cubic in this number of variables, due to the invariant templates over 3 variables. Assertion inference is also linearly dependent on the following:

- The number of program points
- The number of times a program point is executed
- The number of assertions that are checked at every program point (In practice this number keeps decreasing as execution progresses, because invariants are not checked anymore after they are falsified once.)

The best single predictor for the time required by the assertion detector is the number of pairs of variable values that the assertion detector encounters [55]. This number correlates in turn with the total number of values, but cannot be predicted from other factors.

A study of users' experience with Daikon [116] showed that using Daikon does not speed up or slow down users trying to annotate programs with assertions, but it improves recall (how many of the assertions written by users are correct) and bonus (ratio of verifiable assertions to the minimal set). Half of the users participating in this study considered Daikon to be helpful, especially because they could use the generated assertions as support for finding others. More than half the users found removing incorrect assertions easy. About a third of the users complained about the textual size of the generated assertions.

7.2 CITADEL

The Daikon front-end for Eiffel is called CITADEL. It was implemented as part of a master's thesis by Nadezda Polikarpova [126].

CITADEL performs the following steps:

- It instruments the Eiffel class(es) for which it must infer assertions so that its/their execution produces a trace file in the format required by Daikon.
- It generates a declarations file describing the format of the data trace file.
- After running the given tests on the instrumented program, it feeds the generated data trace files to the Daikon engine, which infers likely assertions.

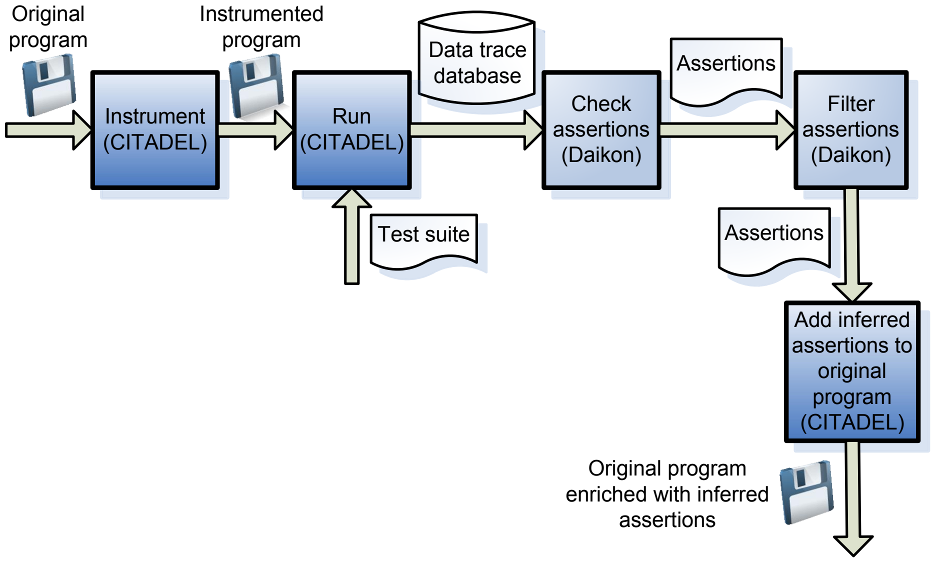
Original program
Instrumented program
Data trace database
Assertions
Instrument (CITADEL)
Run (CITADEL)
Check assertions (Daikon)
Filter assertions (Daikon)
Test suite
Assertions
Add inferred assertions to original program (CITADEL)
Original program enriched with inferred assertions

Figure 7.2: Overview of the assertion inference process for Eiffel programs. Adaptation of Figure 7.1, showing which steps are performed by CITADEL and which by Daikon.

- It adds these inferred assertions to the program text.

Figure 7.2 shows an overview of the process, evidencing which steps are performed by CITADEL and which by Daikon.

Instrumentation and examined variables

The instrumenter component of CITADEL adds logging instructions to Eiffel programs at points of interest. In its current implementation, these points are: routine entry (for inferring preconditions), routine exit (for inferring postconditions) and loops (for inferring loop invariants). At each such point, the instrumented version of the program will output the values of variables in scope to the data trace file. These variables are the following:

- At routine entry: the **`Current`** object, routine arguments (if any), and all zero-argument queries of the enclosing class exported to the same clients as the routine itself
- At routine exit: the **`Current`** object, routine arguments (if any), all zero-

argument queries of the enclosing class, and **`Result`**

- For object invariants (inferred from assertions that hold at all entry and exit points for exported routines): the **`Current`** object and all zero-argument queries of the enclosing class

- For loop invariants: the **`Current`** object, routine arguments (if any), all zero-argument queries of the enclosing class, **`Result`** (if the routine is a query), and local variables, if any

Using function results as part of the variables whose values Daikon examines raises two problems. First, executing the functions to get the return values can have side effects. The Daikon front-end for Java solves this problem by requiring that users who turn on the option of including functions in assertions provide a so-called "purity file", listing all functions which are side-effect free and hence safe to evaluate at runtime as part of the assertion inference process. In Eiffel the situation is different, because of the command-query separation principle, which states that it is should be transparent to clients if a zero-argument query is implemented through a function or attribute. Hence, to be consistent with this principle, in CITADEL we opted for including both attributes and functions with no arguments in inferred assertions. There is indeed no mechanism in Eiffel preventing side-effects in functions, but such side-effects are strongly discouraged. Therefore we consider that the added expressiveness of inferred assertions including zero-argument functions outweighs the potential danger of changing system state through assertion evaluation.

Second, it can be the case that the programmer-written precondition of a function is not satisfied when the function is called as part of the state-logging process; any valued returned by a function when executed outside of its precondition is not representative of the functions's behavior. CITADEL hence checks, before any function evaluation, that the function's precondition holds. If this is not the case, CITADEL just indicates that the function cannot be evaluated by using the special Daikon value "nonsensical". This value is also used when the target object on which the query should be evaluated is Void.

The instrumenter also creates the declarations file, which contains declarations of program points. Such declarations must contain the name of the program point and information about the variables whose values are recorded in the data trace file for the program point: the variable's name, its type, and whether the variable is comparable or not.

Given this declarations file and the data trace files, Daikon infers likely assertions at the program points listed in the declarations file. It outputs these inferred invariants to standard output in textual form and also serializes them to a file as Java objects representing the inferred invariants. The annotator component of CITADEL takes these inferred assertions and adds them to the source code of the Eiffel classes on which it was run.

CITADEL currently supports almost all Eiffel's advanced language constructs and allows inferring contracts for real-world classes. It also has some limitations: for example, currently deferred and external features (written in other languages) cannot be instrumented, and the return values of functions with arguments are not monitored and hence cannot be included in the inferred assertions.

7.3 Inferred vs. programmer-written contracts

We developed CITADEL in order to investigate how automatically inferred assertions compare to the ones written by programmers, with the final purpose of determining to what degree an assertion inference tool can assist programmers in the assertion-writing process.

In the context of Design by Contract, the benefits of a Daikon-like assertion inference tool are inherently limited by the tool's dependency on the existence of an executable for the target system: as explained in detail above, Daikon works by monitoring the states reached at runtime by a system executed through a set of passing test cases, and by inferring the conditions that these states fulfill. This naturally implies the existence of a fully functional system or at least one implementing a subset of its use cases, whereas in the Design by Contract software development method programmers are encouraged to write contracts before or at the same time as the implementation. The reason for this is that contracts play a role starting from the analysis and design phases of software development: writing contracts helps developers understand requirements and split up functionality between software modules, so that each module implements a certain subset of the system functionality, and it achieves this under certain conditions. These benefits cannot be achieved by a tool like Daikon.

In the experiment we set out to answer the following questions, relating to the absolute quality of the inferred assertions and to how these compare to programmer-written ones:

- How many of the assertions inferred by CITADEL are correct and interesting?
- How many of the programmer-written assertions are implied by the inferred assertions and vice-versa? Is there an inclusion relationship between these two types of assertions?
- How can assertion inference be used to assist programmers or to improve the programmer-written assertions?
- What factors influence the quality of the inferred assertions? More precisely, can we find correlations between any code metrics and the quality of the assertions inferred for that code?

It is likely that a developer examining the correct and interesting inferred assertions would find some of them more important than others. We did not investigate the question of how many inferred assertions would be classified by a developer as "important" or "worth adding to the code", because for such a subjective decision the input of a creator or maintainer of the code would be necessary and we did not have the possibility of involving such a person in the study.

7.3.1 Experimental setup

To answer these questions, we ran CITADEL on 15 Eiffel classes, 11 of them taken from 3 widely-used Eiffel libraries and 4 written by students of Computer Science. None of these classes were written especially for the study or modified in any way. We tried to choose classes of different size, depth in the inheritance hierarchy, with and without loops and also with diverse, but clear semantics.

Table 7.9 shows various code metrics for each class used in the experiment. In the last column, it shows the percent of programmer-written assertions from the instrumented routines that are expressible in Daikon's grammar. This percent varies widely, from a fourth of assertions for class *GENEALOGY_2* to all assertions for class *ST_WORD_WRAPPER*.

In the figures given below, to save space, we did not include the full names of the classes, but used the abbreviations shown in table 7.9 instead.

The classes taken from libraries come from Base, Time and Gobo, libraries which are included in the standard distribution of the most popular IDE for Eiffel (EiffelStudio [2]) and are used by virtually all applications written in Eiffel. We used the versions of these library included in version 6.2 of EiffelStudio. These classes are highly reusable and presumably the effort spent to ensure their quality is accordingly high. In particular, library classes are usually equipped with relatively high quality contracts.

Because we wanted to also include in the study code written by less experienced programmers, we also ran CITADEL on classes created by students of Computer Science at the ETH Zurich: classes *FRACTION1* and *FRACTION2* were implemented as assignments in an introductory programming course in fall 2007 and classes *GENEALOGY1* and *GENEALOGY2* were implemented as part of a project given in a software engineering course in spring 2008, course which could be taken by students starting from the third year of their undergraduate studies. These last two classes had a given common interface, including some contracts. The students were not allowed to change the routine preconditions, but they could add clauses to the other contracts. This must be considered when interpreting the results of our study, because the contracts were not written by the students alone, hence it can be assumed that the contract quality is higher than for applications written entirely by students.

We ran CITADEL on each class with 3 manually-constructed test suites of different sizes:

- A small test suite, containing approximately 10 calls with different random inputs to each instrumented routine and exercising the most typical behavior of the class
- A medium-size test suite, which adds some partition tests that exercise less typical behavior and contains about 50 calls to each instrumented routine, and
- A large test suite, which does not differ from the medium one qualitatively, but has 10 times more routine calls (naturally using different inputs); this test suite hence contains about 500 calls to each instrumented routine.

As there have been previous investigations [55, 115, 116, 69] of the effect of test suite size and other characteristics on the quality of assertions inferred by Daikon, the purpose of our study was not related to such an investigation. Nevertheless, we still evaluated CITADEL's results on each class for 3 different test suites because, as previous studies show, various characteristics of the test suite influence the results of the tool.

7.3.2 Results

To give the reader an idea of the number and kind of assertions that CITADEL infers, appendix A contains the listing of class *ST_SPLITTER* annotated with the contracts inferred by CITADEL based on the medium test suite.

In the following we present the results grouped by the overarching questions that they are trying to answer: investigating the quality of the inferred assertions (IA) in absolute terms and comparing them to programmer-written assertions (PA). Since we use 3 different test suites to generate IA and the test suite has a significant influence on the results, we present the quality measures for each test suite separately.

Investigating the quality of the inferred assertions

To define the quality of the IA, we looked at two measures: the percent of correct assertions out of all IA and the percent of relevant assertions out of all IA. By definition, we say that an IA is *correct* if it reflects a property of the source code. We say that an IA is *relevant* if it is correct and it expresses a property that is interesting. We identify uninteresting IA based on:

- semantics – IA involving unrelated variables whose relation is purely accidental are uninteresting. For instance, an assertion of the form `person.birth_year > person.number_of_siblings` is true (assuming that years are represented as integers with 4 digits), but most likely uninteresting.

Table 7.1: Averages of the percentages of correct inferred assertions.

	Small TS	Medium TS	Large TS
Loop invariants	80%	90%	93%
Preconditions	46%	80%	83%
Postconditions & class invariants	68%	90%	92%
Total	67%	89%	91%

- constant definitions – IA that are trivially true because they refer to constants (e.g. `time1.hours_in_day = time2.hours_in_day`, where *time1* and *time2* are instances of the *TIME* class, which has a constant attribute *hours_in_day*) are uninteresting.
- trivial implication (no knowledge of the source code is necessary to deduce it) – IA that are trivially implied by other IA at the same program point are uninteresting. As an example of a trivial implication, if it is already inferred that `(sorted_items /= Void)= Result`, then an assertion `(sorted_items = Void)= (not Result)` is uninteresting.
- misplacement of assertions – IA that conceptually belong in another type of contract than where they were inferred are uninteresting. For instance, a pre- or postcondition about an entity *x* that is implied by *x*'s class invariant is uninteresting.

This provides an admittedly lenient definition of relevancy, but it is also an objective definition. Another option would have been to ask a developer or maintainer of the tested code to rate the relevancy of the IA, but this was not possible in our case.

Figure 7.3 shows the percent of correct IA other than loop invariants for each class, for each test suite size. Figure 7.4 shows the percent of correct inferred loop invariants for the classes containing loops. Figure 7.5 shows the total percent of correct IA. Table 7.1 shows the averages, over all classes, of the percent of correct IA, for each test suite size.

The medium-size test suite generally brings a substantial improvement over the small one. The large test suite only brings a small improvement over the medium-size one, if at all. For 13 of the classes, more than 80% of the assertions inferred for the medium-size and large test suites are correct; this percentage drops under 50% only for one class. For 4 classes, 100% of the assertions inferred for the medium and large test suites are correct. For 4 other classes, this proportion exceeds 95%. For the large test suite, the average correctness of IA exceeds 90%.

Figure 7.6 shows the percent of relevant IA other than loop invariants for each class, for each test suite size. Figure 7.7 shows the same information for inferred

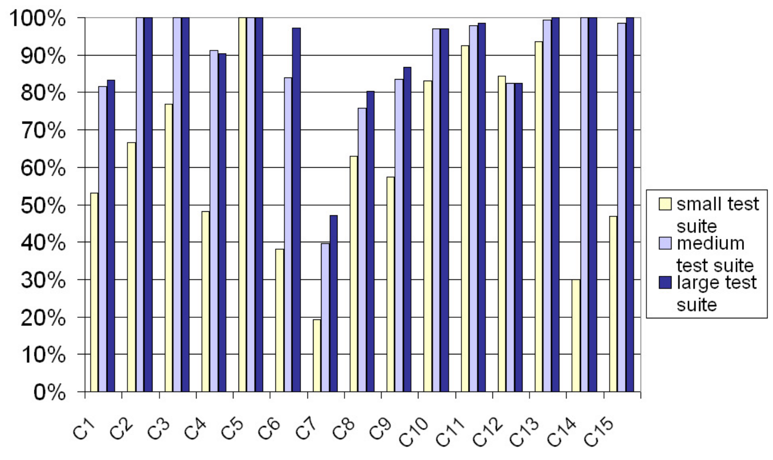

Figure 7.3: Percentage of correct inferred assertions excluding loop invariants.

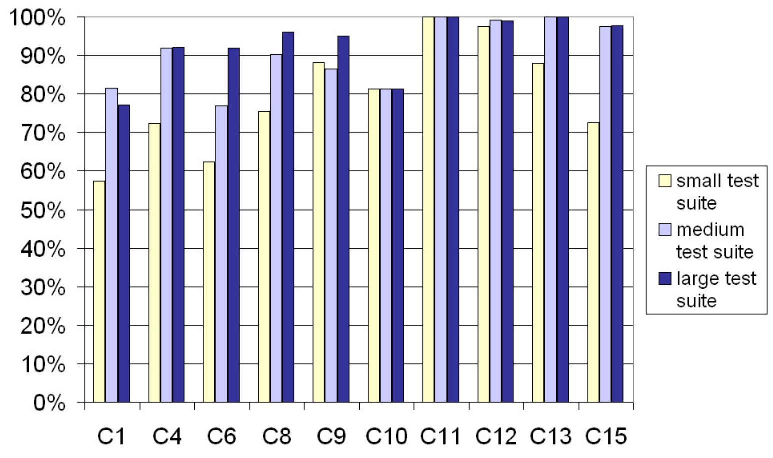

Figure 7.4: Percentage of correct inferred loop invariants (only for the classes containing loops).

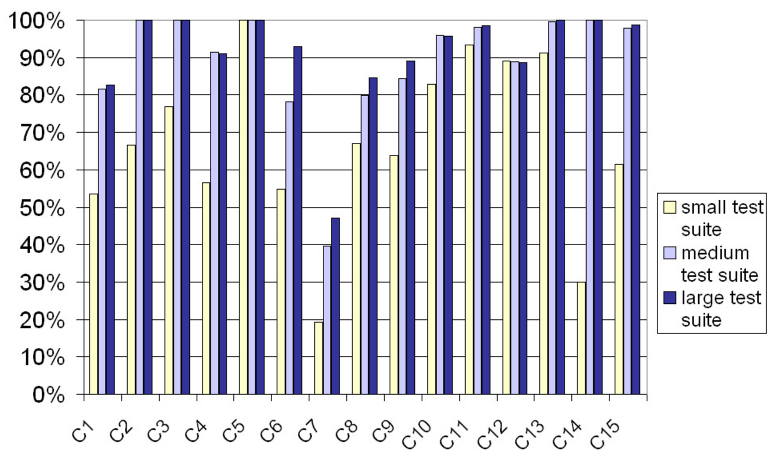

Figure 7.5: Total percentage of correct inferred assertions.

Table 7.2: Averages of the percentages of relevant inferred assertions.

	Small TS	Medium TS	Large TS
Loop invariants	63%	70%	72%
Preconditions	33%	53%	54%
Postconditions & class invariants	57%	69%	67%
Total	53%	66%	66%

loop invariants (only for the classes containing loops) and figure 7.8 shows the total percent of relevant IA. Table 7.2 shows the averages, over all classes, of the percent of relevant IA, for each test suite size.

Again, the medium-size test suite generally brings an improvement over the small one, but for 4 classes smaller percents of relevant assertions are found through the medium and large test suites than through the small one. The percents of relevant assertions vary widely, from less than 20% to 100%; section 7.3.4 discusses various code metrics to which relevancy correlates and provides possible explanations for its variations. On average, around 65% of IA are relevant for the medium and large test suites.

Comparing inferred assertions to programmer-written assertions

The first measure that we use to compare inferred to programmer-written assertions is *recall*, showing what proportion of the PA were inferred or implied by

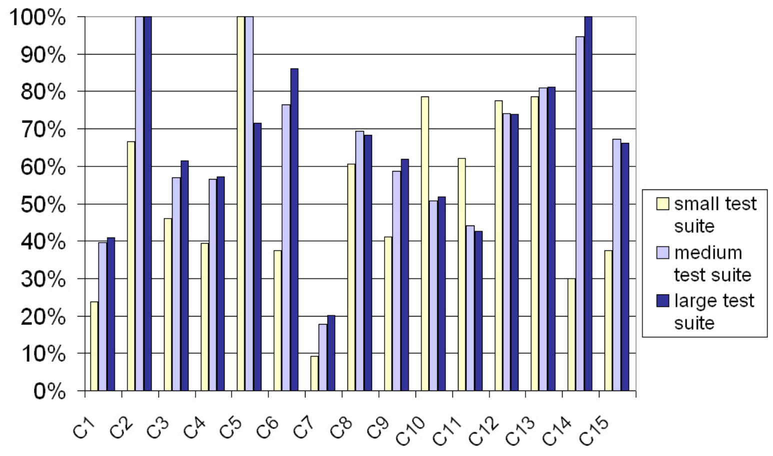

Figure 7.6: Percentage of relevant inferred assertions excluding loop invariants.

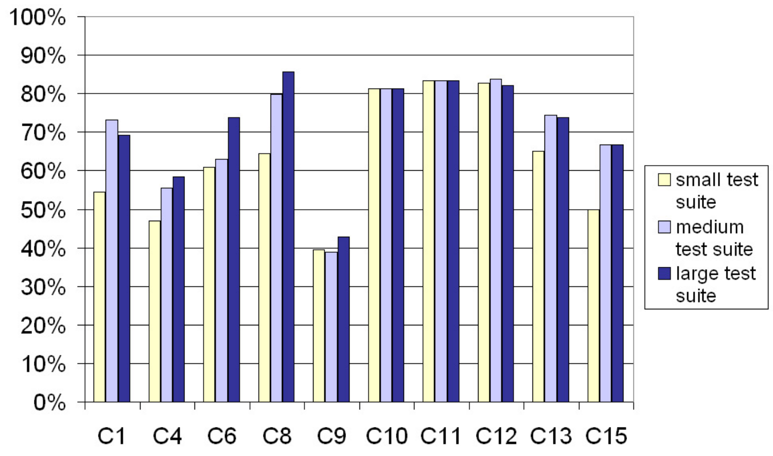

Figure 7.7: Percentage of relevant inferred loop invariants.

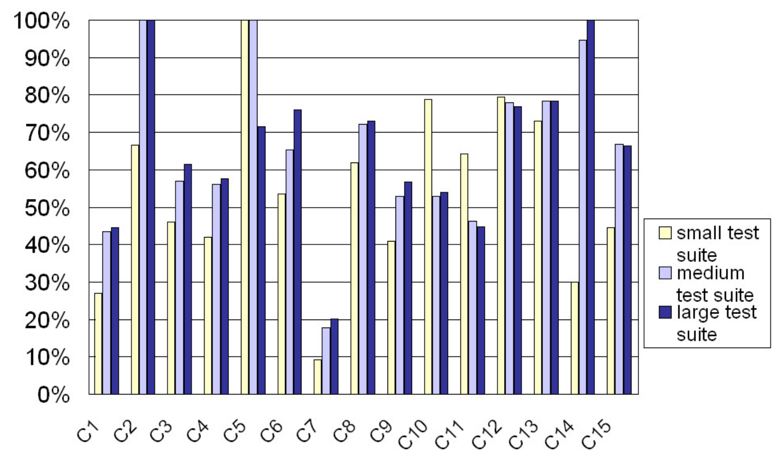

Figure 7.8: Total percentage of relevant inferred assertions.

inferred assertions. We consider two types of recall: recall of PA expressible in Daikon, which we refer to as "expressible recall", and recall of all PA, which we refer to as "total recall".

Figure 7.9 shows the expressible recall and figure 7.10 shows the total recall, for each class and for each test suite size. Tables 7.3 and 7.4 show the averages of the expressible recall and of the total recall over all classes.

While the expressible recall is higher than 0.9 for 6 out of the 15 classes for the medium-size and the large test suites, the total recall exceeds 0.9 only for 1 class for the large test suite. This is also reflected in the averages: 0.86 for the expressible recall and 0.51 for the total recall for the medium test suite. These values show that not all PA are inferred by CITADEL, not even all expressible ones.

It is also interesting to note that for all classes containing programmer-written loop invariants, the expressible recall is 100% for all test suites for these loop invariants. The same stands for the total recall, with the exception of class *FRACTION1*, for which the total recall is 67% for all test suite sizes. So overall the recall for loop invariants is very high, but the low number of programmer-written loop invariants in the code we examined suggests special care in generalizing this result.

IA and PA can also be compared based on the numbers of clauses they contain. In general, the number of relevant IA is much higher than the number of clauses of programmer-written assertions, as illustrated in figure 7.11, which shows a comparison of the number of clauses in PA and the number of relevant IA for the different test suite sizes.

Table 7.5 shows the averages of the relevant IA to PA for all test suite sizes.

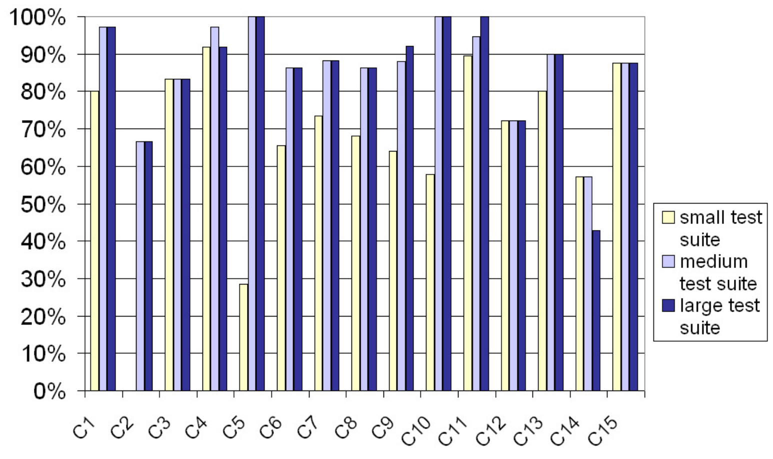

Figure 7.9: Expressible recall: proportion of programmer-written assertions expressible in Daikon that are inferred or implied by the inferred assertions.

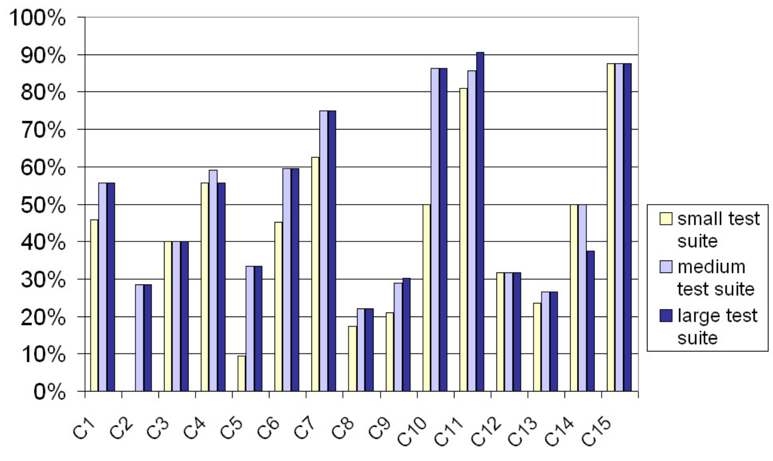

Figure 7.10: Total recall: proportion of all programmer-written assertions that are inferred or implied by the inferred assertions.

Table 7.3: Averages of the expressible recall.

	Small TS	Medium TS	Large TS
Loop invariants	100%	100%	100%
Preconditions	72%	97%	98%
Postconditons & class invariants	68%	81%	80%
Total	67%	86%	86%

Table 7.4: Averages of the total recall.

	Small TS	Medium TS	Large TS
Loop invariants	89%	89%	89%
Preconditions	54%	73%	74%
Postconditions & class invariants	42%	49%	48%
Total	41%	51%	51%

For loop invariants, which programmers hardly ever write, the ratios are very high. Somewhat surprisingly, the ratios are different for preconditions than for postconditions and class invariants: with the small test suite, CITADEL finds fewer relevant preconditions than programmers write; for the medium and large test suites, it finds only marginally more preconditions than written by programmers. This factor is significantly higher for postconditions and class invariants: CITADEL finds about 5 times more relevant assertions in these categories than programmers write.

A possible reason for this striking difference between preconditions on the one side and postconditions and class invariants on the other side has to do with the way in which developers write code: they take care to accurately specify preconditions, because preconditions make the job of implementing routines easier (these conditions can be assumed and must not be checked for in the implementation), while postconditions and class invariants do not have such an effect, so developers tend to neglect them. Another possible explanation has to do with the way in which assertion inference tools work, namely with the dependency between their results and the quality of the test suites they use: while finding the postconditions from given preconditions is easy (it only requires executing the routines), finding the correct preconditions is much harder, as the test cases have to exercise enough calling situations, which requires inter-procedural rather than intra-procedural cover. Thus, it is hard to come up with good test cases for inferring preconditions, and this affects the performance of the tool.

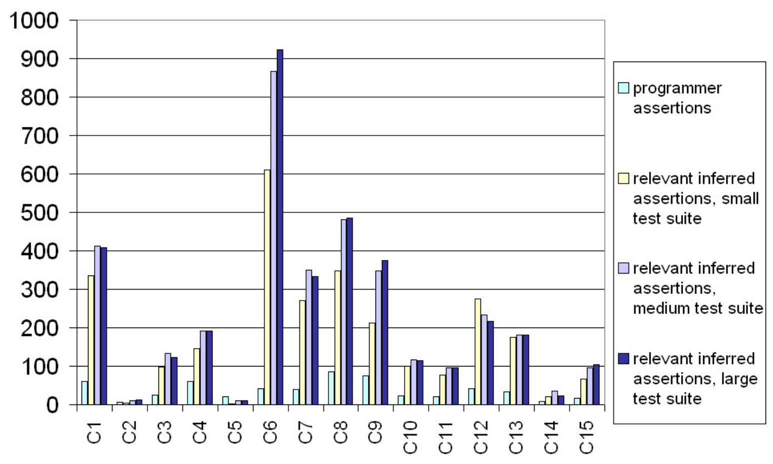

Figure 7.11: Comparison of the number of clauses in programmer-written assertions and the number of relevant assertions inferred for the small, medium-size and large test suite.

Table 7.5: Averages of the ratios of relevant inferred assertions to programmer-written assertions.

	Small TS	Medium TS	Large TS
Loop invariants	12.4	13.3	14.1
Preconditions	0.6	1.4	1.6
Postconditons & class invariants	4.5	5.8	5.4
Total	4.5	5.9	5.8

We also calculated the percent of program points where no PA exist, but for which there are relevant IA. Program points are routine entry and exit points (corresponding to pre- and postconditions), loops (for loop invariants), and one point per class for the class invariant. So the number of program points per class is 2 * number_of_processed_routines + number_of_loops + 1.

Figure 7.12 shows the percent of program points without PA but for which relevant assertions were inferred. (The numbers in this figure include loop invariants. The results excluding loop invariants are not significantly different, because the examined classes contain only very few loops.) This percent varies a lot, from 0% for class *COMPARABLE* to over 50% for the large test suite for class *BASIC_ROUTINES*. The averages of this percent for loop invariants, preconditions, and postconditions and class invariants (shown in table 7.6) show again

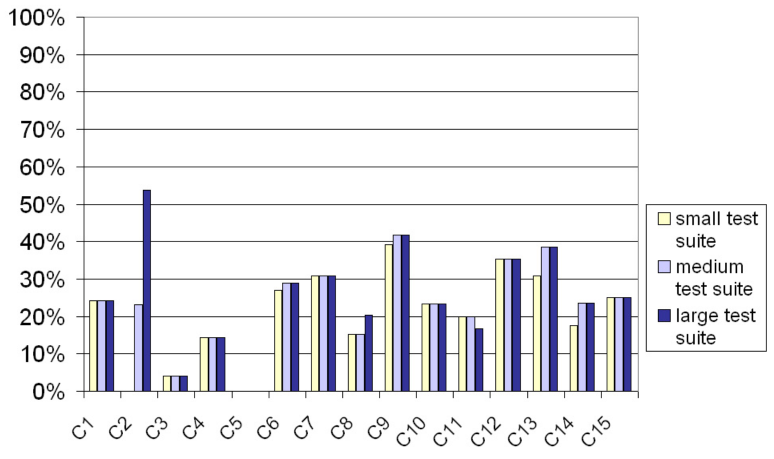

Figure 7.12: Percent of total program points where there are no programmer-written assertions, but relevant assertions could be inferred.

Table 7.6: Averages of the percent of program points with relevant inferred assertions and no programmer-written assertions.

	Small TS	Medium TS	Large TS
Loop invariants	68%	70%	70%
Preconditions	2%	3%	5%
Postconditions & class invariants	28%	33%	35%
Total	20%	23%	25%

that programmers write more preconditions than postconditions and class invariants, and that they write very few loop invariants. Naturally, these results are highly dependent on the number of contracts written by developers, which vary with the class and author of the code, so it is hard to generalize from them. What these results do show, however, is that contract-inference tools can indeed produce relevant assertions for program points for which programmers did not write any assertions.

We also calculated two factors showing how PA and IA complement each other:

- The *strengthening factor of IA over PA* α_1 reflects how much stronger PA become when IA are added and is calculated as the sum of the number of relevant IA and PA, from which the IA implied by PA are subtracted, and the result is divided by the number of PA:

Table 7.7: Averages for the strengthening factor of inferred assertions over programmer-written assertions.

	Small TS	Medium TS	Large TS
Loop invariants	12.6	13.4	14.2
Preconditions	1.1	1.1	1.2
Postconditions & class invariants	5.3	6.1	5.9
Total	5.2	6.1	6.0

$\alpha_1 = \frac{relevantIA + PA - IA\_implied\_by\_PA}{PA}$

- The *strengthening factor of PA over IA* α_2 reflects how much stronger IA become when PA are added and is calculated as the sum of the number of relevant IA and PA, from which the PA implied by IA are subtracted, and the result is divided by the number of relevant IA:
  $\alpha_2 = \frac{PA + relevantIA - PA\_implied\_by\_IA}{relevantIA}$

Values strictly greater than 1.00 for these factors mean that strengthening occurs.

Table 7.7 shows the averages for α_1 for loop invariants, preconditions, postconditions and class invariants, and finally the averages for α_1 for all assertions. Figures 7.13, 7.14 and 7.15 show the values of α_1 for preconditions (figure 7.13), postconditions and class invariants (figure 7.14), and for all inferred assertions, so preconditions, postconditions, class invariants, and loop invariants (figure 7.15). Classes *BASIC_ROUTINES* and *BI_LINKABLE* (abbreviated *C2* and *C14* respectively) do not contain any programmer-written preconditions, hence the gap in the graph for it for the strengthening factor for preconditions. The experiment results thus show that overall IA can strengthen PA, but the strengthening factors for preconditions are generally much lower than those for postconditions and class invariants.

Table 7.8 shows the averages for α_2 for loop invariants, preconditions, postconditions and class invariants, and finally the averages for α_2 for all assertions. Figures 7.16, 7.17 and 7.18 show the values of of α_2 for preconditions (figure 7.16), postconditions and class invariants (figure 7.17), and for all inferred assertions, so preconditions, postconditions, class invariants, and loop invariants (figure 7.18). The experiment results thus show that overall PA can strengthen IA, but to a lesser degree than IA can strengthen PA.

7.3.3 *Discussion*

The experiment results show that inferred assertions can be used to strengthen programmer-written contracts (postconditions and invariants) and that inferred

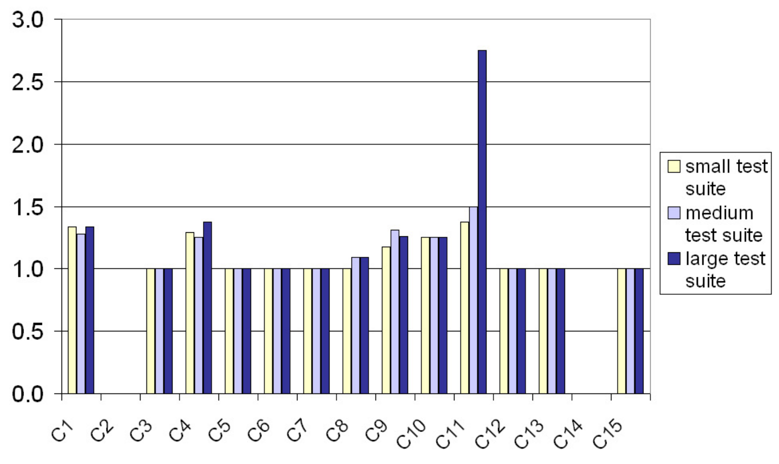

Figure 7.13: Strengthening factor of inferred preconditions over programmer-written preconditions.

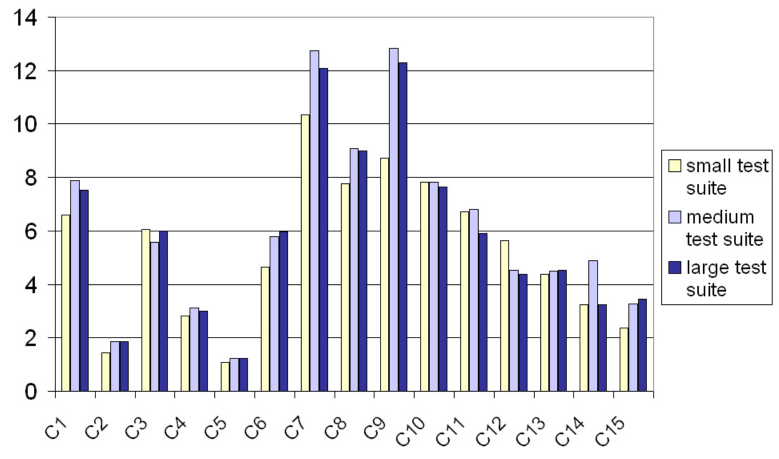

Figure 7.14: Strengthening factor of inferred postconditions and class invariants over programmer-written postconditions and class invariants.

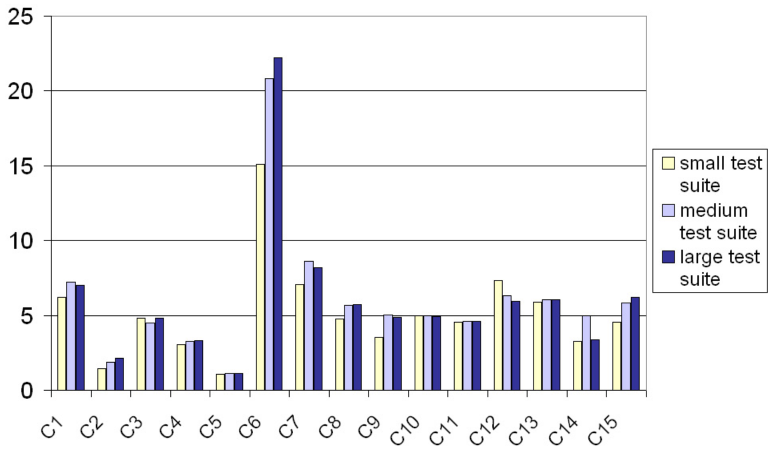

Figure 7.15: Strengthening factor of all inferred assertions over all programmer-written assertions.

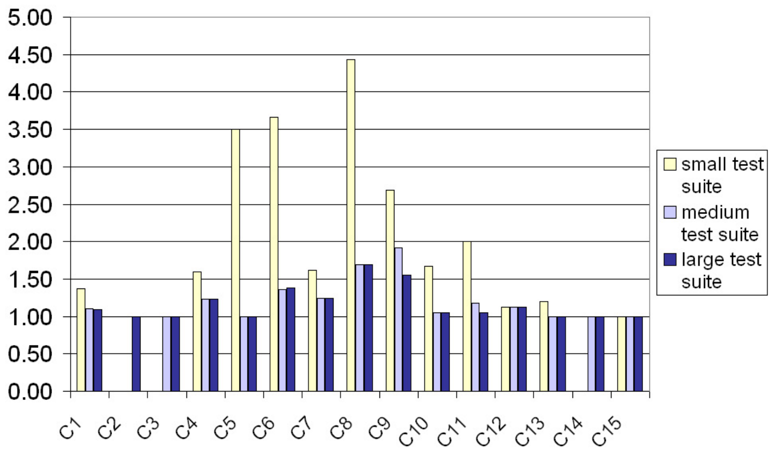

Figure 7.16: Strengthening factor of programmer-written preconditions over inferred preconditions.

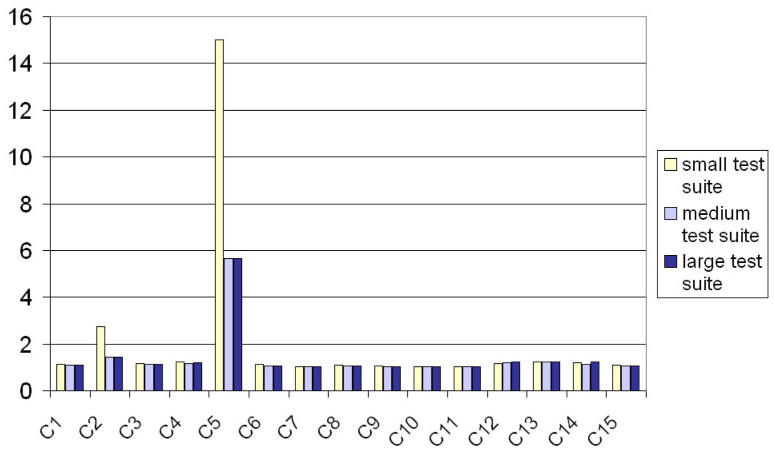

Figure 7.17: Strengthening factor of programmer-written postconditions and class invariants over inferred postconditions and class invariants.

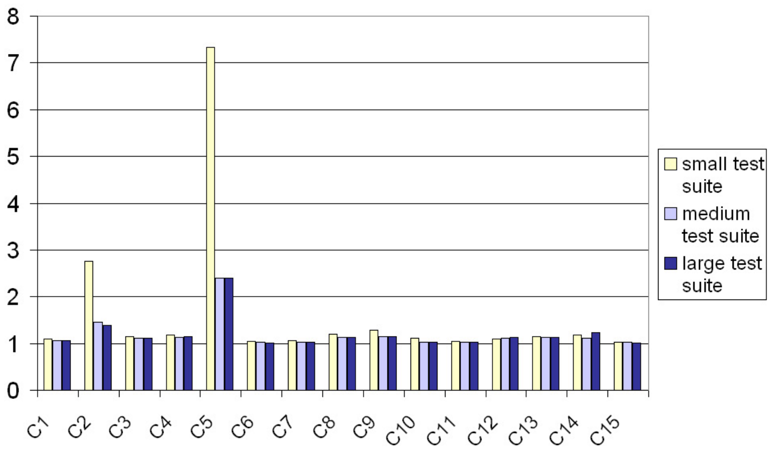

Figure 7.18: Strengthening factor of all programmer-written assertions over all inferred assertions.

Table 7.8: Averages for the strengthening factor of programmer-written assertions over inferred assertions.

	Small TS	Medium TS	Large TS
Loop invariants	1.0	1.0	1.0
Preconditions	2.2	1.2	1.2
Postconditions & class invariants	2.1	1.4	1.4
Total	1.6	1.2	1.2

assertions can be sometimes used to correct existing contracts (strengthening preconditions). Still, not all PA are inferred or implied by the IA and there are about 20% program points where the programmer found nothing to specify and Daikon could infer relevant assertions. So, in short, although IA strengthen PA to a greater extent than vice versa, automated assertion inference does not find a superset of the PA.

There is a further methodological point to be discussed here. In the Design by Contract software development method, the manual process of writing contracts starts already before the implementation work and can expand until after the implementation is finished. Assertion inference tools can only be used when the implementation is ready. Hence, by relying only on such a tool to produce contracts, all the benefits involved in writing contracts already in the software analysis and design phases and during implementation are lost. Thus, even if an assertion inference tool could find all contracts written by programmers, such a tool should not completely replace the manual work of writing contracts, but only be used to strengthen existing contracts when the implementation is finished.

7.3.4 *Correlations*

In trying to establish which, if any, properties of the classes influence the quality of the IA, we examined correlations between class metrics and the quality of assertions inferred for each class. All correlations listed below were computed for the experiment results using the Pearson product-moment correlation coefficient and calculated for the set of assertions inferred for the medium-size test suite, including the generated loop invariants. We define as *strong correlations* those for which the correlation coefficient is between 0.7 and 1.0 for positive correlations and -1.0 and -0.7 for negative correlations. We define as *medium correlations* those for which the correlation coefficient is between 0.5 and 0.7 for positive correlations and -0.7 and -0.5 for negative correlations.

Correctness and relevancy have strong negative correlations to the total number of IA (-0.95 and -0.69 respectively).

Correctness and relevancy of IA also have strong negative correlations to the number of integer zero-argument queries in a class (-0.81 and -0.73 respectively). Integer queries with no arguments increase Daikon's assertion search space significantly, because Daikon has many assertion templates for integer variables, some of these templates involving relations between 2 or 3 variables. The strong positive correlation (0.76) between the number of integer queries with no arguments and the total number of IA also shows this.

The number of immediate routines of a class (routines implemented in the class itself, not inherited) also correlates negatively to the correctness (-0.59) and relevancy (-0.55) of the IA. Since the number of immediate routines of a class determines the number of program points where assertions can be inferred, the number of immediate routines also increases the search space for assertions. The positive correlation (0.62) between the total number of IA and the number of immediate routines of a class also shows this.

Thus, all these results point to the conclusion that the increased search space has a strong negative influence on the correctness and relevancy of the IA.

7.3.5 *Threats to generalizations*

Probably the biggest threat to generalizations of these results is the limited number of classes examined in the experiment. We selected classes written by programmers with various degrees of experience, classes having different semantics and sizes in terms of various code metrics, but naturally their representativeness is limited.

Furthermore, we only unit tested library classes; testing entire applications would likely have produced different results. Examining classes from applications written by other developers is difficult mainly because the semantics of such classes is often clear only to the programmers who originally wrote the code. Without clear understanding of the semantics of a class it is hard to devise test cases for it. In the present study we did not have the opportunity to involve industrial developers.

As shown both by the results of this study and of previous investigations [116, 69], the quality and size of the test suite has a strong influence on the quality of the inferred assertions. We ran the experiment for 3 different test suites for each class, but test suites of other sizes and with other characteristics would have led to different results.

Since we could not discuss the IA with the developers of the classes used in the study, we judged the correctness of the IA based on the implementation and we used a fixed set of rules for determining which IA are interesting and which not, as explained in section 7.3.2. It is likely that the results of the study would have been different if the code developers had performed these classifications.

The results of our study are influenced by the technical characteristics of Daikon and of the Eiffel front-end we developed for it. As such, the results apply only to assertions inferred with this tool combination and are naturally sensitive to the presence of faults in either of the tools.

7.3.6 Conclusions

From the experiment results we can draw the following conclusions:

- A high proportion of inferred assertions are correct (reflect true properties of the source code): around 90% for the medium and large test suites.
- A high proportion of inferred assertions are also relevant (correct and interesting): around 66% for the medium and large test suites.
- Assertion inference can be used to strengthen programmer-written contracts, as shown by a strengthening factor averaging at 6 for the medium and large test suites.
- Assertion inference cannot find all contracts written by programmers; this is proved by an average recall value of 0.5, hence only around half of the programmer-written assertions are inferred or implied by the inferred assertions.
- The quality of inferred assertions decreases with the size of the search space: the more zero-argument integer queries and routines a class contains, the more incorrect and uninteresting assertions are inferred. (Integer queries with no arguments increase Daikon's assertion search space because of the numerous assertion templates involving integer variables used by the tool; the number of routines directly influences the number of program points where assertions can be inferred.)

As a general conclusion, assertion inference cannot completely replace the manual work of writing contracts, nor should it: contracts are a support in software development starting from its very first phases of requirements engineering and analysis, and the task of writing them should not be postponed to the stage when a full implementation of the system is already available. Nevertheless, at this stage assertion inference tools can be used to strengthen the existing programmer-written contracts, resulting in more accurate specification. This comes at a price though: test suites are necessary for exercising the system and programmers still need to sort out the irrelevant assertions (around a third of all generated ones).

Table 7.9: Classes used in the experiment.

Name	**Abbrev.**	**Source**	**LOC**[1]	**LOCC**[2]	**#R**[3]	**#A**[4]	**Pre.**[5]	**Post.**[6]	**Inv.**[7]	**Expr.**[8]
INTEGER_INTERVAL	C1	Base	469	113	26	11	18	34	9	57%
BASIC_ROUTINES	C2	Base	92	7	6	1	0	7	0	42%
BOOLEAN_REF	C3	Base	174	72	12	2	6	16	3	48%
ST_SPLITTER	C4	Gobo	389	61	14	2	24	32	3	60%
COMPARABLE	C5	Base	117	21	7	2	7	13	1	33%
DS_TOPOLOGICAL_SORTER	C6	Gobo	487	42	21	1	11	23	8	69%
TIME	C7	Time	401	41	27	9	14	17	9	85%
GENEALOGY_2	C8	students	1501	132	37	1	55	31	0	25%
GENEALOGY_1	C9	students	874	120	37	1	58	18	0	32%
FRACTION_1	C10	students	166	64	14	3	8	10	1	86%
FRACTION_2	C11	students	156	68	14	3	8	10	1	90%
LINKED_STACK	C12	Base	159	41	7	22	7	8	26	43%
LINKED_QUEUE	C13	Base	202	34	5	22	4	3	27	29%
BI_LINKABLE	C14	Base	138	8	8	3	0	6	2	87%
ST_WORD_WRAPPER	C15	Gobo	186	16	6	2	5	7	4	100%
Average			367.40	56.00	16.07	5.67	15.00	15.67	6.27	59.07%
Minimum			92	7	5	1	0	3	0	25%
Maximum			1501	132	37	22	58	34	27	100%

[1] Lines of code
[2] Lines of contract code
[3] Number of instrumented routines
[4] Number of ancestors
[5] Number of programmer-written precondition clauses
[6] Number of programmer-written postcondition clauses
[7] Number of programmer-written class invariant clauses
[8] Percent of programmer-written contracts expressible in Daikon's grammar

CHAPTER 8

ANALYSIS OF RANDOM-BASED TESTING STRATEGIES

A great variety of testing strategies and tools have been developed in recent years both by the research community and in industry, based on different ideas and assumptions, trying to tackle various issues in various ways. It is thus important to thoroughly evaluate all these approaches' areas of applicability and strengths and weaknesses, because serious advances in software testing require a sound experimental basis to assess the effectiveness of proposed techniques. Such an assessment can be the basis of guidelines for testing practitioners for choosing a testing strategy matching their purposes and constraints.

Such experimental evaluation can focus either on just one testing strategy and investigate its applicability and performance, or on comparing several strategies to find their *relative* strengths and weaknesses. Both types of investigations are necessary.

This chapter presents and discusses the results of experiments we ran to

- Evaluate the performance of random testing
- Compare random testing to ARTOO
- Investigate the kind of faults that manual testing, automated random testing, and user reports reveal

For each experiment, we describe the issues we wanted to investigate, the hypotheses we started from, the setup of the experiment, and the results and their interpretation. We also discuss threats to the validity of generalizations of the results.

8.1 Random contract-based testing

In the software testing literature, the random strategy is often considered to be one of the least effective approaches. The advantages that random testing does present (wide practical applicability, ease of implementation and of understanding, execution speed, lack of bias) are considered to be overcome by its disadvantages. However, what stands behind this claim often seems to be intuition, rather than experimental evidence. A number of years ago Hamlet [75] already pointed out that many of the assumptions behind popular testing strategies, and many ideas that seem intuitively to increase testing effectiveness, have not been backed by experimental correlation with software quality.

We have investigated the *effectiveness of random testing* at finding faults in a significant code base with two distinctive properties: it is extensively used in production applications and it contains a number of faults, which can be found through automated testing. This makes it possible to assess testing strategies objectively, by measuring how many of these faults they find and how fast. More precisely, we set out to answer the following questions:

- How does the number of faults found by random testing evolve over time?
- How much does the version of the random algorithm used influence the results and which such version produces the best results?
- Are more faults found due to contract violations or due to other exceptions?

Here is a summary of the main results:

- The number of found faults has a surprisingly high increase in the first few minutes of testing.
- The version of the random testing algorithm that works best for a class for a testing timeout as small as 2 minutes will also deliver the best results for higher timeouts (such as 30 minutes).
- This version is not the same for all classes, but one can identify a solution that produces optimal results for a specified set of tested classes.
- For testing timeouts longer than 5 minutes, more faults are found through contract violations than through other exceptions.

We have also reported these results in [42].

8.1.1 *Experimental setup*

The experiment was performed using AutoTest and the ISE Eiffel compiler version 5.6 on 32 identical machines, each having a Dual Core Pentium III at 1 GHz and 1 Gb RAM, running Fedora Core 1.

Table 8.1: Properties of the classes under test

Class	LOC	LOCC	#Routines	#Parent classes
STRING	2600	283	175	7
PRIMES	262	52	75	1
BOUNDED_STACK	249	44	66	2
HASH_TABLE	1416	156	135	3
FRACTION1	152	36	44	1
FRACTION2	180	32	45	1
UTILS	54	34	32	1
BANK_ACCOUNT	74	43	35	1

We chose the classes to test in the experiment so that they come from different sources and have varying purposes, sizes, and complexity:

- Classes from EiffelBase 5.6 [8], an industrial-grade library used by virtually all projects written in ISE Eiffel, similar to the System library in Java or C#: *STRING, PRIMES, BOUNDED_STACK, HASH_TABLE.*
- Classes written by students of the Introduction to Programming 2006/2007 course at ETH Zurich for an assignment: *FRACTION1, FRACTION2.*
- Classes created by us exhibiting some common faults found in object-oriented applications: *UTILS, BANK_ACCOUNT.*

The last four classes are available at `http://se.inf.ethz.ch/people/ciupa/test_results`. The others are available as part of the EiffelBase library version 5.6 [8]. The classes from the EiffelBase library and those written by students were not modified in any way for this experiment.

Table 8.1 shows various data about the classes under test: number of lines of code (LOC), number of lines of contract code (LOCC), number of routines (including those inherited), number of parent classes (also those that the class indirectly inherits from).

We ran AutoTest on each class for 30 minutes, for three different seeds for the pseudo-random number generator, for all combinations of the following values for each parameter to the input generation algorithm:

- P_{GenNew} (the probability of creating new objects rather than directly using existing ones) $\in \{0; 0.25; 0.5; 0.75; 1\}$
- P_{Div} (the probability of calling a procedure on an object chosen randomly from the pool after running each test case) $\in \{0; 0.25; 0.5; 0.75; 1\}$

- $P_{GenBasicRand}$ (the probability of generating values for basic types randomly rather than selecting them from a fixed predefined set of values) $\in \{0; 0.25; 0.5; 0.75; 1\}$

These parameters are explained in more detail in section 5.5.

Thus, we ran AutoTest for each of these classes for 30 minutes, for every combination of the 3 seed values, 5 values for P_{GenNew}, 5 values for P_{Div}, and 5 values for $P_{GenBasicRand}$. So there were 3 * 5 * 5 * 5 = 375 30-minute test sessions per class, amounting to a total test time of 90000 minutes or 1500 hours.

We then parsed the saved test logs to get the results for testing for 1, 2, 5, 10, and 30 minutes. (This approach is valid since AutoTest tests routines in the scope in a fair manner, by selecting at each step the routine that has been tested the least up to the current moment. This means that the testing timeout does not influence how AutoTest selects which routine to test at any time.)

Hence, for each combination of class, seed, timeout, and probability values, we get the total number of found faults and the number of these faults which were found due to contract violations and due to other exceptions, respectively. Since there are 5 timeout values and 375 test sessions/class, this produced 5 * 375 = 1875 test session results per class.

Here we only reproduce part of the raw data. The results and conclusions are based on the entire raw data, available at `http://se.inf.ethz.ch/people/ciupa/test_results`.

The criterion we used for evaluating the efficiency of the examined strategies is *the number of faults found in a set time*. Although several other criteria are commonly used to evaluate testing strategies, we consider this criterion to be the most useful, since the main purpose of unit testing is to find faults in the modules under test.

8.1.2 Results and analysis

This section analyzes the results with respect to the questions stated as the goals of the experiment.

How does the number of found faults evolve over time?

To determine how the number of found faults evolves with the elapsed time we look at the highest number of faults (averaged over the three seeds) found for each timeout for every class. For the classes tested in the experiment, the evolution was inversely proportional to the elapsed time: the best fitting that we could find was against a function $f(x) = \frac{a}{x} + b$. Table 8.2 shows, for each class under test, the parameters characterizing the fitting of the evolution of the number of found faults over time against this function with 95% confidence level. The parameters quantifying the goodness of fit are:

- SSE (sum of squared errors): measures the total deviation of the response

Table 8.2: Parameters characterizing the fitting of the evolution of the number of found faults over time against the function $f(x) = a/x + b$.

Class name	a	b	SSE	R-square	RMSE
BANK_ACCOUNT	-1.25	2.58	0.03	0.96	0.10
BOUNDED_STACK	-4.20	11.6	0.11	0.98	0.23
FRACTION1	-0.45	3.14	0.01	0.86	0.10
FRACTION2	-2.19	2.93	0.45	0.86	0.38
HASH_TABLE	-14.48	19.57	9.64	0.93	1.79
PRIMES	-4.8	6.96	0.39	0.97	0.36
STRING	-10.37	11.47	1.24	0.98	0.64
UTILS	-3.16	3.82	0.15	0.97	0.22

values from the fit. A value closer to 0 indicates that the fit will be more useful for prediction.

- R-square: measures how successful the fit is in explaining the variation of the data. It can take values between 0 and 1, with a value closer to 1 indicating that a greater proportion of variance is accounted for by the model.
- RMSE (root mean squared error): an estimate of the standard deviation of the random component in the data. An RMSE value closer to 0 indicates a fit that is more useful for prediction.

Figures 8.1, 8.2, and 8.3 illustrate the results of the curve fitting for classes *STRING*, *PRIMES*, and *HASH_TABLE* respectively. They show both the best fitting curves (as given in Table 8.2) and the actual data obtained in the experiment.

How much do the values of the probabilities influence the number of found faults for every timeout?

Table 8.3 shows the minimum and maximum number of faults found (averaged over the three seeds) for every timeout using all combinations of probability values for each class.

These results show that the *minimum* number of faults found stays constant or increases very little with the increase of the timeout. In other words, for each timeout, there exists at least one combination of probabilities which performs surprisingly badly compared to others. For all classes under test (with the exception of *PRIMES*) a value of 0 for the probability of generating new objects delivered bad results. A value of 1 for the probability of generating basic values randomly had the same effect.

Classes *HASH_TABLE*, *STRING*, and *BOUNDED_STACK* especially show a high difference between the maximum and minimum numbers of faults found for

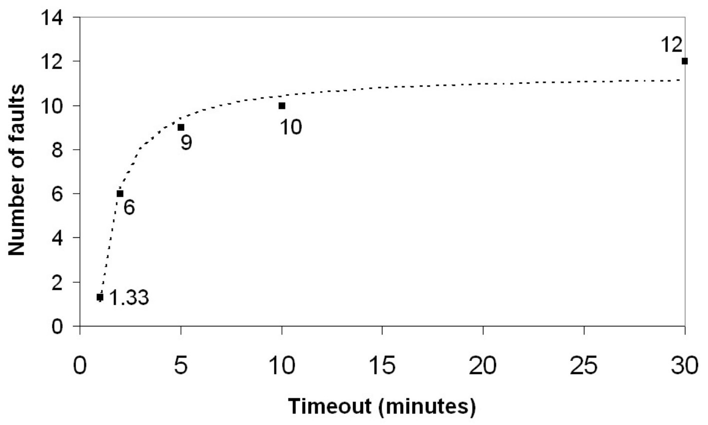

Figure 8.1: Evolution of the number of found faults over time for class STRING

every timeout. This shows that the performance of the random testing algorithm can vary widely with the combination of probabilities that is chosen.

Which version of the random generation algorithm maximizes the number of found faults?

The goal of this analysis is to find the combination of probability values that maximizes the number of found faults, first over all tested classes and over all timeouts, and then individually per class and timeout (1, 2, 5, 10, and 30 minutes). Since the seed influences the results, the results are averaged over the 3 seed values.

The best combination of probabilities averaged over all classes and timeouts is C_0 such that $P_{GenNew0} = 0.25$, $P_{Div0} = 0.5$, and $P_{GenBasicRand0} = 0.25$. With this combination of probabilities, the *average* percent of faults that is lost by timeout and by class compared to the highest number of faults that could be found (by the optimal combination of probability values for that specific class and timeout) is 23%.

If the low timeout values (1 and 2 minutes) are excluded from this calculation, then combination C_0 does not find at most 44% of the faults. This is not significantly different from the best value: 43%.

Another analysis groups results by classes and looks for tendencies over each of the classes. Table 8.4 gives a more detailed view of the results. For each class and for each timeout of 2, 5, 10, and 30 minutes, the table shows values for P_{GenNew} and $P_{GenBasicRand}$ that uncover the highest number of faults. When there is more than one value, several values uncover the same number of faults or

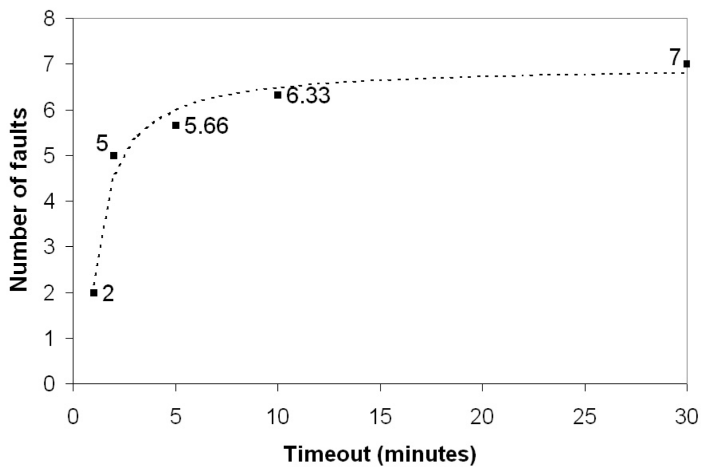

Figure 8.2: Evolution of the number of found faults over time for class PRIMES

the difference between the highest and second highest numbers of faults is very low. The question marks stand for inconclusive results, that is cases where there were several values for the probabilities which uncovered the same maximum number of faults, and there was no clearly predominant value for the probability in question. The probability of diversifying is not shown in the table because clear correlation could not be made between its value and the number of faults. Results for classes *FRACTION1* and *FRACTION2* were also unclear. The issue with these classes was that the total number of faults is small (3) and a minimal variation impacts greatly on the tendency.

The results show that the most effective probability values differ from class to class, but, in most cases, they either change very little or not at all with the timeout for a particular class. In other words, for a certain class, the same combinations provide the best results regardless of the timeout of testing.

According to the results in Table 8.4, a value of 0.25 for P_{GenNew} seems generally to deliver good performance, confirming the result explained above. Exceptions from this rule are classes *BOUNDED_STACK*, *PRIMES*, and *UTILS*. A likely explanation for the different results for the last two classes is that very little of their behavior is dependent on their state, so, if they contain any faults, these faults will manifest themselves on newly created objects too, and not only on objects in a certain state.

Low values for $P_{GenBasicRand}$ (0, 0.25) also seem to deliver the best results in most cases. Again, classes *BOUNDED_STACK* and *PRIMES* have different behavior: in these classes and in class *HASH_TABLE*, the most faults are uncovered for $P_{GenBasicRand}$ = 0.75 or 1. In the case of class *PRIMES*, the most obvious rea-

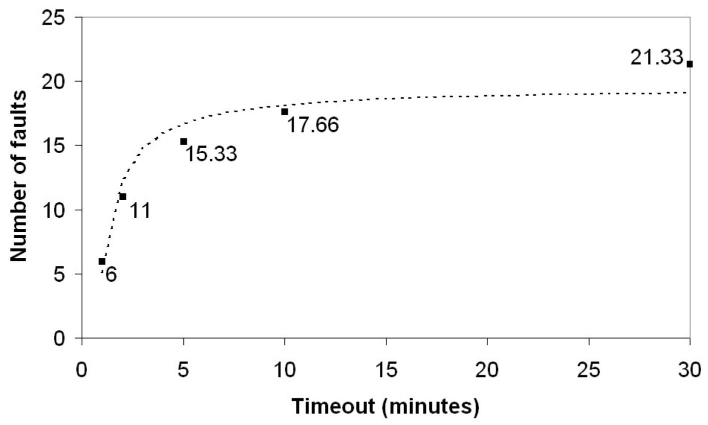

Figure 8.3: Evolution of the number of found faults over time for class HASH_TABLE

son is its very nature: a class implementing the concept of prime numbers is best tested with random values, not with values chosen from a limited set, which have no relation to the characteristics of the class.

As a general conclusion, the combination of factors C_0 gives the best overall result but the process can be fine-tuned depending on the classes that are tested. This fine-tuning is not dependent on the timeout value chosen. However, if time permits, one should run random testing several times with different values for the parameters, because, even though a certain combination of parameters may find fewer faults than another, it may find *different* faults.

Are more faults found due to contract violations or due to other exceptions?

The Design by Contract software development method recommends that the contracts be written at the same time as (or even before) the implementation. Eiffel programmers generally follow this practice, but the contracts that they write are most often weaker than the intended specification of the software and sometimes even contradictory to this intended specification. When contracts are used as oracles in testing, any condition that is not expressed in them cannot be checked, so faults might be missed. For this reason we consider uncaught exceptions also to signal faults. But what contribution does each of these two factors have to the total number of found faults?

Figure 8.4 shows the evolution over time of the number of faults found through contract violations and that of the number of faults found through other exceptions for class *STRING*. The values shown on the graph are obtained by av-

Table 8.3: Minimum and maximum number of faults found for each timeout, averaged over the 3 test runs for each class using different seeds.

Time-out	UTILS Min	UTILS Max	PRIMES Min	PRIMES Max	BANK_ACCOUNT Min	BANK_ACCOUNT Max
1	0.00	0.66	0.33	2.00	0.00	1.33
2	0.00	2.33	3.00	5.00	0.00	2.00
5	0.00	3.00	3.33	5.66	0.00	2.33
10	0.00	3.33	3.33	6.33	0.00	2.33
30	0.00	4.00	3.33	7.00	0.00	2.66
	FRACTION1 Min	FRACTION1 Max	FRACTION2 Min	FRACTION2 Max	HASH_TABLE Min	HASH_TABLE Max
1	0.00	2.66	0.00	1.00	1.00	6.00
2	2.00	3.00	0.00	1.33	1.00	11.00
5	2.00	3.00	0.00	2.33	1.00	15.33
10	2.00	3.00	0.00	3.00	1.00	17.66
30	2.00	3.00	0.00	3.00	1.00	21.33
	STRING Min	STRING Max	BOUNDED_STACK Min	BOUNDED_STACK Max		
1	0.00	1.33	0.66	7.33		
2	0.00	6.00	1.00	9.66		
5	0.00	9.00	1.00	11.00		
10	0.00	10.00	1.00	11.00		
30	0.00	12.00	1.00	11.00		

eraging over the numbers of faults found for every timeout by all versions of the random algorithm. For most other classes this evolution is similar. One concludes that over time the proportion of faults found through contract violations becomes much higher than that of faults found through other thrown exceptions. For short timeouts (1 or 2 minutes) the situation is reversed.

Extreme cases are those of classes *BOUNDED_STACK*, *BANK_ACCOUNT*, and *PRIMES*. Figure 8.5 shows the evolution over time of the number of faults found through contract violations and that of the number of faults found through other exceptions being thrown for class *BOUNDED_STACK*. One notices that, regardless of the timeout, the number of faults found by contract violations is always higher than the number of faults found through other exceptions. Furthermore, this latter number increases only slightly from timeout 1 to timeout 2, and then does not increase at all. For classes *BANK_ACCOUNT* and *PRIMES*, all versions of the random generation algorithm constantly find more or an equal number of faults through contract violations than through other exceptions.

Table 8.4: Probability values that maximize the number of found faults for each timeout

Class name	P_{GenNew}				$P_{BenBasicRand}$			
	2	5	10	30	2	5	10	30
BANK_ACCOUNT	0.25	0.25	0.25	0.25	?	0	0.25	0
				0.5		0.25		0.25
BOUNDED_STACK	0.5	0.75	0.75	0.75	0.5	0.75	0.75	0.75
	0.75							
HASH_TABLE	0.25	0.25	0.25	0.25	>0	0.75	0.5	0.5
	0.5	0.5	0.5	0.5			0.75	0.75
PRIMES	1	1	1	1	0.75	0.75	0.75	1
					1			
STRING	0.25	0.25	0.25	0.25	0.75	0.25	0	0
UTILS	?	0.75	0.75	0.5	0	0	0	0
				0.75				0.25

8.1.3 *Threats to the validity of generalizations of the results*

As is the case for any experimental study, the conclusiveness of the results depends on the representativeness of the samples examined. Testing a higher number of classes would naturally have increased the reliability of the results. The very high number of tests that we had to run for each class in order to explore all possible combinations of parameters (the probabilities and the seed) and the duration of each such test (30 minutes) made it impossible for us to test more classes in the time during which we had exclusive access to the hardware necessary for the experiment. Hence, we chose the classes that we tested so that they come from different sources, implement different concepts and functionality, were produced by programmers with different levels of expertise, and have different sizes. Despite this, these classes do not exhibit all types of faults that can be found in object-oriented software; hence, it is likely that, for some of these faults, the behavior of a random testing algorithm would be different.

In any random-based testing approach, the seed used to initialize the pseudo-random number generator introduces some variability in the results. In this experiment we tested each class for each combination of probabilities for 3 seeds. Using a higher number of seeds would have improved the reliability of the results, but practical time constraints prevented this.

Furthermore, the results reported here apply only to the random testing algorithm implemented in AutoTest. Other strategies for randomly generating inputs for testing O-O applications are also possible, and the behavior of such strategies would likely be different.

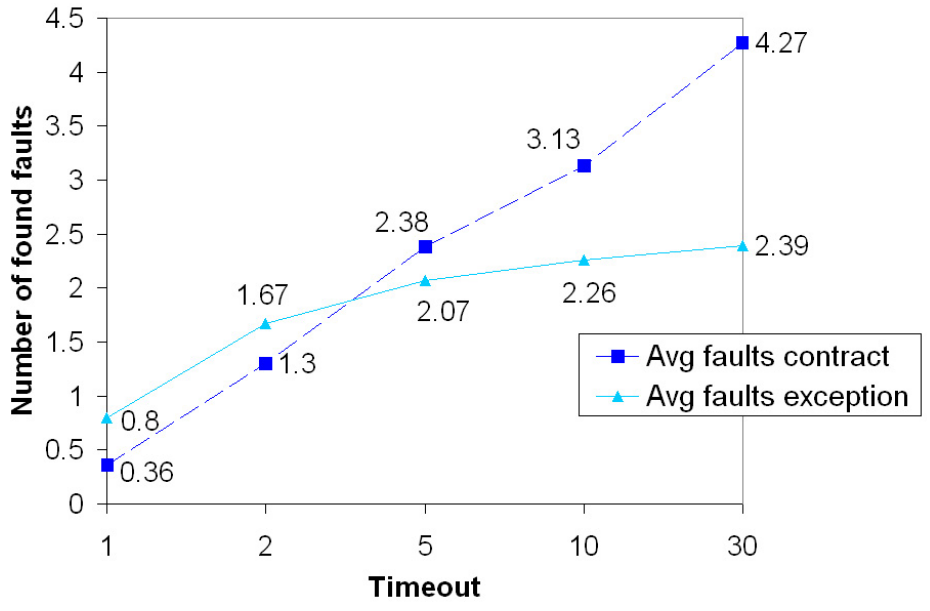

Figure 8.4: Evolution over time of the number of faults found through contract violations and through other exceptions for class STRING

In AutoTest, the notions of faults and failures are largely determined by the contract-based oracle. In general, a *failure* is an observed difference between actual and intended behaviors. In AutoTest we interpret every contract violation (except for immediate precondition violations) as a failure. However, programmers are interested in *faults* in the software: wrong pieces of code that trigger the failures, and the same fault may trigger arbitrarily many failures. Hence, an analysis of random testing should consider the detected faults, not the failures. Mapping failures to faults is part of the debugging process and is usually done by humans. This was not feasible for the experiment reported here. Instead we rely on an approximation that groups failures based on the following assumption: two failures are a consequence of the same fault if and only if they manifest themselves through the same type of exception, being thrown from the same routine and the same class. This automatic mapping from failures to faults could have led to faults being missed.

8.1.4 Conclusions

This study answered several questions about the performance of random testing in general and about the factors that influence it. It has shown that, despite its simplicity and unguided nature, random testing does indeed find faults, not only

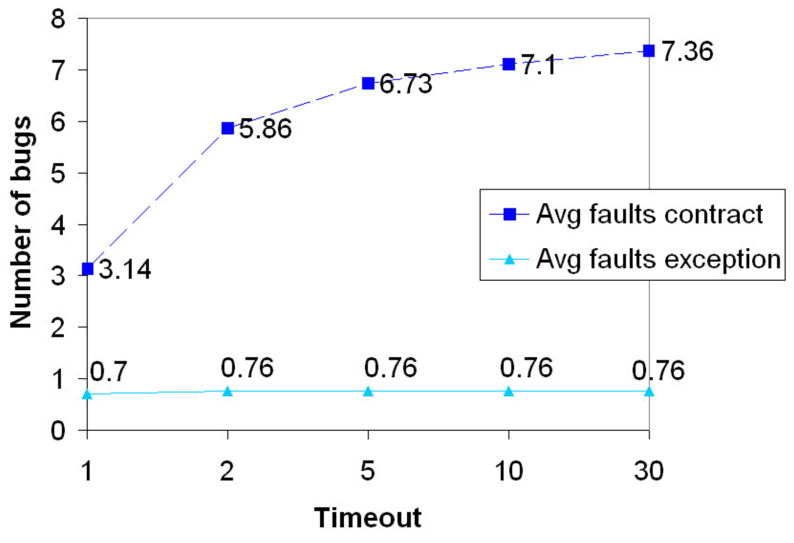

Figure 8.5: Evolution over time of the number of faults found through contract violations and through other exceptions for class BOUNDED_STACK

seeded ones, but also faults present in widely used, industrial-grade code. Of particular importance is the observation that random testing finds a very high number of faults in the first few minutes of testing a certain class. This indicates that, although this strategy might not find all faults present in the code, its rate of finding faults over short timeouts makes it a good candidate for combining with other testing strategies, more expensive in terms of the computational resources they require.

8.2 How predictable is random testing?

The experiments described in the previous section exhibited a seemingly high variation of the number of faults detected over time. From an engineer's point of view, a high variance means low predictability of the process – which immediately reduces its value. One might argue that random testing can be performed overnight and when spare processor cycles are available; the sheer amount of continuous testing would then compensate for any potential variance. However, arbitrary computation resources may not be available, and insights into the efficiency of a testing strategy are useful from the management perspective: such numbers make it comparable to other strategies.

Problem We set out to answer the following questions. How predictable is random testing? What are the consequences? In particular, how would this technique be best used?

More concretely, the questions that we study include the following:

1. What percentage of defects can we expect to detect after n minutes of random testing? (This percentage will be computed with respect to the overall number of defects that we found when performing random tests.) What is the variance, and thus the predictability, of the associated random variable?
2. Does running a long uninterrupted testing session uncover more defects than running several short sessions with different seeds?
3. How much time can we expect to spend before the first defect is found? How predictable is random testing with respect to this?

While the effectiveness of random testing has been studied before, we do not know of any other investigations of the *predictability* of the process.

Solution Using AutoTest, we generated and ran random tests for 27 classes from EiffelBase. Each class was tested for 90 minutes. To assess the predictability of the process, we repeated the testing process for each class 30 times with different seeds for the pseudo-random number generator. This results in an overall testing time of 1215 hours with more than 6 million triggered failures. The main results are the following.

1. When averaging over all 27 classes, 30% of the overall number of faults detected during the experiments are found after 10 minutes. After 90 minutes, on average, an additional 8 percent points of the overall number of randomly detected faults are found.
2. In terms of the relative number of detected faults (relative to the overall number of faults detected via random testing), random testing is highly predictable, as measured by a low standard deviation.
3. Different runs of the testing process reveal different faults.
4. For 24 out of the 25 classes in which faults were found, at least one out of the 30 experiments detected a fault within the first second. This can be read as follows: random testing very likely detects a fault within at most 30 seconds.

We have reported on these results in [45]. A package including the results and the source code of AutoTest is available online.[1] It contains everything needed for the replication and extension of our experiments.

[1] `http://se.inf.ethz.ch/people/ciupa/public/random_oo_testing_experiment.zip`

8.2.1 Experimental setup

In the experiments, each of 27 classes was tested in 30 sessions of 90 minutes each, where in each session a different seed was used to initialize the pseudo-random number generator used for input creation. The total testing time was thus $30*27*90$ minutes $= 50.6$ days. All the tested classes were taken unmodified from the EiffelBase library version 5.6. The tested classes include widely used classes like *STRING* or *ARRAY* and also more seldom used classes such as *FIBONACCI* or *FORMAT_DOUBLE*. Table 8.5 shows various code metrics of the classes under test: lines of code (LOC), number of routines, attributes, contract clauses (LOCC) and total number of faults found in the experiments. The number of routines, attributes and contracts includes the part that the class inherits from ancestors, if any.

We set the AutoTest parameters to the values shown to deliver the best results: $P_{GenNew} = 0.25$ and $P_{GenBasicRand} = 0.25$ and did not use diversification. Hence, the strategy for creating inputs that we used is not purely random: it is random combined with limit testing (because we only use truly random basic values in 25% of the cases and in the rest 75% we select basic values from predefined sets). The experiments reported in the previous section showed that this strategy is much more effective at uncovering faults than purely random testing at no extra cost in terms of execution time, so we consider it more relevant to investigate the more effective strategy. As this strategy is still random-based but also uses special predefined values (which influence the results), we refer to it from now on as *random*$^{+}$ *testing*.

During the testing sessions, AutoTest may trigger failures in the class under test and also in classes on which the tested class depends. There are two ways in which failures can be triggered in other classes than the one currently under test. First, a routine of the class under test calls a routine of another class, and the latter contains a fault which affects the caller. Second, the constructor of another class, of which an instance is needed as argument to a routine under test, contains a fault.

AutoTest reports faults from the first category as faults in the class under test. This is because, although the routine under test is not responsible for the failure, this routine cannot function correctly due to a faulty supplier and any user of the class under test should be warned of this. Faults from the second category, however, are not counted. This is because in these experiments we focus on faults found in the class under test only. Such tests are nevertheless also likely to reveal faults (according to the related analysis on the benefits of "interwoven" contracts [95]). How many of them are found there and how this impacts the predictability of random testing is a subject of further studies.

Computing infrastructure The experiments ran on 10 dedicated PCs equipped with Pentium 4 at 3.2GHz, 1Gb of RAM, running Linux Red Hat En-

Table 8.5: Metrics of the tested classes

Class name	LOC	#Routines	#Att.	#LOCC	#Faults
ACTION_SEQUENCE	382	156	16	164	71
ACTIVE_INTEGER_INTERVAL	141	77	6	75	43
ACTIVE_LIST	115	145	9	140	76
ARRAY	694	86	4	98	74
ARRAYED_LIST	684	139	6	146	74
ARRAYED_SET	95	152	6	155	27
BOUNDED_QUEUE	335	63	5	53	22
CHARACTER_REF	450	77	1	60	0
CLASS_NAME_TRANSLATIONS	206	128	13	141	39
FIBONACCI	166	70	2	55	8
FIXED_LIST	355	124	6	123	38
FORMAT_DOUBLE	304	111	12	116	8
FORMAT_INTEGER	599	83	8	93	5
HASH_TABLE	1416	122	13	178	49
HEAP_PRIORITY_QUEUE	367	96	5	107	27
INTEGER_INTERVAL	484	75	5	87	44
INTEGER_REF	618	99	1	106	1
LINKED_CIRCULAR	401	129	4	88	59
LINKED_LIST	719	106	6	92	25
LINKED_TREE	366	152	9	128	71
RANDOM	256	98	5	65	10
STRING	2600	171	4	296	32
STRING_SEARCHER	304	37	2	64	0
TWO_WAY_CHAIN_ITERATOR	120	71	2	98	36
TWO_WAY_CIRCULAR	62	129	4	70	94
TWO_WAY_CURSOR_TREE	170	117	7	107	87
TWO_WAY_LIST	488	113	8	94	47
Average	477.7	108.4	6.3	111	39.5
Median	366	111	6	98	38
Minimum	62	37	1	53	0
Maximum	2600	171	16	296	94

terprise 4 and ISE Eiffel 5.6. The AutoTest session was the only CPU intensive program running at any time.

8.2.2 *Results and analysis*

Over all 27 classes, the number of detected faults ranges from 0 to 94, with a median of 38 and a standard deviation of 28. In two of the classes (*CHARACTER_REF* and *STRING_SEARCHER*) the experiments did not uncover any faults. Figure 8.6 shows the median absolute number of faults detected over time for each class.

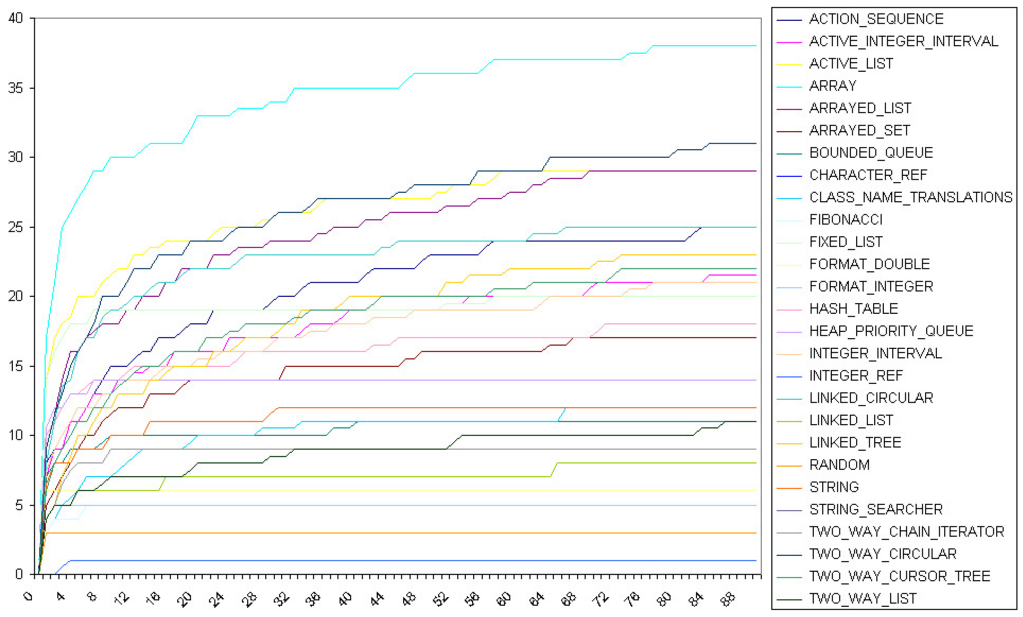

Figure 8.6: Medians of the absolute numbers of faults found in each class

In order to get aggregated results, we look at the *normalized* number of faults over time. For each class, we normalize by dividing the number of faults found by each test run by the total number of faults found for this particular class. The result is shown in Figure 8.7. When averaging over all 27 classes, 30% of the overall number of faults detected during our experiments are found after 10 minutes, as witnessed by the median of medians reaching .3 after 10 minutes in Figure 8.7. After 90 minutes, on average, an additional 8 percent of the overall number of randomly detected faults are found.

The main question is: **how predictable is random$^+$ testing?** We consider two kinds of predictability: one that relates to the number of faults, and one that relates to the kind of faults and that essentially investigates if we are likely to detect

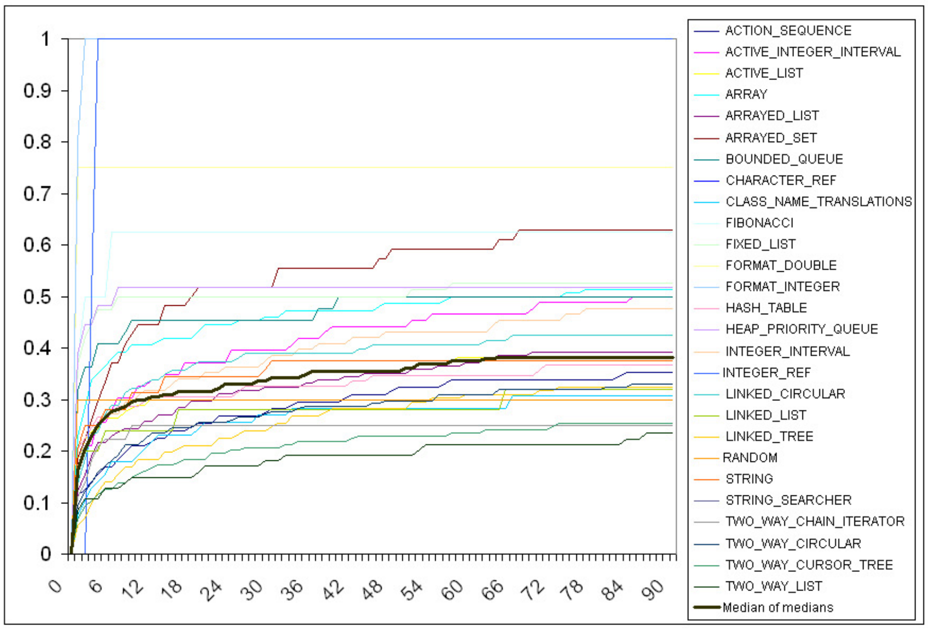

Figure 8.7: Medians of the normalized numbers of faults found in each class; their median. On average 30% of the faults are found in the first 10 minutes. Testing for 80 additional minutes uncovers a further 8% of faults.

the same faults, regardless of which of the thirty experiments is chosen. This provides insight into the influence of randomness (or, in more technical terms, the influence of the seed that initializes the pseudo-random number generator). Furthermore, we also consider how long it takes to detect a first fault and how predictable random$^+$ testing is with respect to this duration.

In terms of *predictability of the number of detected distinct faults*, we provide an answer by considering the standard deviations of the normalized number of faults detected over time (Figure 8.8). With the exception of *INTEGER_REF*, an outlier that we do not show in the figure, the standard deviations lie roughly in-between 0.02 and 0.06, corresponding to 2% to 6% of the relative number of errors detected. Because we want to get one aggregated result across all classes, we display the median and the standard deviation of the standard deviations of the normalized number of detected faults in the same figure (median of standard deviations: upper thick line; standard deviation of standard deviations: lower thick line in Figure 8.8). The median of the standard deviations of the normalized numbers of detected faults decreases from 4% to 2% in the first 15 minutes and then remains constant. Similarly, the standard deviation of the standard deviations of the normalized number of detected faults linearly decreases from 3% to 1.5% after 10 minutes, and then remains approximately constant.

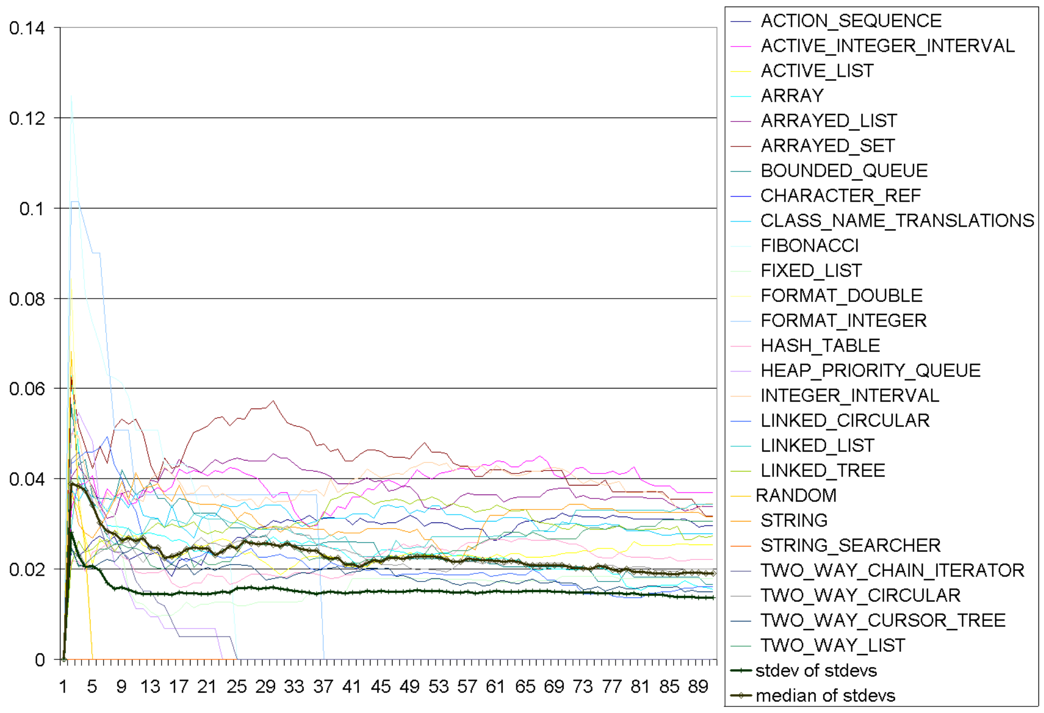

Figure 8.8: Standard deviations of the normalized numbers of faults found in each class; their median and standard deviation. This median and standard deviation being rather small suggests that random$^+$ testing is, in terms of the relative number of detected faults, rather predictable in the first 15 minutes, and strongly predictable after 15 minutes.

The median and standard deviation of the standard deviations being rather small suggests that random$^+$ testing is, *in terms of the relative number of detected faults*, rather predictable in the first 15 minutes, and strongly predictable after 15 minutes. In sum, this somewhat counter-intuitively suggests that *in terms of the relative number of detected faults, random$^+$ testing OO programs is indeed predictable*.

An identical relative number of faults does not necessarily indicate that the *same* faults are detected. If all runs detected approximately the same errors, then we could expect the normalized numbers of detected faults to be close to 1 after 90 minutes. This is not the case (median 38%) in our experiments: random$^+$ testing exhibits a high variance in terms of the actual detected failures, and thus appears *rather unpredictable in terms of the actual detected faults*.

In order to investigate more closely what are the overlappings and differences, in terms of the actual faults found, between different runs of the test generator for each class, we looked at the number of experiments that revealed each fault. Each fault can be found 1 to 30 times, that is, it can be found only in 1 or in several and up to all 30 experiments (for a particular class). Figure 8.9 shows how many faults were found 1 to 30 times and it reveals an interesting tendency of random$^+$ testing: 25% of all faults are uncovered by only 1 or 2 out of 30 experiments, 19% of all faults are uncovered in all 30 experiments.

The sharp increase in the number of faults found in all 30 experiments compared to those found in 23 to 29 experiments may be surprising at first, but is explainable through the nature of the testing process that we performed: random testing combined with special value testing. There are certain faults that AutoTest finds with high probability because it tries the inputs that trigger them (Void, minimum/maximum integer) with high probability. It is hence more likely that AutoTest finds, for example, a Void-related fault in all 30 experiments than that it finds it in 23 to 29 experiments. Of course, occasionally this still happens, hence the relatively small numbers of faults found in 23 to 29 experiments.

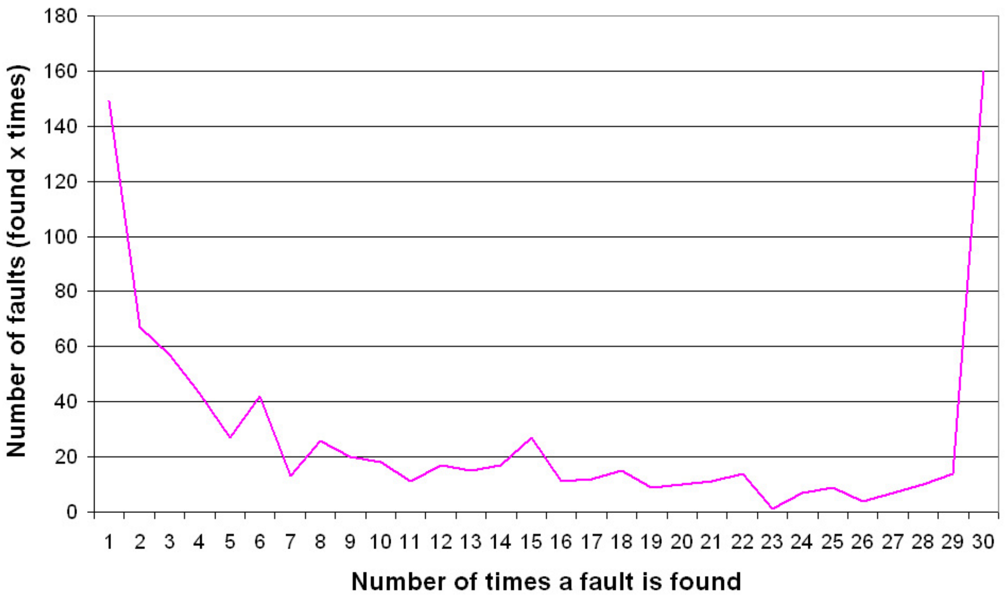

Figure 8.9: Number of faults found in 1, 2, ..., all 30 experiments for every class. 25% of all faults are uncovered by only 1 or 2 out of 30 experiments, 19% of all faults are uncovered in all 30 experiments.

Finally, when analyzing the results, we were surprised to see that *for 24 out of the 25 classes in which we found faults, at least one experiment detected a fault in the first second.* Taking a slightly different perspective, we could hence test any class thirty times, one second each. This means that within our experimental setup, random$^+$ testing *is almost certain to detect a fault for any class within 30 seconds.* In itself, this is a rather strong predictability result.

This unexpected finding led us to investigate a question that we originally did not set out to answer: in terms of the efficiency of our technology, is there a difference between long and short tests? In other words, does it make a difference if we test one class once for ninety minutes or thirty times for three minutes?

To answer this question, we analyzed how the number of faults detected when testing for 30 minutes and changing the seed every minute compares to the number of faults found when testing for 90 minutes and changing the seed every 3 minutes and to the number of faults found when testing for 90 minutes without changing the seed, with longer test runs being an approximation of longer test sequences. Changing the seed also means restarting the testing session, in particular emptying the object pool. Figure 8.10 shows the results of a class-by-class comparison.

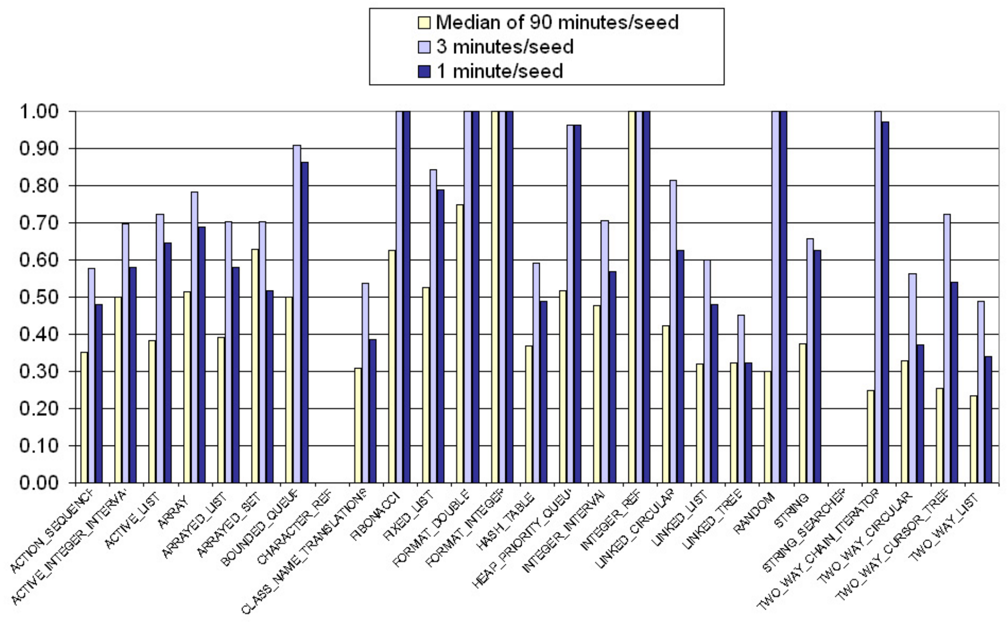

Figure 8.10: Cumulated normalized numbers of faults after 30*3 and 30*1 minutes; median normalized number of faults after 90 minutes. Collating 30*3 minutes of test yields considerably better results than testing for 90 minutes.

The results indicate that the strategy using each of the 30 seeds for 3 minutes (90 minutes altogether) detects more faults than using each of the thirty seeds for 1 minute (30 minutes altogether). Because the testing time is three times larger in the former when compared to the latter case, this is not surprising. Note, however, that the normalized number of faults is not three times higher. On more comparable grounds (90 minutes testing time each), *collating thirty times 3 minutes of test yields considerably better results than testing for 90 minutes.*

This suggests that short tests are more effective than longer tests. However, a more detailed analysis reveals that this conclusion is too simple. For the technical reasons described in section 5.4, the interpreter needs to be restarted at least once during most of the experiments. In fact, there were only 60 experiments during which no interpreter restart occurred. Such a restart is similar to beginning a new experiment, in that the object pool is emptied and must be constructed anew. Interpreter restarts do not, however, affect the scheduling of calls to routines under test: AutoTest preserves the fairness criteria and tries to call first routines that were tested the least up to that point. Because of these interpreter restarts, we cannot directly hypothesize on the length of test cases. In fact, we do not have any explanation of this stunning result yet, and its study is the subject of ongoing and future work.

8.2.3 *Threats to the validity of generalizations of the results*

The classes used in the experiment belong to the most widely used Eiffel library and were not modified in any way. They are diverse both in terms of various code metrics and of intended semantics, but naturally their representativeness of OO software is limited.

A further threat to the validity of this empirical evaluation is a technical detail in the implementation of the used tool, AutoTest. As explained in section 5.4, AutoTest uses a two-process model: a driver process is responsible for orchestrating the test generation and execution process, while an interpreter process receives simple commands from the driver (such as object creation, routine call, value assignment) and can execute them and output the result. Thus, the interpreter carries out the actual test execution. If any failures occur during test execution from which the interpreter cannot recover, the driver will shut it down and restart it. Such restarts have the consequence that the object pool is emptied, hence subsequent calls to routines under test will start from an empty pool and build it anew. Interpreter restarts do not cause the routine priorities to be reset, hence the fairness criterion is fulfilled. The emptying of the pool, however, puts a limit to the degree of complexity that the test inputs can reach. In our experiments, interpreter restarts occurred at intervals between less than a minute and over an hour. Even for the same class, these restarts occur at widely varying intervals, so that some sessions reach presumably rather complex object structures, and others only very simple ones.

AutoTest implements one of several possible algorithms for randomly generating inputs for OO programs. Although we tried to keep the algorithm as general as possible through various parameters, there exist other methods for generating objects randomly, as explained in section 4.3.2. As such, the results of this study apply only to the specific algorithm together with specific choices for technical parameters (e.g., P_{GenNew})) for random^{+} testing implemented in AutoTest.

The full automation of the testing process – necessary also due to the sheer number of tests generated and executed in the experiment – required an automated oracle: contracts and exceptions. This means that naturally any fault which does not manifest itself through a contract violation or another exception could not be detected and included in the results presented here.

Mapping failures to faults manually was not feasible for the over 6 million failures triggered in this experiment. Instead we used the same approximation as in the experiments reported in the previous section: two failures are a consequence of the same fault if they manifest themselves through the same type of exception, being thrown from the same routine and the same class. This is naturally only an approximation and may have led to faults being missed.

The experiments reported here were performed only on classes "in isolation," not on a library as a whole or on an entire application. This means that the routines of these classes were the only ones tested directly. The routines that they transitively call are also implicitly tested. The results would probably be different for a wider testing scope, but timing constraints did not allow testing of entire applications and libraries.

As explained above, for 24 out of the 25 classes in which our experiments uncovered faults, there was at least one experiment in which the first fault for a class was found within the first second of testing. The vast majority of these faults are found in constructors when using either an extreme value for an integer or Void for a reference-type argument. It is thus questionable if these results generalize to classes which do not exhibit one of these types of faults. However, as stated above, in our experiment 24 out of the 27 tested classes did contain a maximum integer or Void-related fault.

8.2.4 Conclusions

Random$^+$ testing, by its very nature, is subject to random influences. Intuitively, choosing different seeds for different generation runs should lead to different results in terms of detected defects. Earlier studies had given initial evidence for this intuition. We set out to do a systematic study on the predictability of random$^+$ tests. To the best of our knowledge (and somewhat surprisingly), this question has not been studied before.

In sum, the main results are the following. Random$^+$ testing is predictable in terms of the relative number of defects detected over time. In our experiments, random$^+$ testing detects a defect within 30 seconds, if a fault exists in the class under test. On the other hand, random$^+$ testing is much less predictable in terms of the actual defects that are detected.

8.3 Random testing vs. ARTOO

8.3.1 *Experimental setup*

The subjects used in the evaluation of ARTOO are classes from the EiffelBase library [8] version 5.6. No changes were made to this library for the purposes of this experiment: the classes tested are taken from the released, publicly-available version of the library and all faults mentioned are real faults, present in the 5.6 release of the library.

Table 8.6 presents some properties of the classes under test: number of lines of code (LOC), number of lines of contract code (LOCC), and number of routines, of attributes and of parent classes. All the metrics except the last column refer to the flat form of the classes, that is a form of the class text that includes all the features of the class at the same level, regardless of whether they are inherited or introduced in the class itself.

Class	LOC	LOCC	#Routines	#Attributes	#Parents
ACTION_SEQUENCE	2477	164	156	16	24
ARRAY	1208	98	86	4	11
ARRAYED_LIST	2164	146	39	6	23
BOUNDED_STACK	779	56	62	4	10
FIXED_TREE	1623	82	125	6	4
HASH_TABLE	1791	178	122	13	9
LINKED_LIST	1893	92	106	6	19
STRING	2980	296	171	4	16

Table 8.6: Properties of the classes under test

All tests were run using the ISE Eiffel compiler version 5.6 on a machine having a Pentium M 2.13 GHz processor, 2 GB of RAM, and running Windows XP SP2. The tests applied both the basic random strategy of AutoTest (called RAND for brevity below) and ARTOO, testing one class at a time. Since the seed of the pseudo-random number generator influences the results, the results presented below are averaged out over 5 10-minute tests of each class using different seeds.

It is important to note that the testing strategy against which we compare ARTOO is in fact not purely random, as explained in Section 5.5: values for primitive types are not selected randomly from the set of all possible values, but from a restricted, predefined set of values considered to be more likely to uncover faults. We chose this strategy as the basis for comparison because the experiments reported in section 8.1 have shown it to be more efficient than purely random testing.

The results were evaluated according to two factors: *number of tests to first fault* and *time to first fault*. Other criteria for the evaluation are also possible. Measuring the time elapsed and the number of test cases run until the first fault is found is driven by practical considerations: software projects in industry usually run under tight schedules, so the efficiency of any testing tool plays a key role in its success.

8.3.2 Results

Tables 8.7 and 8.8 show a routine-by-routine comparison of the tests to first fault and time to first fault respectively for both ARTOO and RAND applied to classes *ARRAYED_LIST* and *ACTION_SEQUENCE*. The tables show, for each routine in which both strategies found faults, the number of tests and time required by each strategy to find a fault in that particular routine. Both the tests and the time are averages, over the five seeds, of the time elapsed since the beginning of the testing session and the decimal part is omitted. All calls to routines and creation procedures of the class under test are counted as test cases for that class. The tables also show, for each of these two factors, the ratios between the performance of ARTOO and that of RAND rounded to two decimal digits, showing in bold the cases for which ARTOO performed better.

At coarser granularity, Table 8.9 shows for every class under test the average (over all routines where both strategies found at least one fault) of the number of tests to first fault and time to first fault for each strategy and the proportions ARTOO/RAND, rounded to two decimal digits. Figures 8.11 and 8.12 show the same information, comparing for every class the number of tests to first fault and the time to first fault, respectively.

In most cases ARTOO reduces the number of tests necessary to find a fault by a considerable amount, sometimes even by two orders of magnitude. However, calculating the object distances is time-consuming. The overhead ARTOO introduces for selecting which objects to use in tests (the distance calculations, the serializations of objects, etc.) causes it to run fewer tests over the same time than RAND. For the tested classes, which all have fast-executing routines, although ARTOO needs to run fewer tests to find faults, RAND needs less time.

The experiments also show that there are cases in which ARTOO finds faults which RAND does not find (over the same test duration). Table 8.10 lists the classes and routines where only ARTOO was able to find some faults, the number of tests and the time that ARTOO needed to find the first fault, and the number of faults it found in each routine.

Class	Routine	Tests to first fault		
		ARTOO	RAND	$\frac{ARTOO}{RAND}$
ARRAYED_ LIST	append	**432**	**5517**	**0.08**
	do_all	**296**	**737**	**0.40**
	do_if	**16**	**1258**	**0.01**
	fill	**159**	**7130**	**0.02**
	for_all	**303**	**517**	**0.59**
	is_inserted	**31**	**126**	**0.25**
	make	23	3	7.44
	make_filled	**13**	**117**	**0.11**
	prune_all	**51**	**10798**	**0.00**
	put	96	89	1.08
	put_left	**146**	**9739**	**0.01**
	put_right	**278**	**8222**	**0.03**
	resize	**355**	**1143**	**0.31**
	there_exists	**307**	**518**	**0.59**
	wipe_out	**594**	**3848**	**0.15**
ACTION_ SEQUENCE	arrayed_list_make	**748**	**6800**	**0.11**
	call	**109**	**2382**	**0.05**
	duplicate	**378**	**410**	**0.92**
	for_all	**286**	**623**	**0.46**
	is_inserted	115	95	1.21
	make_filled	**183**	**449**	**0.41**
	put	81	67	1.21
	remove_right	**448**	**17892**	**0.03**
	resize	**399**	**5351**	**0.07**
	set_source_connection_agent	**265**	**3771**	**0.07**
	there_exists	215	104	2.07

Table 8.7: Results for two of the tested classes, showing the number of tests required by ARTOO and RAND to uncover the first fault in each routine in which they both found at least one fault, and their relative performance. In most cases ARTOO requires significantly less tests to find a fault.

RAND also finds faults which ARTOO does not find in the same time. This suggests that the two strategies have different strengths and, in a fully automated testing process, should ideally be used in combination. As shown in section 8.1, experimental data indicates that the evolution of the number of new faults that RAND finds is inversely proportional to the elapsed time. This means in particular that after running random tests for a certain time, it becomes unlikely to uncover new faults. At this point ARTOO can be used to uncover any remaining faults. In cases where the execution time of the routines under test is high, AR-

Class	Routine	Time to first fault (seconds)		
		ARTOO	RAND	$\frac{ARTOO}{RAND}$
ARRAYED_ LIST	append	311	191	1.62
	do_all	137	18	7.48
	do_if	**2**	**39**	**0.05**
	fill	**40**	**256**	**0.16**
	for_all	138	17	7.93
	is_inserted	**3**	**7**	**0.43**
	make	2	1	2.80
	make_filled	**2**	**4**	**0.50**
	prune_all	**3**	**367**	**0.01**
	put	11	4	2.67
	put_left	**32**	**331**	**0.10**
	put_right	**132**	**291**	**0.45**
	resize	320	30	10.40
	there_exists	151	17	8.78
	wipe_out	546	123	4.41
ACTION_ SEQUENCE	arrayed_list_make	564	174	3.24
	call	**10**	**67**	**0.15**
	duplicate	196	13	14.46
	for_all	64	21	3.00
	is_inserted	5	2	2.36
	make_filled	49	13	3.65
	put	4	4	1.15
	remove_right	**201**	**475**	**0.42**
	resize	187	160	1.17
	set_source_connection_agent	**96**	**112**	**0.86**
	there_exists	67	2	33.83

Table 8.8: Results for two of the tested classes, showing the time required by ARTOO and RAND to uncover the first fault in each routine in which they both found at least one fault, and their relative performance. In most cases ARTOO requires more time than RAND to find a fault.

TOO is more attractive due to the reduced number of tests it generates before it uncovers a fault.

8.3.3 Discussion

We chose to compare the performance of the two strategies when run over the same duration, although other similar comparative studies in the literature use rather the number of generated tests or an achieved level of code coverage as

Class	Tests to first fault		
	ARTOO	RAND	ARTOO/RAND
ACTION_SEQUENCE	**293.72**	**3449.76**	**0.09**
ARRAY	**437.19**	**856.39**	**0.51**
ARRAYED_LIST	**206.80**	**3317.80**	**0.06**
BOUNDED_STACK	**282.50**	**357.17**	**0.79**
FIXED_TREE	**333.99**	**463.91**	**0.71**
HASH_TABLE	**581.21**	**2734.42**	**0.21**
LINKED_LIST	**238.20**	**616.71**	**0.38**
STRING	**279.64**	**1561.60**	**0.17**
Average	*331.66*	*1669.72*	*0.19*
Class	Time to first fault (seconds)		
	ARTOO	RAND	ARTOO/RAND
ACTION_SEQUENCE	131.53	95.11	1.38
ARRAY	133.21	21.23	6.27
ARRAYED_LIST	122.16	113.42	1.07
BOUNDED_STACK	128.00	11.45	11.18
FIXED_TREE	**127.73**	**136.64**	**0.93**
HASH_TABLE	164.41	65.85	2.49
LINKED_LIST	98.39	18.14	5.42
STRING	**85.03**	**144.28**	**0.58**
Average	*123.81*	*75.77*	*1.63*

Table 8.9: Averaged results per class. ARTOO constantly requires fewer tests to find the first fault: on average 5 times less tests than RAND. The overhead that the distance calculations introduce in the testing process causes ARTOO to require on average 1.6 times more time than RAND to find the first fault.

Class	Routine	Tests to first fault	Time to first fault (seconds)	#faults
ARRAYED_LIST	remove	167	46	1
FIXED_TREE	child_is_last	717	283	1
FIXED_TREE	duplicate	422	134	1
STRING	grow	492	163	2
STRING	multiply	76	17	2

Table 8.10: Faults which only ARTOO finds

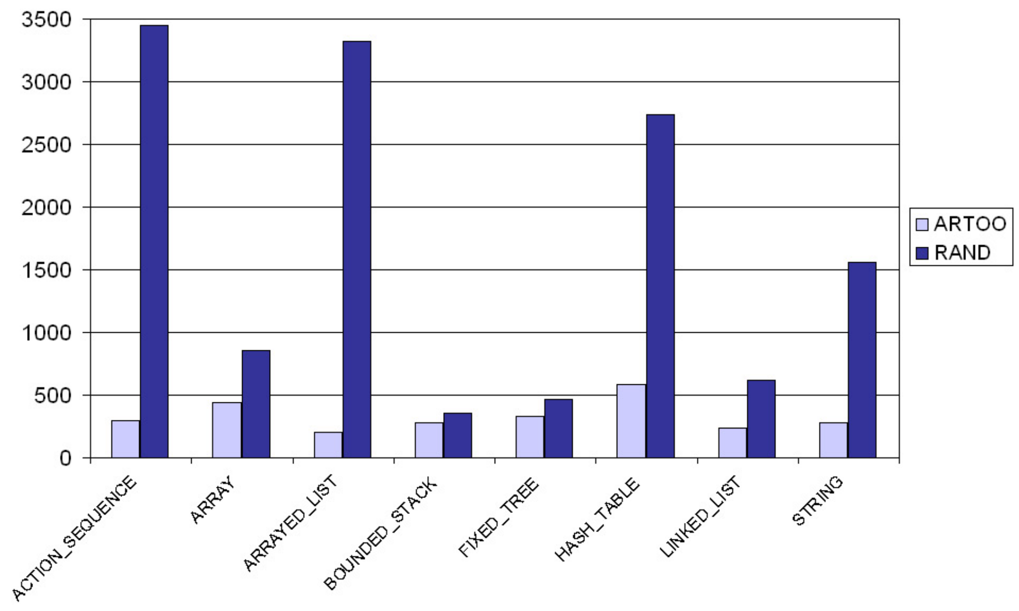

Figure 8.11: Comparison of the average number of tests cases to first fault required by the two strategies for every class. ARTOO constantly outperforms RAND.

the stopping criterion. We preferred to use time because, in industrial projects and especially in the testing phases of these projects, time is probably the most constraining factor.

These results show that, compared to a random testing strategy combined with limit testing, ARTOO generally reduces the number of tests required until a fault is found, but suffers from a time performance penalty due to the extra computations required for calculating the object distances. The times reported here are *total* testing times; they include both the time spent on generating and selecting test cases and the time spent on actually running the tests. Total time is the measure that most resembles how the testing tool would be used in practice, but this measure is highly dependent on the time spent running the software under test. The test scope of the experiment described above consists of library classes whose routines generally implement relatively simple computations. When testing more computation-intensive applications, the number of tests that can be run per time unit naturally decreases, hence the testing strategy that needs less tests to uncover faults would be favored.

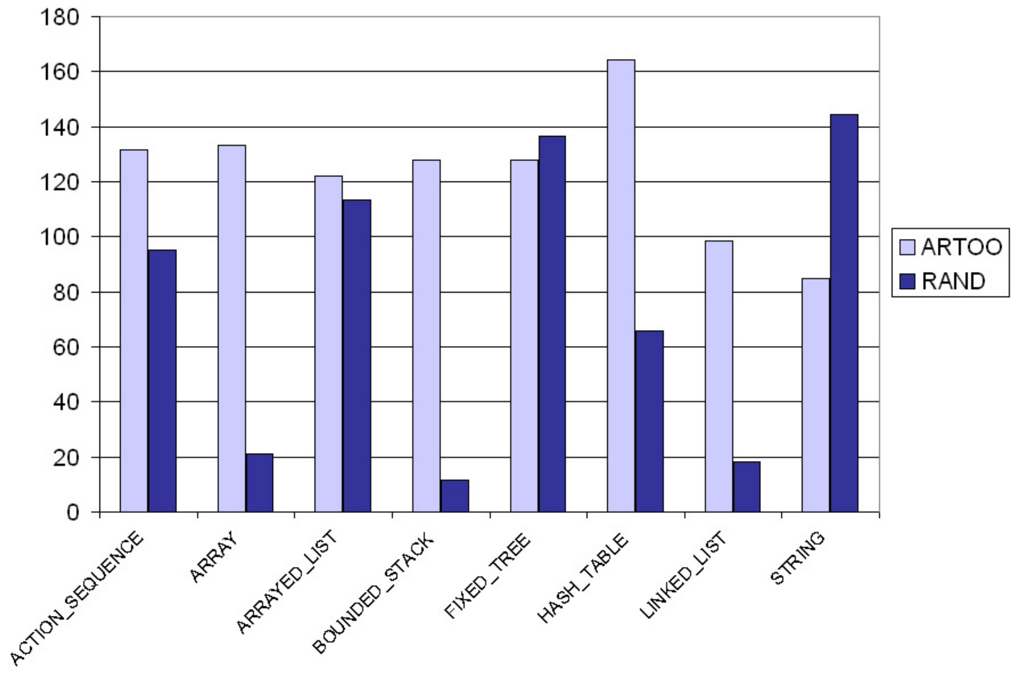

Figure 8.12: Comparison of the average time to first fault required by the two strategies for every class. RAND is generally better than ARTOO.

8.3.4 *Threats to the validity of generalizations of the results*

The biggest threat to the validity of generalizations of these results is probably the test scope: the limited number of seeds and the tested classes. The results presented here were obtained by averaging out over 5 seeds of the pseudo-random number generator. Given the role randomness plays in both the compared testing algorithms, averaging out over more seeds would produce more reliable results. Likewise, we have chosen the tested classes so that they are fairly diverse (in terms of their semantics and of various code metrics), but testing more classes would yield more generalizable results.

8.3.5 *Conclusions*

The experimental results show that ARTOO finds real faults in real software. Compared to RAND (random testing combined with limit value testing), ARTOO significantly reduces the number of tests generated and run until the first fault is found, on average by a factor of 5 and sometimes by as much as two orders of magnitude. The guided input selection process that ARTOO employs does, however, entail an overhead which is not present in unguided random testing.

This overhead leads to ARTOO being on average 1.6 times slower than RAND in finding the first fault. These results indicate that ARTOO, at least in its current implementation, should be applied in settings where the number of test cases generated until a fault is found is more important than the time it takes to generate these test cases; in other words, settings in which the cost of running and evaluating test cases is higher than the cost of generating them. This can be the case, for instance, when there is no automated oracle and thus manual inspection of test cases is necessary.

The results also show that both ARTOO and RAND find faults that the other does not find in the same time, so they should be used in combination.

8.4 Finding faults: manual testing vs. random testing vs. user reports

As stated in the beginning of this chapter, we consider that progress in testing requires comparing the various testing tools and methods that exist nowadays in terms of their fault detection ability. Many studies [61, 64, 62, 86, 71, 143, 53, 73, 127, 80] have tried to answer this question, but none of the studies focusing on the number of faults detected by different strategies conclusively shows that one testing strategy clearly outperforms another.

We hence conjecture that the *number of faults* is too coarse a criterion for assessing testing strategies. Consequently, in the experiments described below, we investigate whether or not the *kind of faults* is a more suitable discriminator between different fault detection strategies. To this end, we classify faults into categories, and analyze which strategies find which categories of faults. This work's main result is empirical evidence that different strategies do indeed uncover significantly different kinds of faults [44]. This complements the seminal work by Basili and Selby [21] who compared the types of faults found by code reading, manual functional testing, and manual structural testing.

The three fault detection strategies we analyzed are manual unit testing, field use (with corresponding bug reports), and automated random testing. They are representative of today's state of the art: the first two are widely used in industry, and the last one reflects the research community's current interest in automated testing solutions. Although random testing is only one among trends in current research, it is attractive because of its simplicity and because it makes test case generation comparatively cheap. Moreover, there is no conclusive evidence that random testing is any worse at finding faults than other automated strategies.

To investigate the performance of *random testing* we ran AutoTest on 39 classes from EiffelBase, which we did not modify in any way. AutoTest found a total of 165 faults in these classes. To investigate the performance of *manual testing*, we analyzed the faults found by students who were explicitly asked to test three classes, two created by us and one slightly adapted from EiffelBase. *Faults in the field* are taken from user-submitted bug reports on the EiffelBase library. We evaluated these three ways of detecting faults by comparing the number and distribution of faults they detect via a custom-made classification containing 21 categories.

Fault classifications have previously been used to analyze the difference between inspections and software testing. Yet, as far as we know, this is the first study that

- Develops a classification of faults specifically adapted to contracted O-O software

- Uses this classification to compare an automated random testing strategy to manual testing, and to furthermore compare testing results to faults detected in the field.

8.4.1 Classifications of faults

Two dimensions characterize a fault in programming languages with support for embedding executable specifications: the fault's *location* — whether it occurs in the specification or in the implementation; the fault's *cause*, the real underlying issue.

The following paragraphs discuss both dimensions and introduce the resulting fault categories. The classification is not domain-specific. Although other fault classification schemes exists, as discussed in section 4.5, we are not aware of any such schemes for contracted code.

Specification and implementation faults

In contract-equipped programs, the software specification is embedded in the software itself. Contract violations are one of the sources of failures. Hence, faults can be located both in the implementation and in the contracts. From this perspective we distinguish between the following two types of faults:

- A *specification fault* is a mismatch between the intended functionality of a software element and its explicit specification (in the context of this study, the contract). Specification faults reflect specifications that are not valid, in the sense that they do not conform to user requirements. The correction of specification faults requires changing the specification (plus possibly also the implementation). As an example, consider a routine *deposit* of a class *BANK_ACCOUNT* with an integer argument representing the amount of money to be deposited into the account. The intention is for that argument to be positive, and the routine only works correctly in that case. If the precondition of *deposit* does not list this property, the routine has a specification fault.

- In contrast, an *implementation fault* occurs when a piece of software does not fulfill its explicit specification, here its contract. The correction of implementation faults never requires changing the specification. Suppose the class *BANK_ACCOUNT* also contains a routine *add_value* that should add a value, positive or negative, to the account. If the precondition does not specify any constraint on the argument but the code assumes that it is a positive value, then there is a fault in the implementation.

Figure 8.13 represents graphically the levels where each kind of fault occurs.

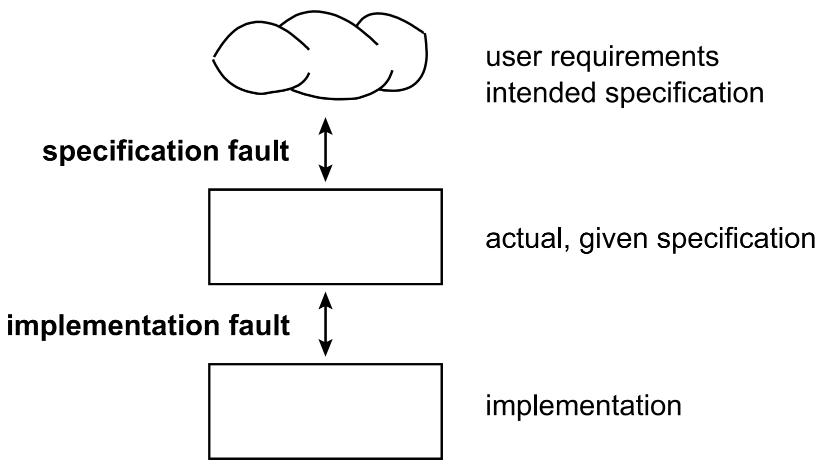

Figure 8.13: Specification vs. implementation faults

The notion of specification fault assumes that we have access to the "intended specification" of the software: the real specification that it should fulfill. When analyzing the faults in real-world software, this is not always possible. Access to the original developers is generally not available. To infer the intended specification, one must rely on subjective evidence such as:

- The comments in the routines under test
- The specifications and implementations of other routines in the same class

- How the tested routines are called from various parts of the software system

This strategy resembles how a developer not familiar with the software would proceed to find out what it is supposed to do.

Classification of faults by their cause

Some kinds of specification and implementation faults tend to recur over and over again in practice. Their study makes it possible to obtain a more detailed classification by grouping these faults according to the corresponding human mistakes or omissions — their causes. By analyzing the cause for all faults encountered in our study, we obtained the categories described below. The classification was created with practical applicability in mind and mainly focuses on either a mistake in the programmer's thinking or a misused programming mechanism.

Specification faults. An analysis of specification faults led to the following cause-based categories, grouped by the type of contract that they apply to.

1. We identified the following faults related to *preconditions*:
   - **Missing non-voidness precondition**: a precondition clause is missing, specifying that a routine argument, class attribute, or other reference denoted by an argument or attribute should not be void.
   - **Missing min/max-related precondition**: a precondition clause is missing, specifying that an integer argument, class attribute, or other integer denoted by an argument or attribute should have a certain value related to the minimum/maximum possible value for integers.
   - **Missing other precondition part**: a precondition is under-specified in another way than the previous cases.
   - **Precondition disjunction due to inheritance**: with multiple inheritance it can be the case that a routine merges several others, inherited from different classes. In this case, the preconditions of the merged routines are composed, using disjunction, with the most current ones. Faults in this category appear because of this language mechanism.
   - **Precondition too restrictive**: the precondition of a routine is stronger than it should be.
2. Faults related to the *postcondition* include:
   - **Wrong postcondition**: the postcondition of a routine is incorrect.
   - **Missing postcondition**: the postcondition of a routine is missing.
3. Faults related to *class invariants* include only one kind: the **missing invariant clause** — a part of a class invariant is missing.

4. Faults related to `check` assertions include only one kind: the **wrong check assertion** — the routine contains a `check` condition that does not necessarily hold.

5. Finally, the following faults apply to *all contracts*:

    - **Faulty specification supplier**: a routine used by the contract of the routine under test contains a fault, which makes the contract of the routine under test incorrect.
    - **Calling a routine outside its precondition from a contract**: the fault appears because the contract of the routine under test calls another routine without fulfilling the latter's precondition.
    - **Min/max int related fault in specification** (other than missing precondition): the specification of the routine under test lacks some condition(s) related to the minimum/maximum possible value for integers. (Our examples so far do not cover floating-point computation.)

The categories in this classification have various degrees of granularity. The reason is that the classification was derived from faults obtained through widely different mechanisms: by AutoTest; by manual testers; by users of the software. The categories emerged by inductively identifying recurring patterns in existing faults, rather than by trying to fit faults deductively into a scheme defined a priori. Where such patterns could not be found, the categories are rather coarse-grained.

Implementation faults. The analysis of implementation faults led to the following cause-based categories.

- **Faulty implementation supplier**: a routine called from the body of the routine under test contains a fault, which does not allow the routine under test to function properly.
- **Wrong export status**: this category refers particularly to the case of creation procedures, which in Eiffel can also be exported as normal routines. The faults classified in this category are due to routines being exported as both creation procedures and normal routines, but which, when called as normal routines, do not fulfill their contract, as they were meant to be used only as creation procedures.
- **External fault**: Eiffel allows the embedding of routines written in C. This category refers to faults located in such routines.
- **Missing implementation**: the body of a routine is empty, often signaling an uncompleted attempt at top-down algorithm design.
- **Case not treated**: the implementation does not treat one of the cases that can appear, typically in an `if` branch.
- **Catcall**: due to the implementation of type covariance in Eiffel, the compiler cannot (in the Eiffel version used) detect some routine calls that are

not available in the actual type of the target object. Such violations can only be detected at runtime. This class groups faults that stem from this deficiency of the type system.

- **Calling a routine outside its precondition from the implementation**: the fault appears because the routine under test calls another routine without fulfilling the latter's precondition.
- **Wrong operator semantics**: the implementation of an operator is faulty, in the sense that it causes the application of the operator to have a different effect than intended.
- **Infinite loop**: executing the routine can trigger an infinite loop, due to a fault in the loop exit condition.

Three of the above categories are specific to the Eiffel language and would not be directly applicable to languages which do not support multiple inheritance (precondition disjunction due to inheritance), covariant definitions (catcalls), or the inclusion of code written in other programming languages (external faults). All other categories are directly applicable to other object-oriented languages with support for embedded and executable specifications.

8.4.2 Experimental setup

To see how random testing performs, we ran AutoTest on classes from the EiffelBase library. Overall, we randomly tested 39 classes from the 5.6 version of the library and found a total of 165 faults in them.

We then examined bug reports from users of the EiffelBase library. From the database of bug reports, we selected those referring to faults present in version 5.6 of the EiffelBase library and excluded those which were declared by the library developers to not be faults or those that referred to the .NET version of EiffelBase, which we cannot test with AutoTest. Our analysis hence refers to the remaining 28 bug reports fulfilling these criteria.

To determine how manual testing compares to random testing, we organized a competition for students of computer science at ETH Zurich. 13 students participated in the competition. They were given 3 classes to test. The task was to find as many faults as possible in these 3 classes in 2 hours. Two of the classes were written by us (with implementation, contracts, and purposely introduced faults from various of the above categories), and one was an adapted version of the *STRING* class from EiffelBase. Table 8.11 shows some code metrics for these 3 classes: number of lines of code (LOC), number of lines of contract code (LOCC), and number of routines. We intentionally chose one class that was significantly larger and more complex than the others to see how the students would cope with it. Although such a class is harder to test, intuition suggests that it is more likely to contain faults.

The students had various levels of experience in testing O-O software; most of them had had at least a few lectures on the topic. 9 out of the 13 students stated

Table 8.11: Classes tested manually

Class	LOC	LOCC	#Routines
MY_STRING	2444	221	116
UTILS	54	3	3
BANK_ACCOUNT	74	13	4

Table 8.12: Random Testing vs. User Reports. Random tests find many specification faults; users find many implementation faults.

	Spec. faults	Implem. faults
AutoTest	103 (62.42%)	62 (37.58%)
User reports	10 (35.71%)	18 (64.29%)

in a questionnaire they filled in after the competition that they usually or always unit test their code as they write it. They were allowed to use any technique to find faults in the software under test, except for running AutoTest on it. Although they would have been allowed to use other tools (and this was announced before the competition), all the students performed only manual testing. In the end they had to produce test cases revealing the faults that they had found, through a contract violation or another exception.

8.4.3 Random testing vs. user reports

Table 8.12 shows the distribution of specification and implementation faults (1) found by random testing (labeled "AutoTest" in the table) 39 classes from the EiffelBase library and (2) recorded in bug reports (provided by the maintainers of the library) from professional users. Note that the results in this table refer to more classes tested with AutoTest than for which there are user reports: even if there are no user reports on a specific class, the class may still have been used in the field.

Almost two thirds of the faults found by random testing were located in the specification of the software, that is, in the contracts. This indicates that random testing is especially good at finding faults in the contracts. In the case of faults collected from users' bug reports, the situation is reversed: almost two thirds of user reports refer to faults in the implementation.

Table 8.13 shows the identifiers used for brevity for each fault category. Table 8.14 presents a more detailed view of the specification and implementation faults found by AutoTest and recorded in users' bug reports, grouping the faults by their cause, as explained above.

This sheds more light on differences between faults reported by users and

Table 8.13: Identifiers for the fault categories

Cause	Id
Specification faults	
Missing non-voidness precondition	S1
Missing min/max-related precondition	S2
Missing other precondition part	S3
Faulty specification supplier	S4
Calling a routine outside its precondition from a contract	S5
Min/max int related fault in spec (other than missing precondition)	S6
Precondition disjunction due to inheritance	S7
Missing invariant clause	S8
Precondition too restrictive	S9
Wrong postcondition	S10
Wrong check assertion	S11
Missing postcondition	S12
Implementation faults	
Faulty implementation supplier	I1
Wrong export status	I2
External fault	I3
Missing implementation	I4
Case not treated	I5
Catcall	I6
Calling a routine outside its precondition from the implementation	I7
Wrong operator semantics	I8
Infinite loop	I9

those found by automated testing, and exposes strengths and weaknesses of both approaches. One difference that stands out relates to faults related to extreme values (either Void references or numbers at the lower or higher end of their representation interval) in the specification. Around 30% of the faults uncovered by AutoTest are in one of these categories, whereas users do not report any such faults. Possible explanations are that such situations are not encountered in practice; that users do not consider them to be worth reporting; or that users rely on their intuition on the range of acceptable inputs for a routine, rather than the routine's precondition, and their intuition corresponds to the intended specification, not to the erroneous one provided in the contracts.

A further difference results from AutoTest's ability to detect faults from the categories "faulty specification supplier" and "faulty implementation supplier." They mean that AutoTest can report that certain routines do not work properly because they depend on other faulty routines. In our records users never report such faults: they only indicate the routine that contains the fault, without mentioning other routines that also do not work correctly because of the fault in their supplier. An important piece of information gets lost this way: after fixing the fault, there is no incentive to check whether the clients of the routine now work properly, meaning to check that the correction in the supplier allows the client to work properly too.

Random testing is particularly bad at detecting some categories of faults: too strong preconditions, faults that are a result of wrong operator semantics, infinite loops, missing routine implementations. None of the 165 faults found by AutoTest and examined in this study belonged to any of the first three categories, but the users reported at least one fault in each. It is not surprising that AutoTest has trouble detecting such faults, because:

- If AutoTest tries to call a routine with a too strong precondition and does not fulfill this precondition, the testing engine will simply classify the test case as invalid and try again to satisfy the routine's precondition by using other inputs.
- AutoTest also cannot detect infinite loops: if the execution of a test case times out, it will classify the test case as "bad response"; this means that it is not possible for the tool to decide if a fault was found or not — the user must inspect the test case and decide.
- Users of the EiffelBase library could report faults related to operators being implemented with the wrong semantics. Naturally, to decide this, it is necessary to know the intended specification of the operator.
- AutoTest also cannot detect that the implementation of a routine body is missing unless this triggers a contract violation. An automatic tool can of course, through code analysis, find empty routine bodies statically, but not decide if this is a fault. Note that in these cases, the overall number of detected faults is rather low, which suggests special care in generalizing these findings.

We also ran AutoTest exclusively on the classes for which users reported faults to see if it would find those faults (except three classes which AutoTest cannot currently process as they are either expanded or built-in). When run on each class in 10 sessions of 3 minutes (where each session used a different seed for the pseudo-random number generator), AutoTest found a total of 268 faults[2]. 4 of these were also reported by users, so 21 faults are solely reported by users and 264 solely by AutoTest. AutoTest detected only one of the 18 implementation faults (5%) reported by users and 3 out of the 7 specification faults (43%). While theoretically it could, AutoTest did not find the user-reported faults belonging to

[2]However, 183 of these faults were found through failures caused by the classes RAW_FILE, PLAIN_TEXT_FILE and DIRECTORY through operating system signals and I/O exceptions, so it is debatable if these can indeed be considered faults in the software.

such categories as "wrong export status" or "case not treated." Longer testing times might, however, have produced different results.

8.4.4 Random testing vs. manual testing

To investigate how AutoTest performs when compared to manual testers (the students participating in the competition), we ran AutoTest on the 3 classes that were tested manually. The tool tested each class in 30 sessions of 3 minutes, where each session used a different seed for the pseudo-random number generator. Table 8.15 shows a summary of the results. It displays a categorization of the fault according to our classification scheme (the category ids are used here; they can be looked up in Table 8.13), the name of the class where a fault was found by either AutoTest or the manual testers, how many of the manual testers found the fault out of the total 13 and a percent representation of the same information, and finally, in the last column, x's mark the faults that AutoTest detected.

The table shows that AutoTest found 9 out of the 14 (64%) faults that humans detected and 2 faults that humans did not find. The two faults that only AutoTest found do not exhibit any special characteristics, but they occur in class *MY_STRING*, which is considerably larger than the other 2 classes. We conjecture that, because of its size, students tested this class less thoroughly than the others. This highlights one of the clear strengths of the automatic testing tool: the sheer number of tests that it generates and runs per time unit and the resulting routine coverage.

Conversely, three of the faults that AutoTest does not detect were found by more than 60% of the testers. One of these faults is due to an infinite loop; AutoTest, as discussed above, classifies timeouts as test cases with a bad response and not as failures. The other two faults belong to the categories "missing non-voidness precondition" and "missing min/max-related precondition." Although the strength of AutoTest lies partly in detecting exactly these kinds of faults, the tool fails to find them for these particular examples in the limited time it is given. This once again stresses the role that randomness plays in the approach, with both advantages and disadvantages.

8.4.5 Summary and consequences

Three main observations emerge from the preceding analysis. First, random testing is good at detecting problems in specifications. It is particularly good with problems related to limit values. Problems of this kind are not reported in the field but tend to be caught by manual testers.

Second, AutoTest is not good at detecting problems with too strong preconditions, infinite loops, missing implementations and operator semantics. This is due to the very nature of automated contract-based random testing.

Third, in a comparison between automated and manual testing (i.e., not taking into consideration bug reports), AutoTest detects around two thirds of faults also detected by humans, plus a few others. This speaks strongly in favor of running the tool on the code before having it tested by humans. The human testers may find faults that the tool misses, but a great part of their work will be done at no other cost than CPU power.

8.4.6 Discussion

AutoTest finds significantly more faults in contracts than in implementations. This might seem surprising, given that contracts are boolean expressions and typically take up far fewer lines of code than the implementation (14% of the code on average in our study). Two questions naturally arise. One, are there more faults in contracts than in implementations, i.e., do the results obtained with AutoTest reflect the actual distribution of faults? Two, is it interesting at all to find faults in contracts, knowing that contract checking is usually disabled in production code?

We do not know the answer to the first question. We cannot deduce from our results that there are indeed more problems in specifications than in implementations. The only thing we can deduce is that random testing that takes special care of extreme values detects more faults in specifications than in implementations. Around 45% of the faults are uncovered in preconditions, showing that programmers often fail to specify correctly the range of inputs or conditions on the state of the input accepted by routines.

It is also important to point out that a significant proportion of specification errors are due to void-related issues, which are scheduled to go away as the new versions of Eiffel, starting with 6.2 (Spring 2008), implement the "attached type" mechanism [109] which removes the problem by making non-voidness part of the type system and catches violations at compile time rather than run time.

On whether it is useful or interesting to detect and analyze faults in contracts, one must keep in mind that most often the same person writes both the contract and the body of a routine. A fault in the contract signals a mistake in this person's thinking just as a fault in the routine body does. Once the routine has been implemented, client programmers who want to use its functionality from other classes look at its contract to understand under what conditions the routine can be called (expressed by its precondition) and what the routine does (the postcondition expresses the effect of calling the routine on the state). Hence, if the routine's contract is incorrect, the routine will most likely be used incorrectly by its callers, which will produce a chain of faulty routines. The validity of the contract is thus as important as the correctness of the implementation.

The existence of contracts embedded in the software is a key assumption both for the proposed fault classification and for the automated testing strategy used. We do not consider this to be too strong an assumption because it has been shown [31] that programmers willingly use a language's integrated support for Design by Contract, if available.

The evaluation of the performance of random testing performed here always considers the faults that AutoTest finds over several runs, using different seeds for the pseudo-random number generator. We have shown that random testing is predictable in the *number* of faults that it finds, but not in the *kind* of faults that it finds (section 8.2). Hence, in order to reliably assess the types of faults that random testing finds, it is necessary to sum up the results of different runs of the tool.

In addition to pointing out strengths and weaknesses of a certain testing strategy, a classification of repeatedly occurring faults based on the cause of the fault also brings insights into those mechanisms of the programming language that are particularly error-prone. For instance, faults due to wrong export status of creation procedures show that programmers do not master the property of the language that allows creation procedures to be exported both for object creation and for being used as normal routines.

8.4.7 Threats to the validity of generalizations of the results

The biggest threat to the generalization of the results presented here is the small size of the set of manually tested classes, of the analyzed user bug reports, and of the group of human testers participating in the study. In future work this study should be expanded to larger and more diverse code bases.

As explained above, we only had access to bug reports submitted by users for the EiffelBase library. Naturally, these are not all the faults found in field use, but only the ones that users took the time to report. Interestingly, for all but one of these reports the users set the priority to either "medium" or "high"; the severity, on the other hand, is "non-critical" for 7 of the reports and either "serious" or "critical" for the others. This suggests that even users who take the time to report faults only do so for faults that they consider important enough.

As we could not perform the study with professional testers, we used bachelor and master students of computer science; to strengthen the incentive for finding as many faults as possible, we ran this as a competition with attractive prizes for the top fault finders. In a questionnaire they filled in after the competition, 4 of the students declared themselves to be proficient programmers and 9 estimated they had "basic programming experience". 7 of them stated that they had worked on software projects with more than 10,000 lines of code and the others had only worked on smaller projects. As mentioned above, two of the classes under test given to the students were written by us and we also introduced the faults in them. These faults were meant to be representative of actual faults occurring in real software, so they were created as instances of various categories described above, but the very approach introduces a bias. All these aspects limit the generality of our conclusions.

A further threat to the generalization of our results stems from the peculiarities of the random testing tool used. AutoTest implements one particular algorithm for random testing and the results described here would probably not be identical for other approaches to the random testing of O-O programs (e.g., [47]). In particular, we make use of extreme values to initialize the object pool (5.5). While void objects are rather likely to occur in practice, extreme integer values are not. In other words, the approach, as mentioned, is not entirely random.

Also as noted, compile-time removal of void-related errors will affect the results, for ISO Eiffel and other languages that have the equivalent of an "attached type" mechanism (e.g. Spec# [20]).

Another source of uncertainty is the assignment of defects to a classification. Finding a consistent assignment among several experts is difficult [83]. In our study, the author performed this task. While this yields consistency, running the experiment with a different person might produce different results.

Finally, the programming language used in the study, Eiffel, also influenced the results. As explained above, a few of the fault categories are closely related to the language mechanisms that are misused or that allow the fault to occur. This is to be expected in a classification of software faults based on the cause of the faults.

8.4.8 Conclusions

One of the main goals of this work is to understand if different ways of detecting faults detect different kinds of faults. This would help answer a question of utmost importance: which testing strategy should be applied under which circumstances? A further motivation was the conjecture that a reason for the inconclusiveness of earlier comparative studies is that the number of detected faults alone is too strong an abstraction for comparing testing strategies.

We examined the kind of faults that random testing finds, and whether and how these differ from faults found by human testers and by users of the software. The experiments suggest that these three strategies for finding software faults have different strengths and areas of applicability. None of them subsumes any other in terms of performance. Random testing with AutoTest has the advantage of being completely automatic and the experiments show that the tool indeed finds a high number of faults in little time. Humans, however, find faults that AutoTest misses. AutoTest also finds faults that testers miss. The conclusion is that random tests should be used alongside with manual tests. Given earlier results on comparing different QA strategies, this is not surprising, but we are not aware of any systematic studies that showed this for random testing.

Table 8.14: Random Testing vs. User Reports: a finer-grained classification of specification and implementation faults. AutoTest is good at detecting limit value problems and bad at detecting problems such as too strong preconditions, missing implementations, and infinite loops.

Category id	Number of faults		Percentage of faults	
	AutoTest	Users	AutoTest	Users
Specification faults				
S1	22	0	13.33%	0.00%
S2	23	0	13.94%	0.00%
S3	28	3	16.97%	10.71%
S4	7	0	4.24%	0.00%
S5	0	0	0.00%	0.00%
S6	4	0	2.42%	0.00%
S7	2	0	1.21%	0.00%
S8	3	0	1.82%	0.00%
S9	0	2	0.00%	7.14%
S10	12	2	7.27%	7.14%
S11	2	0	1.21%	0.00%
S12	0	3	0.00%	10.71%
Specification faults total	*103*	*10*	*62.42%*	*35.71%*
Implementation faults				
I1	47	0	28.48%	0.00%
I2	0	2	0.00%	7.14%
I3	1	0	0.61%	0.00%
I4	2	2	1.21%	7.14%
I5	4	7	2.42%	25.00%
I6	3	1	1.82%	3.57%
I7	5	1	3.03%	3.57%
I8	0	1	0.00%	3.57%
I9	0	4	0.00%	14.29%
Implementation faults total	*62*	*18*	*37.58%*	*64.29%*

Table 8.15: Random Testing vs. Manual Testing AutoTest finds almost all faults detected by humans, plus two extra faults.

Id	Class	# testers	AutoTest
S1	BANK_ACCOUNT	8 (61.5%)	
S1	UTILS	5 (38.5%)	x
S1	MY_STRING	1 (7.7%)	x
S2	BANK_ACCOUNT	8 (61.5%)	
S2	UTILS	7 (53.8%)	x
S2	MY_STRING	1 (7.7%)	x
S2	MY_STRING	5 (38.5%)	x
S2	MY_STRING	1 (7.7%)	x
S2	MY_STRING	0 (0%)	x
S3	BANK_ACCOUNT	1 (7.7%)	x
S3	UTILS	4 (30.8%)	x
S10	MY_STRING	1 (7.7%)	
I2	BANK_ACCOUNT	4 (30.8%)	x
I6	MY_STRING	1 (7.7%)	
I7	MY_STRING	0 (0%)	x
I9	MY_STRING	9 (69.2%)	

CHAPTER 9

FUTURE WORK

This thesis explores and contributes to several research directions related to random contract-based tests and opens the way for investigating several other research questions. In this chapter we discuss possible directions for future research, grouped by the topic they address.

9.1 Random contract-based testing and AutoTest

Several improvements of the AutoTest tool are possible, both in terms of the algorithms that it implements and of improving its performance and usability.

AutoTest allows the easy integration and objective comparison of various testing strategies, due to the tool's pluggable architecture. In the same manner in which we implemented ARTOO as a plug-in strategy for AutoTest and could compare purely random testing to adaptive random testing on the same grounds, other test generation strategies can be implemented in the tool and then be objectively compared to its other plug-ins. Modifications of the existing algorithms are of course possible:

- Tweaking the random-based algorithm to improve its performance, for instance by selecting values for basic types from values used verbatim in the program source code rather than from the predefined sets.

- Using dynamic inference of abstract types [67]. This method finds sets of related variables (such as variables declared as integers which actually represent sums of money, or others declared as integers which represent ages of people). Determining these sets allows for them to be treated differently by the testing strategy.

- Investigating new ways of integrating manual and generated tests and of allowing testers more control over the testing process of AutoTest (for instance, specifying initializations, associating weights with routines and classes, providing oracles).

AutoTest currently does not use the preconditions of the routines under test except as filters for invalid generated inputs. Finding ways of guiding the input generation process so that it produces only inputs satisfying these preconditions is another avenue of interesting research.

AutoTest can also still be improved in terms of its usability by any software developer. Although the tool has been available in open source on the Internet for several years [97] and has a constantly growing community of users, we believe the tool's popularity would greatly increase if it were seamlessly integrated into EiffelStudio, the standard development environment for Eiffel. Ongoing work is focusing on this task.

9.2 Adaptive random testing for object-oriented systems

The basic algorithm presented in section 6.2 for applying adaptive random testing techniques to O-O software can be modified in several ways.

Having the possibility to compute a distance between objects allows clustering techniques to be applied to objects. Any of the known clustering algorithms can be applied based on this distance, so it is possible to group together objects that are similar enough. This allows ARTOO to be optimized by computing the distances to the cluster centers only, and not to each individual object. The precision is dependent on the maximum distance between two distinct objects within a given cluster. A preliminary implementation of this testing strategy shows an average improvement of the time to first fault over ARTOO of 25% at no cost in terms of faults found.

The implementation of ARTOO used in the experiments presented in section 8.3 uses the "complete" definition of the object distance, as described in Section 6.1. Using a less computationally intensive definition of the object distance might still require fewer tests to uncover a fault than random testing, but also less time. Such alternatives can be explored in order to improve the performance of ARTOO.

The definition of the object distance provided in Section 6.1 uses several constants, whose values can be changed to tweak the distance computation and to change the contribution of each component of the distance to the overall measure. How these constants affect the performance of ARTOO remains to be investigated.

The definition of the object distance only uses the syntactic form of the objects and does not take their semantics into account in any way. This approach has the merit of being completely automatable. Integrating semantics into the computation would require human intervention, but would certainly enrich the model and its accuracy and allow finer-grained control of the developer over the testing process in ARTOO. Some support for this is already available through the constants used in the object distance calculation, whose values can easily be changed for fine tuning the distance. Future work can investigate this idea and other possibilities for providing further support for integrating semantics into the object distance.

Furthermore, other applications of the object distance than to testing are possible, but have not been addressed in the work presented here.

9.3 Contract inference

The case study that we performed to compare programmer-written to inferred contracts only considered library classes and student code (which also consisted of library-style classes). It would be interesting to also investigate how assertions inferred for applications compare to programmer-written assertions.

A promising idea seems to be "push-button inference": using automated testing tools to generate the test suites necessary for the assertion inference instead of handmade tests.

Furthermore, other applications for assertion inference are possible, such as improving a test suite: since the quality of the inferred assertions depends on the test suites used to exercise the system, the quality of the inferred assertions (both in absolute terms and compared to programmer-written assertions) can be used to estimate the quality of the test suite. This assumption can be verified through an experiment checking if tests that produce better inferred assertions also uncover more faults.

Future work also includes improving both Daikon and its Eiffel front-end, based on the insights gained through the study described in section 7.3.

9.4 Evaluations of testing strategies

Comparisons between testing strategies, both in absolute and in relative terms, are a wide and important field of study. The results presented in chapter 8 contribute to the knowledge about the absolute performance of random testing and how it compares to ARTOO and to manual testing, but they do not offer comparisons between random testing and other strategies. We consider such comparisons are essential, because the great variety of automated testing strategies that are now available leaves developers and testers wondering as to the choice of testing tool that would deliver the best results for their projects.

Further investigations into the performance of random testing are also possible and should answer questions such as:

- What are the characteristics of randomly-generated fault-revealing tests and test inputs? Are they simple or relatively complicated?
- What are the characteristics of the programs to which random testing is best suited?
- What is the performance of random testing over timeouts longer than the ones used in our experiments?
- What influence does a system's coupling and cohesion have on the efficiency of random tests?
- What other strategy best complements random testing, hence should be used in combination with it?

Studying the evolution over time of the number of faults found by a strategy could provide an answer to one of the fundamental questions in software testing: when should one stop testing?[1] If an approximation of the curve describing this evolution is known in advance or can be deduced from existing data, then it should also be possible to estimate, at any given time t in the testing session, the time interval δt after which the next fault will be found. When this interval exceeds a preset limit, one can decide to stop the testing session. (This naturally does not take into account a varying degree of severity for the faults, but assumes that all faults are equally important.)

In section 8.1.2 we presented an estimation for random testing of the evolution of found faults over time based on an experiment involving 1500 hours of testing 8 classes. However, because each class was tested in several sessions of 30 minutes, these results do not provide any insight into the evolution of the number of faults found by random testing after this timeout. Experiments are hence necessary which examine this evolution for longer timeouts. We are currently working on such experiments.

As an automated oracle is indispensable for and largely influences the performance of any automated testing strategy, more research into automatable oracles is necessary. Assertion inference tools and static analyzers are a first step in this direction. We investigated (section 8.1.2) the influence in random testing of the two oracles used widely by automated testing tools (contract violations and exceptions), but experiments are necessary to study their contributions when integrated in other testing strategies too.

The fault classification presented in section 8.4.1 can be used for and possibly extended by comparing other testing strategies. Additionally, it is necessary to further research the grounds, methods and metrics for comparing testing strategies. In particular, since the purpose of testing is to find faults, all other possible

[1]Often measures of code coverage are used to provide an estimate of how much testing is enough, but there is no conclusive evidence yet on the existence of correlations between code coverage and the remaining number of faults in a system.

measures of its efficiency must directly relate to its fault-finding ability. Thus, more evidence is necessary for showing that code/data coverage correlates with finding faults and that mutation testing is indeed indicative of the fault-revealing capability of a test suite.

A series of questions that have been investigated only very little so far relates to the psychology of testing: how do testers decide what to test? How do novices learn how to test? Why are testers reluctant to use automating tools? What and how can tools learn from the way a tester proceeds when testing a program?

Chapter 10

Summary and conclusions

The state-of-the-art in software testing has evolved greatly in recent years. Both academia and industry have been investing significant resources in improving existing testing strategies and tools and creating new ones. Since testing accounts for between 30% and 90% of the total software development effort [23] and the costs of inadequate testing infrastructure reach tens of billions of dollars annualy in the US alone [133], this effort is justified.

Research on software testing can follow many directions: proposing new algorithms and tools, improving existing ones, investigating the performance of existing techniques both theoretically and empirically, defining best practices, developing automated approaches, combining such approaches with manual ones, improving the support for testing during the other phases of the software development process, improving testing through other artifacts that it depends on (specifications, models, design, usage scenarios, etc.), combining testing and proving techniques, investigating developers' needs and adapting the testing tools to them, etc.

Despite significant recent contributions from academia in all these directions, industry has adopted very few of the emerging ideas and technologies. Even when such adoptions occur, they are slow and difficult. The challenges that industry cites most often with respect to this are usability, scalability, reliability, performance requirements, the lack of research work attempting to address analysis and generation of test inputs across several programming languages, the lack of sufficient information and documentation on research prototypes, etc. Because of these challenges, significant investments are necessary to turn research ideas into industrial-strength tools. Furthermore, the psychology of programming is also involved: developers must be motivated to make the effort involved in adopting new tools; if this motivation is not present, developers will not be willing to make the adoption effort, however small it may be.

One of the most pressing needs of the software industry is for *automated* testing solutions. Consequently, much of the recent research on software testing has been focusing on developing such solutions, with different degrees of automation. The challenges in developing techniques and tools for automated testing lie not only in creating new algorithms and approaches, but also in comparing existing tools and coming up with recommendations for testing practitioners.

This thesis contributes to both directions: it proposes new testing techniques integrated into an open source tool, which is already being used in industry, and it investigates the performance of existing and newly-proposed testing strategies.

Most of the developments and experiments performed are based on the AutoTest tool. AutoTest is an open source tool for the fully automated testing of contract-equipped Eiffel programs, but can be applied with minor modifications to other pure object-oriented programming languages using static typing and dynamic binding and with support for embedded executable specification. AutoTest uses contracts present in the code for input selection (any generated input that does not satisfy the precondition of the routine under test is discarded) and as automated oracle (any contract violation, except for direct precondition violations by the tool, signals a fault in the code). By default, AutoTest uses a random strategy for generating inputs, but other algorithms for creating test inputs can easily be plugged in.

We extended AutoTest with another method for input creation which tries to maximize the diversity of the test inputs, based on the ideas of Adaptive Random Testing [35]. This new testing strategy, called ARTOO, generates candidate inputs randomly and selects at each step the candidate that is most different from the already used test inputs. To compute a measure of how different two test inputs (objects) are, we developed the *object distance*, a distance function based on the types of the objects, their immediate values, and the distances between their attributes. Compared to the random input generation algorithm of AutoTest, ARTOO needs about 5 times less tests to uncover a first fault, but needs on average about 1.6 times more time. This measure of the testing time, however, comprises both test generation and test execution time, and is hence dependent on the running time of the tested routines.

We also explored ways of integrating automated and manual tests, by using the testers' knowledge contained in manual tests in the creation and selection of automated tests.

The contracts present in the tested classes are essential to the quality of the results delivered by AutoTest, because any property not specified in the contracts cannot be checked by the tool. Eiffel programmers are aware of the benefits that contracts bring and do include contracts in their code, as shown in a large-scale study [31], but most often they do not write complete or even correct contracts. We hence investigated the benefits that a contract inference tool can bring by performing a comparative study of assertions written by programmers and inferred by such a tool. In this study we used CITADEL, an Eiffel front-end for the Daikon assertion inference tool [55]. The study showed that such a tool produces around 6 times more relevant assertion clauses than programmers write, but only infers about 50% of all programmer-written assertions. Hence a tool like CITADEL can be used to strengthen assertions written by the programmers, but cannot completely replace their work. Among the disadvantages of such tools are the fact that they require a running system and a set of passing test cases and that they produce many incorrect and uninteresting assertions (around a third of all generated assertions).

As noted above, progress in testing requires not only the introduction of new techniques and the improvement of existing ones, but also thorough *evaluations* of techniques and comparisons between them, making clear the absolute and relative advantages and disadvantages of each approach. We hence performed extensive case studies in this direction, investigating the performance of purely random testing, of random testing combined with boundary value testing (we call this strategy "random$^+$ testing"), the predictability of random$^+$ testing, and how the faults found through random$^+$ testing compare to faults found through manual testing and by users of the software. Among the results:

- Random testing extended with boundary value testing finds significantly more faults than purely random testing
- The number of new faults found by random$^+$ testing per time unit is inversely proportional to the elapsed time
- Random testing uncovers more faults due to contract violations than to other exceptions
- Parameters of the random input generation algorithm have a strong influence on its performance
- Random$^+$ testing is predictable in the relative number of faults it uncovers over a particular timeout, but different runs of the algorithm uncover different faults
- Random$^+$ testing finds around 66% of faults in contracts and the rest in implementation, while for manual testing the situation is reversed
- Random$^+$ testing finds a much higher number of faults than manual testing and than users of the software, but it does not find all faults revealed through manual testing or reported by the users

This thesis thus brings contributions in a few of the directions of current research, in an attempt to advance the state-of-the-art in software testing and to contribute to the ultimate goal in software engineering: to enable the faster and easier development of better quality software.

APPENDIX A

EXAMPLE OF CONTRACT INFERENCE

Listing A.1 shows an example illustrating the assertions that CITADEL, our Eiffel front-end for the Daikon tool, infers for class *ST_SPLITTER* from the Gobo library, based on a test suite combining random and partition testing and executing each routine of the class around 50 times. All CITADEL-inferred assertions have the `inferred` tag. The original contracts of the class are also shown. The listing shows *all* inferred contracts, including the incorrect and uninteresting ones. For brevity, the listing only shows some of the routines of class *ST_SPLITTER*.

```
indexing

  description: "Split a string into tokens"
  library: "Gobo Eiffel String Library"
  copyright: "Copyright (c) 2004, Eric Bezault and others"
  license: "MIT License"
  date: "$Date: 2007-01-26 10:55:25 -0800 (Fri, 26 Jan
     2007) $"
  revision: "$Revision: 5877 $"

class ST_SPLITTER

inherit

  ANY
  KL_IMPORTED_STRING_ROUTINES
    export
      {NONE} all
    end
```

```
create

  make,
  make_with_separators

feature {NONE}
  -- Initialization
  make is
      -- Create a new string splitter.
    do
      set_separators (Default_separators)
    ensure
      inferred: separators = Default_separators
      inferred: not has_escape_character
      inferred: escape_character = 0
      default_separators: separators = Default_separators
      no_escape_character: not has_escape_character
    end

  make_with_separators (a_string: STRING) is
      -- Create a new string splitter with separators
         specified in 'a_string'.
    require
      inferred: a_string /= Void
      inferred: not a_string.is_empty
      a_string_not_void: a_string /= Void
      a_string_not_empty: not a_string.is_empty
    do
      set_separators (a_string)
    ensure
      inferred: separators = a_string
      inferred: has_escape_character = a_string.is_empty
      inferred: has_escape_character = old a_string.is_empty
      inferred: a_string.count = old a_string.count
      inferred: not has_escape_character
      inferred: escape_character = 0
      inferred: a_string.is_equal (old a_string.string)
      separators_set: separators = a_string
      no_escape_character: not has_escape_character
    end

feature
```

```
  -- Access
  separators: STRING
     -- Characters used as token separators

  has_escape_character: BOOLEAN
     -- Is there an escape character set?

  escape_character: CHARACTER
     -- Escape character
     -- (When in an input string, it is removed and
        replaced
     -- with the character following it. This following
     -- character is not treated as a separator.)

  Default_separators: STRING is " %T%R%N"
     -- Space, Tab, CR, Newline

feature
  -- Setting
  set_separators (a_string: STRING) is
     -- Set characters used as separators within string.
    require
      inferred: a_string /= Void
      inferred: not a_string.is_empty
      inferred: a_string.count >= 1
      a_string_not_void: a_string /= Void
      a_string_not_empty: not a_string.is_empty
      escape_character_not_separator: has_escape_character
         implies not a_string.has (escape_character)
    local
      i, nb: INTEGER
    do
      separators := a_string
        -- Initialize codes hash set from separators.
      nb := a_string.count
      create separator_codes.make (nb)
      from
        i := 1
      invariant
        inferred: separators = a_string
        inferred: a_string.count = nb
        inferred: separators /= Void
        inferred: escape_character >= 0
        inferred: Default_separators /= Void
        inferred: separator_codes /= Void
        inferred: not a_string.is_empty
        inferred: i >= 1
      until
        i > nb
      loop
        separator_codes.put (a_string.item_code (i))
        i := i + 1
      end
    ensure
      inferred: separators = a_string
      inferred: has_escape_character = old
         has_escape_character
      inferred: escape_character = old escape_character
      inferred: Default_separators = old Default_separators
      inferred: a_string.is_empty = old a_string.is_empty
      inferred: a_string.count = old a_string.count
      inferred: not a_string.is_empty
      inferred: a_string.count >= 1
      inferred: a_string.is_equal (old a_string.string)
      separators_set: separators = a_string
    end

  set_escape_character (a_character: CHARACTER) is
      -- Set escape character.
    require
      escape_character_not_separator: not separators.has (
         a_character)
    do
      escape_character := a_character
      has_escape_character := True
    ensure
      inferred: separators = old separators
      inferred: escape_character = a_character
      inferred: Default_separators = old Default_separators
      inferred: has_escape_character
      has_escape_character: has_escape_character
      escape_character_set: escape_character = a_character
    end

feature
  -- Operation(s)
```

```
split (a_string: STRING): DS_LIST [STRING] is
    -- Split a string according to separator
    -- and escape character settings.
    -- A sequence of separators is a single separator,
    -- a separator at start/end of string is ignored.
  require
    inferred: a_string /= Void
    inferred: a_string.count >= 0
    a_string_not_void: a_string /= Void
  do
    Result := do_split (a_string, False)
  ensure
    inferred: has_escape_character = old
       has_escape_character
    inferred: escape_character = old escape_character
    inferred: Default_separators = old Default_separators
    inferred: Result.off = Result.before
    inferred: Result.after = Result.is_first
    inferred: Result.after = Result.is_last
    inferred: Result /= Void
    inferred: Result.index = 0
    inferred: Result.new_cursor /= Void
    inferred: Result.cloned_object /= Void
    inferred: Result.count >= 0
    inferred: Result.off
    inferred: Result.equality_tester = Void
    inferred: Result.to_array /= Void
    inferred: not Result.after
    inferred: (Result /= Void) and then (not Result.
       is_empty or else not Result.is_empty) implies
       Result.first /= Void
    inferred: (Result /= Void) and then (not Result.
       is_empty or else not Result.is_empty) implies
       Result.last /= Void
    inferred: a_string.is_equal (old a_string.string)
    inferred: Result.index <= Result.count
    inferred: Result.index <= old a_string.count
    inferred: Result.count <= old a_string.count
    separators_as_sequence: split (separators).is_empty
    split_not_void: Result /= Void
    no_void_item: not Result.has (Void)
    no_empty_item: not has_empty (Result)
    count_ceiling: Result.count <= a_string.count // 2 + 1
```

```
      last_escape_character_verbatim: (a_string.count >= 2
         and then a_string.item (a_string.count) =
         escape_character and then a_string.item (a_string.
         count - 1) /= escape_character) implies (Result.
         last.item (Result.last.count) = escape_character)
    end

  join (a_linear: DS_LINEAR [STRING]): STRING is
      -- Join sequence to a string using the first of the
      -- 'separators' as separator, and escape separators
      -- within tokens.
    require
      inferred: a_linear.is_empty = a_linear.off
      inferred: has_escape_character
      inferred: a_linear /= Void
      inferred: (a_linear /= Void) and then not a_linear.
         is_empty implies a_linear.first /= Void
      inferred: a_linear.new_cursor /= Void
      inferred: not a_linear.after
      inferred: a_linear.to_array /= Void
      inferred: a_linear.cloned_object /= Void
      inferred: a_linear.count >= 0
      inferred: (a_linear /= Void) and then not a_linear.off
          implies a_linear.item_for_iteration /= Void
      has_escape_character: has_escape_character
      a_linear_not_void: a_linear /= Void
      no_void_item: not a_linear.has (Void)
      no_empty_item: not has_empty (a_linear)
    do
      Result := do_join (a_linear, False)
    ensure
      inferred: has_escape_character = old
         has_escape_character
      inferred: escape_character = old escape_character
      inferred: Default_separators = old Default_separators
      inferred: Result.is_empty = old a_linear.is_empty
      inferred: Result.is_empty = old a_linear.off
      inferred: has_escape_character
      inferred: Result /= Void
      inferred: Result.count >= 0
      inferred: Result.count >= old a_linear.count
      join_not_void: Result /= Void
      same_count: split (Result).count = a_linear.count
      stable_reversible: STRING_.same_string (join (split (
         Result)), Result)
    end

feature {NONE}
  -- Implementation
  separator_codes: DS_HASH_SET [INTEGER]
      -- Character codes of separators
      -- (Hashed, and integer for unicode compatibility.)

  do_join (a_linear: DS_LINEAR [STRING]; a_greedy: BOOLEAN)
      : STRING is
      -- Join sequence to a string using the first of the
      -- 'separators' as separator, and escape separators
      -- within tokens.
    require
      inferred: a_linear.is_empty = a_linear.off
      inferred: separators /= Void
      inferred: has_escape_character
      inferred: Default_separators /= Void
      inferred: separator_codes /= Void
      inferred: a_linear /= Void
      inferred: (a_linear /= Void) and then not a_linear.
         is_empty implies a_linear.first /= Void
      inferred: a_linear.new_cursor /= Void
      inferred: not a_linear.after
      inferred: a_linear.to_array /= Void
      inferred: a_linear.cloned_object /= Void
      inferred: a_linear.count >= 0
      inferred: (a_linear /= Void) and then not a_linear.off
          implies a_linear.item_for_iteration /= Void
      has_escape_character: has_escape_character
      a_linear_not_void: a_linear /= Void
      no_void_item: not a_linear.has (Void)
    local
      a_cursor: DS_LINEAR_CURSOR [STRING]
      a_separator: STRING
    do
      create Result.make_empty
        -- Using a string for separator is unicode compatible
            .
      a_separator := separators.substring (1, 1)
      a_cursor := a_linear.new_cursor
      from
        a_cursor.start
      invariant
        inferred: a_cursor.container = a_linear
        inferred: a_cursor.after = a_cursor.off
        inferred: a_linear.after = a_separator.is_empty
        inferred: a_linear.is_empty = a_linear.off
        inferred: separators /= Void
        inferred: has_escape_character
        inferred: Default_separators /= Void
        inferred: separator_codes /= Void
        inferred: a_cursor /= Void
        inferred: a_cursor.container /= Void
        inferred: (a_cursor /= Void) and then not a_cursor.
            off implies a_cursor.item /= Void
        inferred: (a_linear /= Void) and then not a_linear.
            is_empty implies a_linear.first /= Void
        inferred: a_linear.new_cursor /= Void
        inferred: a_linear.to_array /= Void
        inferred: a_linear.cloned_object /= Void
        inferred: a_linear.count >= 0
        inferred: (a_linear /= Void) and then not a_linear.
            off implies a_linear.item_for_iteration /= Void
        inferred: a_separator /= Void
        inferred: not a_separator.is_empty
        inferred: a_separator.count = 1
        inferred: Result /= Void
        inferred: Result.count >= 0
      until
        a_cursor.after
      loop
        Result := escape_appended_string (Result, a_cursor.
            item)
        a_cursor.forth
        if not a_cursor.after then
          Result := STRING_.appended_string (Result,
              a_separator)
        end
      end
    ensure
      inferred: has_escape_character = old
          has_escape_character
      inferred: escape_character = old escape_character
      inferred: Default_separators = old Default_separators
      inferred: Result.is_empty = old a_linear.is_empty
      inferred: Result.is_empty = old a_linear.off
      inferred: separators /= Void
      inferred: has_escape_character
      inferred: Default_separators /= Void
      inferred: separator_codes /= Void
      inferred: Result /= Void
      inferred: Result.count >= 0
      join_not_void: Result /= Void
    end

...

invariant

  inferred: separators /= Void
  inferred: escape_character >= 0
  inferred: Default_separators /= Void
  inferred: separator_codes /= Void
  separators_not_void: separators /= Void
  separators_not_empty: not separators.is_empty
  escape_character_not_separator: has_escape_character
      implies not separators.has (escape_character)

end
```

Listing A.1: Class ST_SPLITTER, annotated with inferred assertions.

BIBLIOGRAPHY

[1] CppUnit. `http://sourceforge.net/projects/cppunit`.

[2] EiffelStudio. Eiffel Software. `http://www.eiffel.com/`.

[3] Gobo Eiffel Test. `http://www.gobosoft.com/eiffel/gobo/getest/index.html`.

[4] Jtest. Parasoft Corporation. `http://www.parasoft.com/`.

[5] JUnit. `http://www.junit.org/`.

[6] PyUnit. `http://pyunit.sourceforge.net/`.

[7] SUnit. `http://sunit.sourceforge.net/`.

[8] The EiffelBase Library. Eiffel Software Inc. `http://www.eiffel.com/`.

[9] vbUnit. `http://www.vbunit.com/`.

[10] IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-199), September 1990.

[11] IEEE Standard Classification for Software Anomalies (IEEE Std 1044-1993), June 1994.

[12] E. Allen. *Bug Patterns in Java*. APress L. P., 2002.

[13] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.

[14] J. H. Andrews, S. Haldar, Y. Lei, and C. H. Li. Randomized unit testing: Tool support and best practices. Technical Report 663, Department of Computer Science, University of Western Ontario, January 2006.

[15] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.*, 32(8):608–624, 2006.

[16] James H. Andrews, Susmita Haldar, Yong Lei, and Felix Chun Hang Li. Tool support for randomized unit testing. In *RT '06: Proceedings of the 1st international workshop on Random testing*, pages 36–45, New York, NY, USA, 2006. ACM Press.

[17] Shay Artzi, Sunghun Kim, and Michael D. Ernst. ReCrash: Making software failures reproducible by preserving object states. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, Paphos, Cyprus, July 9–11, 2008.

[18] N. Ayewah, W. Pugh, D. Morgenthaler, J. Penix, and Y. Zhou. Using FindBugs on production software. In *OOPSLA companion*, pages 805–806, 2007.

[19] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. *SIGPLAN Not.*, 36(5):203–213, 2001.

[20] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer, 2004.

[21] V. Basili and R. Selby. Comparing the effectiveness of software testing strategies. *IEEE TSE*, 13(12):1278–1296, 1987.

[22] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):73–96, 2005.

[23] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[24] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04, Edinburgh)*, pages 326–335. IEEE Computer Society Press, 2004.

[25] Robert V. Binder. *Testing Object-Oriented Systems. Models, Patterns, and Tools*. Addison-Wesley, 1999.

[26] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180, New York, NY, USA, 2006. ACM Press.

[27] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the 2002 ACM*

SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), Rome, Italy, 2002.

[28] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to support fault isolation in object-oriented code. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 70–80, New York, NY, USA, 2002. ACM Press.

[29] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Stress testing real-time systems with genetic algorithms. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1021–1028, New York, NY, USA, 2005. ACM.

[30] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[31] Patrice Chalin. Are practitioners writing contracts? In *Springer LNCS 4157*, pages 100–113, 2006.

[32] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Restricted random testing. In *Proceedings of the 7th International Conference on Software Quality*, pages 321 – 330. Springer-Verlag, London, UK, 2002.

[33] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with ADL. *SIGSOFT Softw. Eng. Notes*, 21(3):62–70, 1996.

[34] T. Y. Chen, F. C. Kuo, R. G. Merkel, and S. P. Ng. Mirror adaptive random testing. In *Proceedings of the Third International Conference on Quality Software*, pages 4 – 11, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[35] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In Michael J. Maher, editor, *Advances in Computer Science - ASIAN 2004: Higher-Level Decision Making. 9th Asian Computing Science Conference. Proceedings*. Springer-Verlag GmbH, 2004.

[36] T. Y. Chen and Y. T. Yu. On the relationship between partition and random testing. *IEEE Transactions on Software Engineering*, 20(12):977–980, 1994.

[37] Tsong Yueh Chen and Robert Merkel. Quasi-random testing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 309–312, New York, NY, USA, 2005. ACM Press.

[38] T.Y. Chen, R. Merkel, P.K. Wong, and G. Eddy. Adaptive random testing through dynamic partitioning. In *Proceedings of the Fourth International Conference on Quality Software*, pages 79 – 86, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[39] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 231–255, London, UK, 2002. Springer-Verlag.

[40] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.-Y. Wong. Orthogonal defect classification - a concept for in-process measurements. *IEEE TSE*, 18(11):943–956, 1992.

[41] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *RT '06: Proceedings of the 1st international workshop on Random testing*, pages 55–63, New York, NY, USA, 2006. ACM Press.

[42] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*, 2007.

[43] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. ARTOO: adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 71–80, 2008.

[44] Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, and Alexander Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. to appear in Proceedings of ISSRE'08: The 19th IEEE International Symposium on Software Reliability Engineering, November 2008.

[45] Ilinca Ciupa, Alexander Pretschner, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. On the predictability of random tests for object-oriented software. In *Proceedings of the First International Conference on Software Testing, Verification and Validation (ICST'08)*, April 2008.

[46] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[47] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[48] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 422–431. ACM, May 2005.

[49] Christoph Csallner and Yannis Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 245–254. ACM, July 2006.

[50] Marcelo d'Amorim, Carlos Pacheco, Darko Marinov, Tao Xie, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE 2006: Proceedings of the 21st Annual International Conference on Automated Software Engineering*, Tokyo, Japan, September 20–22, 2006.

[51] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[52] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, London, UK, 1993. Springer-Verlag.

[53] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10:438 – 444, July 1984.

[54] Stephen H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11(2):97–111, May 2001.

[55] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[56] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proc 17th Intl. Symp. on Parallel and Distributed Processing*, page 286.2, 2003.

[57] Long Fei and Samuel P. Midkiff. Artemis: practical runtime monitoring of applications for execution anomalies. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 84–95, New York, NY, USA, 2006. ACM.

[58] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[59] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[60] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *WSS'00: Proceedings of the 4th conference on USENIX Windows Systems Symposium*, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association.

[61] P. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proc. FSE*, pages 153–162, 1998.

[62] P. Frankl and S. Weiss. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, 1993.

[63] P. Frankl, S. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. Technical Report PUCS-100-94, Department of Computer Science, Polytechnic University, Brooklyn, NY, February 1994.

[64] M. Girgis and M. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Proceedings of the IEEE/ACM Workshop on Software Testing*, pages 64–73, July 1986.

[65] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press.

[66] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[67] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 255–265, New York, NY, USA, 2006. ACM Press.

[68] N. Gupta, A.P. Mathur, and M.L. Soffa. Generating test data for branch coverage. *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 219–227, 2000.

[69] Neelam Gupta and Zachary V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *ASE 2003: Proceedings of the 18th Annual International Conference on Automated Software Engineering*, pages 49–58, Montreal, Canada, October 8–10, 2003.

[70] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. *SIGSOFT Softw. Eng. Notes*, 23(6):231–244, 1998.

[71] W. Gutjahr. Partition testing versus random testing: the influence of uncertainty. *IEEE TSE*, 25(5):661–674, 1999.

[72] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 69–82, New York, NY, USA, 2002. ACM.

[73] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16 (12):1402–1411, December 1990.

[74] Dick Hamlet. When only random testing will do. In *Proc. 1st Intl. Workshop on Random Testing*, pages 1–9, 2006.

[75] R. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[76] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM.

[77] M. Harman, C. Fox, R. Hierons, Lin Hu, S. Danicic, and J. Wegener. VADA: a transformation-based system for variable dependence analysis. *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*, pages 55–64, 2002.

[78] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, January 2004.

[79] Mark Harman and Phil McMinn. A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 73–83, New York, NY, USA, 2007. ACM.

[80] M. Heimdahl, D. George, and R. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *Proc. 8th IEEE High Assurance in Systems Engineering Workshop*, February 2004.

[81] Johannes Henkel and Amer Diwan. Discovering algebraic specifications from Java classes. In Luca Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, Darmstadt, July 2003. Springer.

[82] Johannes Henkel and Amer Diwan. A tool for writing and debugging algebraic specifications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 449–458, Washington, DC, USA, 2004. IEEE Computer Society.

[83] K. Henningsson and C. Wohlin. Assuring fault classification agreement – an empirical evaluation. In *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 95–104, 2004.

[84] R.M. Hierons. Testing from a Z specification. *The Journal of Software Testing, Verification, and Reliability*, 7(1):19–33, 1997.

[85] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[86] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[87] B.F. Jones, H.-H. Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, Sep 1996.

[88] E. Kamsties and C. Lott. An empirical evaluation of three defect-detection techniques. In *Proc. ESEC*, pages 362–383, 1995.

[89] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume LNCS 2619, pages 553–568. Springer-Verlag, 2003.

[90] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[91] D. E. Knuth. The errors of TEX. *Softw. Pract. Exper.*, 19(7):607–685, 1989.

[92] B. Korel. A dynamic approach of test data generation. *Software Maintenance, 1990., Proceedings., Conference on*, pages 311–317, Nov 1990.

[93] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 83–93, New York, NY, USA, 2004. ACM.

[94] Ted Kremenek and Dawson R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *SAS*, pages 295–315, 2003.

[95] Yves Le Traon, Benoit Baudry, and Jean-Marc Jézéquel. Design by Contract to improve software vigilance. *IEEE Transactions of Software Engineering*, 32(8):571–586, 2006.

[96] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[97] A. Leitner and I. Ciupa. AutoTest. `http://se.inf.ethz.ch/people/leitner/auto_test/`, 2005 - 2008.

[98] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, November 2007.

[99] Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, and Mark Howard. Reconciling manual and automated testing: the AutoTest experience. In *Proceedings of the 40th Hawaii International Conference on System Sciences - 2007, Software Technology*, January 3-6 2007.

[100] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arnaud Fiva. Contract driven development = test driven development - writing test cases. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, September 2007.

[101] Andreas Leitner, Patrick Eugster, Manuel Oriol, and Ilinca Ciupa. Reflecting on an existing programming language. In *Proceedings of TOOLS EUROPE 2007 - Objects, Models, Components, Patterns,*, July 2007.

[102] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR*, 163(4):845–848, 1965.

[103] R. Lutz. Targeting safety-related errors during software requirements analysis. In *Proc. ACM SIGSOFT FSE*, pages 99–106, 1993.

[104] S. Mankefors, R. Torkar, and A. Boklund. New quality estimations in random testing. page 468, 2003.

[105] Johannes Mayer. Adaptive random testing by bisection with restriction. In *Proceedings of the Seventh International Conference on Formal Engineering Methods (ICFEM 2005)*, LNCS 3785, pages 251–263. Springer-Verlag, Berlin, 2005.

[106] Johannes Mayer. Lattice-based adaptive random testing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, ACM, pages 333–336. ACM Press, New York, NY, USA, 2005.

[107] Bertrand Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, 1992.

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

[109] Bertrand Meyer. Attached types and their application to three open problems of object-oriented programming. In Andrew Black, editor, *Proceedings of European Conference on Object-Oriented Programming, Edinburgh, 25-29 July 2005*, volume Lecture Notes in Computer Science 3586, pages 1–32. Springer Verlag, 2005.

[110] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, 27(12):1085–1110, 2001.

[111] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[112] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Softw. Eng.*, 2(3):223–226, 1976.

[113] Cosmin Mitran. Guided random-based testing strategies. Master's thesis, Technical University of Cluj-Napoca, 2007.

[114] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[115] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[116] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: an empirical evaluation. *SIGSOFT Software Engineering Notes*, 27(6):11–20, 2002.

[117] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, 1997.

[118] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, 1992.

[119] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation. *Softw. Pract. Exper.*, 29(2):167–193, 1999.

[120] A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, October 1999.

[121] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *The Journal of Software Testing, Verification and Reliability*, 13(1):25–53, March 2003.

[122] Catherine Oriat. Jartege: a tool for random generation of unit tests for Java classes. Technical Report RR-1069-I, Centre National de la Recherche Scientifique, Institut National Polytechnique de Grenoble, Universite Joseph Fourier Grenoble I, June 2004.

[123] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, 1988.

[124] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, Scotland, July 25–29, 2005.

[125] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 23–25, 2007.

[126] Nadezda Polikarpova. Dynamic assertion inference in a programming language with Design-by-Contract support (Eiffel case study). Master's thesis, The Saint-Petersburg State University of Information Technologies, Mechanics and Optics (SPbSU ITMO), 2008.

[127] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proc. Intl. Conf. on Software Engineering*, pages 392–401, 2005.

[128] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *Proceedings of the Workshop on Automated and Algorithmic Debugging 2003*, 2003.

[129] D. Richardson, O. O'Malley, and C. Tittle. Approaches to specification-based testing. In *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, pages 86–96, New York, NY, USA, 1989. ACM Press.

[130] M. Roper. Computer aided software testing using genetic algorithms. *10th International Software Quality Week*, 1997.

[131] Koushik Sen and Gul Agha. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. pages 419–423. 2006.

[132] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

[133] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, Planning Report 02-3*, 2002.

[134] Nikolai Tillmann, Feng Chen, and Wolfram Schulte. Discovering likely method specifications. In *ICFEM*, pages 717–736, 2006.

[135] Nikolai Tillmann and Wolfram Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, 2006.

[136] Paolo Tonella. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 119–128, New York, NY, USA, 2004. ACM Press.

[137] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*, page 285, Washington, DC, USA, 1998. IEEE Computer Society.

[138] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, 2003.

[139] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, New York, NY, USA, 2004. ACM Press.

[140] J.M. Voas and K.W. Miller. Software testability: the new verification. *Software, IEEE*, 12(3):17–28, May 1995.

[141] Joachim Wegener and Matthias Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Syst.*, 15(3):275–298, 1998.

[142] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Trans. Softw. Eng.*, 20(5):353–363, 1994.

[143] E.J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, Jul 1991.

[144] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4):218–228, 2002.

[145] M. Wood, M. Roper, A. Brooks, and J. Miller. Comparing and combining software defect detection techniques: a replicated empirical study. *SIGSOFT Softw. Eng. Notes*, 22(6):262–277, 1997.

[146] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pages 365–381, April 2005.

[147] Tao Xie and David Notkin. Tool-assisted unit test selection based on operational violations. *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 40–48, Oct. 2003.

[148] Y. Xie and D. Engler. Using redundancies to find errors. *IEEE Transactions on Software Engineering*, 29(10):915–928, October 2003.

[149] Hai Yuan and Tao Xie. Substra: A framework for automatic generation of integration tests. In *1st Workshop on Automation of Software Test (AST 2006)*, pages 64–70, Shanghai, China, May 2006.

[150] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TSE*, SE-28(2):183–202, February 2002.

CURRICULUM VITAE

General information

Name: Ilinca Ciupa
Date of birth: November 11, 1980
Nationality: Romanian
Web page: `http://se.inf.ethz.ch/people/ciupa`
E-mail address: Ilinca.Ciupa@inf.ethz.ch

Education and career history

- July 2005 - December 2008: PhD student at the Chair of Software Engineering, ETH Zurich (Swiss Federal Institute of Technology Zurich)

- October 2004 - July 2005: academic guest at the Chair of Software Engineering, ETH Zurich

- September 1999 - September 2004: undergraduate student at the Technical University of Cluj-Napoca, Romania

Certificates and diplomas

- Dipl. Eng., Technical University of Cluj-Napoca, Romania (2004)

- Baccalaureate Diploma (1999)

- Cambridge Certificate of Proficiency in English (1998)

Selected publications

- **Ciupa, I.**, Meyer, B., Oriol, M., Pretschner, A.,"Finding Faults: Manual Testing vs. Random+ Testing vs. User Reports", Proceedings of ISSRE'08: The 19th IEEE International Symposium on Software Reliability Engineering, (Redmond, Seattle, USA), November 2008

- **Ciupa, I.**, Leitner, A., Oriol, M., Meyer, B., “ARTOO: Adaptive Random Testing for Object-Oriented Software”, Proceedings of ICSE’08: International Conference on Software Engineering 2008, (Leipzig, Germany), May 2008

- **Ciupa, I.**, Pretschner, A., Leitner, A., Oriol, M., Meyer, B., “On the Predictability of Random Tests for Object-Oriented Software”, Proceedings of ICST’08: IEEE International Conference on Software Testing, Verification and Validation 2008, (Lillehammer, Norway), April 2008 (**Best paper award**)

- **Ciupa, I.**, Leitner, A., Oriol, M., Meyer, B., “Experimental Assessment of Random Testing for Object-Oriented Software”, Proceedings of ISSTA’07: International Symposium on Software Testing and Analysis 2007, (London, UK), July 2007

- **Ciupa, I.**, Leitner, A., Oriol, M., Meyer, B., “Object Distance and Its Application to Adaptive Random Testing of Object-Oriented Programs”, to appear in Proceedings of the First International Workshop on Random Testing (RT 2006)

Languages

- Romanian — native
- English — advanced
- German — intermediate
- Italian — beginner