Doctoral Thesis

Contract-based tests in the software process and environment

Author[s]:
Leitner, Andreas E.

Publication Date:
2008

Permanent Link:
https://doi.org/10.3929/ethz-a-005787922

Rights / License:
In Copyright - Non-Commercial Use Permitted
Contract-based tests in the software process and environment

A dissertation submitted to the
ETH Zurich

for the degree of
Doctor of Sciences

presented by
Andreas Leitner
Diplom Ingenieur, TU Graz

born
January 16th, 1977

citizen of
Austria

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. David Basin, co-examiner
Prof. Dr. Roderick Bloem, co-examiner
Prof. Dr. Andreas Zeller, co-examiner

2008
ABSTRACT

In many projects, software testing comes as an afterthought. Recent development methods try to break this habit by placing testing at the center of development. This heightened role of tests raises new problems: the development process must interleave testing with other activities; developers must prepare more tests; these tests must be executed continuously and hence must be short. Contracts in the form of preconditions, postconditions and invariants help to automate testing and relieve developers: preconditions filter invalid inputs, postconditions serve as oracles. This thesis presents the following contract-based testing techniques:

- **Test cases can be extracted**: contract violations are made reproducible via persistent and quickly executing unit tests. These extracted tests aid debugging and later on regression testing. Unlike capture and replay techniques, our novel extraction mechanism has no run-time overhead.

- **Test cases can be synthesized**: a strategy for automatically generating inputs combined with a contract-based oracle enables fully automated testing. Even a random-based synthesis strategy, simple but widely applicable, finds a variety of faults in production quality source code.

Test cases created by a random-based strategy can grow very large and become difficult to debug, hence they should be minimized. We have developed a combination of slicing and delta debugging which significantly minimizes the generated test cases. We also present auxiliary methods concerning test case selection, prioritization and reflection.

We have implemented these techniques in two tools: **CDD EiffelStudio**, a test case extractor, which is integrated in the EiffelStudio development environment and **AutoTest**, a test case synthesizer, executor, and minimizer. This thesis includes empirical evaluations of both tools to prove their effectiveness and efficiency.
ZUSAMMENFASSUNG


- Tests können *synthetisiert* werden: Eine Strategie zum automatischen Erzeugen von Inputs kombiniert mit vertragsbasiertem Testen automatisiert das Testen vollständig. Sogar zufallsbasierte Synthesestrategien, welche einfach und leicht einzusetzen sind, finden eine Vielfalt von Fehlern in industriellen Quellcode.


Wir haben diese Techniken in zwei Werzeugen implementiert: CDD *EiffelStudio*, ein Testfallextrahierer, der in die EiffelStudio Entwicklungsumgebung eingebettet ist und *AutoTest*, ein Werkzeug zum synthetisieren, ausführen, und verklei-
nern von Tests. Empirische Evaluierungen bezeugen die Effektivität und Effizienz dieser Werkzeuge.
ACKNOWLEDGEMENTS

This thesis marks the end of a long and eventful journey during which I have learned so much and met so many wonderful people who, in different ways, contributed not only to my work but also to my personal growth.

It is difficult to overstate my gratitude to my supervisor, Bertrand Meyer. His enthusiasm, vision, and sense for clarity were always inspirational.

I am also thankful to the other members of my committee. I would like to thank Andreas Zeller, who is a great teacher, and source for many inspiring discussions. I aspire his drive for clarity. I would like to thank Roderick Bloem, with whom I had many insightful and challenging discussions. His sharpness and detailed feedback were of great importance to me. I would also like to thank David Basin, for discussions and his detailed feedback, which led to a (hopefully) much better thesis.

I am grateful to my colleague, Ilinca Ciupa, a steady companion during this journey. She helped me in uncountable many discussions. I learned much by working with her at so many occasions. I would also like to thank my other colleagues and collaborators with whom I enjoyed to work and to discuss: Bernd Schoeller, Vijay D’silva, Patrick Eugster, Manuel Oriol, Michela Pedroni, Piotr Nienaltowski, Mark Howard, Till Bay, Lisa Liu, Ádám Darvas, Martin Nordio, Hermann Lehner, Werner Dietl, Martin Burger, and Valentin Dallmeier.

I owe much to Alexander Pretschner. He was a great teacher, colleague and friend to me. I am grateful for the many challenging and startling discussion.

I am happy to have had inspiring and fruitful discussions with Michael Ernst, Robert Binder, and Jeff Offutt.

Many members of the Eiffel community contributed with their support and the software they wrote. I aspire the development skills of Eric Bezault. He was a great teacher to me. I would like to thank the engineers at EiffelSoftware, formost Manu Stapf and Jocelyn Fiat, who never failed to support me.

I would like to thank the students who I had the pleasure to work with: Arno Fiva, Stefan Mori, and Stefan Sieber.

I owe this thesis to Ilaria Galli, my girlfriend, who always stood by me and supported me in every conceivable way. I would also like to thank my family,
foremost my dad for being a role model and nurturing my interest in science in
general and software engineering in particular. Without him none of this would
exist.
# Contents

## 1 Summary and main results
1.1 The purpose of testing ............................................. 2
1.2 Testing in the development process .............................. 3
1.3 Problems and challenges ........................................... 5
1.4 Contract-based testing ............................................. 6
1.5 Main contributions .................................................. 8
   1.5.1 Leveraging existing tests .................................... 8
   1.5.2 Creating new tests ........................................... 12
   1.5.3 Running tests via reflection ................................. 13
1.6 Outline ................................................................ 13

## 2 Preliminaries
2.1 Eiffel & Eiffel terminology ........................................ 15
2.2 Design By Contract ................................................... 17
2.3 Faults, failures, and mistakes ...................................... 19
2.4 Test outcomes .......................................................... 20
2.5 Specification-based testing ......................................... 22
2.6 Contract-based testing .............................................. 23
   2.6.1 Testing with incomplete specification ...................... 25
2.7 Resilient execution ................................................... 26
2.8 Slicing .................................................................. 28

## 3 Related work
3.1 Automating test execution .......................................... 31
3.2 Leveraging existing tests ........................................... 32
   3.2.1 Reproducing failures ......................................... 32
   3.2.2 Selecting and prioritizing tests ............................ 33
3.3 Creating new tests .................................................... 34
   3.3.1 Test case generation ......................................... 34
   3.3.2 Test minimization ............................................ 36
4 Extracting tests

4.1 Overview ............................................. 41
4.2 Motivation ........................................... 43
4.3 A use case ............................................ 44
4.4 Model ................................................. 48
   4.4.1 The called-by tree ............................ 48
   4.4.2 Test cases ....................................... 52
   4.4.3 Further applications of test case extraction .......................... 55
   4.4.4 Invocation-state extraction .................... 57
   4.4.5 Failure-state extraction ....................... 57
4.5 Implementation ...................................... 58
4.6 Evaluation ........................................... 63
   4.6.1 Experimental setup ............................. 63
   4.6.2 Failure-state extraction without depth limit ...................... 66
   4.6.3 Invocation-state extraction .................... 69
   4.6.4 Contract vs. other runtime failures ....................... 70
   4.6.5 Unique failures ................................. 70
   4.6.6 Interpretation and consequences ................. 72
   4.6.7 Threats to validity ............................. 73

5 Selecting and prioritizing tests ........................................... 75

5.1 Intuition of test selection .................................. 76
5.2 Test selection algorithm .................................. 78
5.3 From test selection to test prioritization ....................... 79
5.4 Evaluation ........................................... 80
   5.4.1 Setup ........................................... 80
   5.4.2 Results ........................................... 80

6 Creating new tests ........................................... 83

6.1 AutoTest framework ..................................... 84
   6.1.1 A tour of AutoTest ............................. 84
   6.1.2 AutoTest architecture .......................... 88
6.2 Unit tests ............................................ 90
6.3 Random-based test synthesis algorithm .......................... 91
6.4 Object pool ........................................... 98
6.5 Evaluation of random contract-based testing .................... 102
6.6 Minimizing tests ..................................... 103
6.7 Minimizing through slicing ................................ 104
6.8 Case study: EiffelBase library ........................................ 110
   6.8.1 Results .......................................................... 111
   6.8.2 Non-determinism .................................................. 113
   6.8.3 Discussion ....................................................... 114
6.9 Minimizing through delta debugging ................................. 114
   6.9.1 Algorithm .......................................................... 115
   6.9.2 Example ............................................................ 116
   6.9.3 Delta debugging compared ....................................... 117
6.10 A joint approach ...................................................... 117
   6.10.1 Discussion ........................................................ 119

7 Running tests via reflection .......................................... 121
   7.1 Introduction ........................................................ 121
   7.2 Reflection generation .............................................. 123
   7.3 Approach overview ................................................ 123
   7.4 A simple language .................................................. 124
   7.5 Heterogeneous approach ......................................... 126
   7.6 Homogeneous approach ........................................... 126
   7.7 The case of Eiffel .................................................. 128
      7.7.1 The Eiffel language ........................................... 128
      7.7.2 Types ............................................................ 130
      7.7.3 Subtyping ....................................................... 133
      7.7.4 The Erl-G library API ......................................... 133
   7.8 Implementation .................................................... 134
      7.8.1 Erl-G ............................................................. 134
      7.8.2 General architecture ......................................... 135
      7.8.3 Introspectable types in the heterogeneous and homo-
            geneous approaches .............................................. 135
      7.8.4 Illustrations .................................................... 136
      7.8.5 Evaluation ....................................................... 137
   7.9 Applicability ........................................................ 139
   7.10 Reflection generation in AutoTest ............................... 140
   7.11 Summary ............................................................ 141

8 Conclusions and future work ........................................ 143
   8.1 Future work .......................................................... 143
   8.2 Conclusions .......................................................... 145
CHAPTER 1

SUMMARY AND MAIN RESULTS

One of the remarkable developments in software engineering in recent years has been the rising role of tests. In the past, testing often occurred late in the software development process. It was started at a point in time when it was already apparent that the system under development was lacking quality. Improving quality late in the process is difficult and costly, because the later faults are found the harder and more expensive they are to fix. Newer approaches to software development (e.g. Test-First Development and Test-Driven Development), recognize this by placing testing to the center of development.

With this increased recognition also come new problems:

- testing must be interleaved with the software development process,
- developers must prepare more tests,
- tests must be run continuously and consequently
- tests must execute quickly.

The present work addresses some of these problems through the following novel ideas:

- Developers do test their systems, but often only in an ad-hoc and unreproducible manner. This thesis proposes a method that, whenever such a test case leads to a run-time failure (e.g. a contract violation), extracts test cases that reproduce the failure. These test cases help the developer to debug the failure (and consequently fix its underlying fault), and serve as regression test thereafter.
Contracts in the form of preconditions, postconditions and invariants enable fully automatic unit testing of source code. While the idea of contract-based test generation is not novel, this thesis presents an automatic tester in which test generation is not hardwired, enabling researchers to assess the effectiveness of various testing ideas. The present work mostly relies on a random-based strategy.

Test cases generated by a random-based synthesizer can grow large and hence become difficult to understand and costly to maintain. We present a test case minimization technique based on a combination of delta-debugging and slicing that effectively and efficiently reduces the size of generated test cases.

If test cases must be executed frequently test execution time becomes an issue. We present a means to efficiently select and prioritize the tests of a test suite according to changes made to the system under test. This allows the developer to reduce testing time by running only those tests most likely to reveal a failure.

The above ideas are embodied in two software systems: AutoTest, a test case synthesizer, executor, and minimizer and CDD EiffelStudio, an integrated development environment with a built-in test case extractor.

The rest of this chapter is structured as follows: Section 1.1 defines what testing is. Then we survey the role of testing in the development process (Section 1.2). Then we discuss some of the new challenges and limitations of current approaches (Section 1.3). Section 1.4 describes how testing benefits from the presence of contracts in source code. The next section (Section 1.5) presents the main concepts and contributions we have developed. Finally, Section 1.6 outlines the rest of this thesis.

1.1 The purpose of testing

Exactly what is the purpose of testing? A widespread believe is that testing is a means to assess the quality of software systems. For example, until the September 2, 2008 Wikipedia defined software testing as follows:\footnote{http://en.wikipedia.org/wiki/Software_testing}

"Software testing is the process of checking software, to verify that it satisfies its requirements and to detect errors."
In practice, the input space of modern software systems is too large to be tested exhaustively and testing cannot guarantee or verify the correctness of a program. In his seminal work Testing Object-Oriented Systems [16], Robert Binder is more modest in his definition for testing:

"Software testing is the execution of code using combinations of inputs and state selected to reveal bugs."

Within this thesis we assume that the purpose of testing is indeed to reveal bugs and not to measure quality.

1.2 Testing in the development process

Towards the release of a software product, testing receives more and more attention. Specialized testers try to assess the system in environments especially prepared for release testing. The testers are often aided by a test lab or virtual machines.

The way software is tested has changed. The development of software requires various activities: analysis, design, implementation, verification, and validation. Traditional software development processes like the V- or waterfall model require that these activities are strictly separated. They do not allow you to switch activities without going through a series of bureaucratic steps. Such methods work best when the software requirements are stable and activity switching is rare.

In contrast, Meyer [77] and Walden et al. [100] proposed a seamless and reversible approach where the source code is at the center of development. While other approaches purposely make it more difficult to switch activities, this approach removes as many obstacles as possible.

Agile methods, such as Extreme Programing [14], which have become increasingly popular, are similar in spirit. They strive to be lean and remove as much bureaucracy as possible. Many developers like these new methods because they perceive them as natural and close to how they would develop in an unrestricted environment. While classic development methods assume that analysis and design precede implementation, agile methods are based on the assumption that requirements can and will change at any time – in particular during implementation. Frequent changes are dangerous, however, because they are likely to introduce regression bugs. Agile methods compensate for this by requiring a thorough test suite and frequent test execution. For example, many agile methods require tests to be written before the implementation. This method (which is a method to write tests and not a development method of its own right) is called
(a) Workflow of Test-First Development

(b) Workflow of Test-Driven Development

Figure 1.1: Workflows of test-centered development practices
1.3 Problems and challenges

When testing is interleaved throughout the development process. Specialized testers can no longer be solely responsible for it: tests have to be created and run so frequently that software developers have to do much of testing themselves. For example, in Test-First Development, where every implementation goal requires unit tests to be written before the implementation, developers cannot afford to wait for a tester to write those tests: they need to write them themselves. Developers prefer writing code however, rather than test cases. Writing code and adding new features is a productive task. It immediately enhances the software system and removes obligations from the system’s list of requirements. Testing, on the other hand, is a destructive task, exactly because it does reveal failures. A test yields one of the following two outcomes:

- Some tests fail, revealing new issues.
- All tests pass.

In the first case, testing seemingly increased the developers workload: he has to fix the newly discovered issues. In the second case, testing did not add to the workload of the developer, but it did not bring the system closer to completion either. The developer may think the time spent testing was wasted.
While developers do not like to write test cases, they do test their software during development repeatedly in an incremental process: they write new code or change existing code; then invoke the program under development in a way such that they can check whether their last change had the anticipated effect. These tests are not written down and cannot be re-executed, because they are executed manually. Inputs are provided by hand and some parts of the program may have been modified to simplify testing. Re-execution is important for two reasons however:

**Debugging.** Before a defect can be fixed it has to be located. The failure is provoked repeatedly and a sequence of observations finally reveals the true defect. It occurs that a failure provoked once by a developer can not be found a second time. This happens because the developer is not aware or cannot control of the precise circumstances required to trigger the failure. (Becoming aware of these circumstances is part of the debugging process [106])

**Regression testing.** A test that revealed a failure once, remains important even after the developer has fixed the underlying fault. A developer made a mistake and introduced the fault once. He or somebody else could make the mistake again. Keeping this test ensures that such regressions are detected as early as possible.

Automatically executing test cases to a project do come with a cost however, because they maintaining. For example, changes in requirements must be reflected in the test suite.

### 1.4 Contract-based testing

Many techniques presented in this thesis rely on the presence of contracts (executable specification) in the source code. If the developer provides contracts (in the form of preconditions and postconditions) testing can be fully automated. The idea of using contracts or specification for testing is not new. Already in 1975 Goodenough et al. [50] highlight the importance of specification for testing. Several specification-based testing tools have been developed. Korat [19], generates all non-isomorphic test inputs (up to a certain bound). It requires executable specification to detect redundant inputs. TestEra [73] uses a constraint solver to find test data.

Figure 1.2 explains the idea of contract-based testing in the form of a workflow diagram. The precondition serves to filter invalid inputs and the postcondition serves as an oracle. For example, let’s say a routine (resp. method) is called with arbitrary arguments in an arbitrary state. First the precondition of the routine
1.4. CONTRACT-BASED TESTING

Figure 1.2: Principle of concept of contract-based testing. A single routine invocation is a complete test. The precondition serves to filter invalid input and the postcondition as an oracle.

is evaluated (note that this happens fully automatic because contracts are executable). If the precondition is not satisfied the chosen inputs are bad and the test is restarted with different inputs. If the precondition is satisfied the routine body gets executed. After the routine terminates, its postcondition is evaluated (again through execution). If the postcondition is satisfied the test passes; if it is violated the test failed. Whether the inputs are chosen arbitrarily or in a more systematic fashion depends on the input generator strategy and is not restricted by the idea of contract-based testing.

Contracts in the form of preconditions, postconditions and invariants are traditionally viewed as a help for documentation, design and debugging. The presence of thorough contracts in source code can push the limits of test automation significantly. While not always as strong as dedicated test case oracles, postconditions are interleaved throughout the source code and can more easily check intermediate states. In addition to that other aspects further strengthen the oracle: runtime-exceptions (such as the ones caused by a feature invocation on a Void target\(^2\)), indirectly violated precondition, and invariant violations.

Contracts are integral part of the Eiffel programming language, which provides a large existing body of annotated source code. The Eiffel language was the target

\(^2\)A feature invocation on a Void target is also referred to as a null pointer exception.
for the implementations developed and Eiffel libraries and programs were target of the experiments conducted as part of this thesis.

The applicability of the presented techniques is not limited to Eiffel: Recent research projects such as JML [21] and Spec# [13] also include contracts.

The object constraint language (OCL)\(^3\), which is part of the Unified Modeling Language (UML), can serve as a contract replacement for most languages. A weaker form of contracts is the \texttt{assert} statement. It is integral part of languages such as Java and C# and available via standard libraries for many other languages such as C and C++. While such languages do not formally provide preconditions and postconditions heuristics can still be used to infer them \cite{36}. Even when source code contains no specification annotations at all, tools can be used to infer them statically \cite{22} or dynamically \cite{45}.

1.5 Main contributions

The main contributions of this thesis can be divided into two categories: leveraging existing tests (Section 1.5.1) and creating new tests (Section 1.5.2).

1.5.1 Leveraging existing tests

The following presents two techniques that leverage existing tests: a way to extract automatically executing tests from failures triggered by playing around with a program and a way to prioritize the tests in a large test suite so that the most relevant subset of tests can be picked.

Test case extraction

Test case extractors convert ad-hoc tests, which developers execute manually, to tests that can be re-executed automatically. Every time the developer reveals a failure by playing around with the application under development, the test extractor produces a set of tests that aim to reproduce this failure. The resulting tests try to reproduce this failure by invoking the routines that were on the stack at the time of the failure. The extracted tests execute automatically and are fast. The developer can re-run them to further diagnose the problem and, once the underlying defect is fixed, use them for regression testing. Figure 1.3 shows the basic principle of how the test case extractor works.

Besides aiding debugging and contributing to the regression test suite, such extracted tests have an additional benefit: while looking to test a certain part of an application, a developer might reveal a failure that is not of concern to the part of

\(^3\)http://www.omg.org/docs/ptc/03-10-14.pdf
1.5. MAIN CONTRIBUTIONS

Figure 1.3: Workflow of test case extraction. A developer plays with the program. The program is observed by the test extractor. In the event of an exception a test cases are extracted that try to reproduce the exception by invoking the code that triggered the exception in the same state.

the application currently under test. Developers may ignore such failures because these problems distract from their current task. Focusing on such an unrelated failure bears the risk of loosing focus on their original task. Hence the unrelated failure is put aside. Once the original task is completed, a developer may have forgotten about the intercepting failure or even if he still remembers he may no longer be able to reproduce the problem.

Test case extraction is of immediate help in this scenario. The test cases extracted for “accidentally” found failures are extracted autonomously and put in the test suite; reminding the developer of the problem he put aside. Once done with the original task the developer can, now fully focused, return to these test cases and fix the intercepting problems.

The technique can also be used more consciously by the developer: as a means to create a systematic test suite, applying a process similar to Test Driven Development. The developer first creates an empty skeleton for his program, providing the necessary contracts where needed. Then he interactively invokes the skeleton with inputs that he deems interesting. These invocations will all trigger contract violations because of the missing implementation. Every contract violation results in a test case. Once sufficiently many test cases are extracted the developer can start implementing with the intend to make failing test cases pass, similar to how he proceeds when following Test Driven Development.

The test case extraction technique presented in this thesis is novel and differs from traditional Capture and Replay techniques in that it does not capture incoming events. Capture and Replay techniques are based on one or more of the following three concepts:

- Capturing snapshots of the state at checkpoints.
• Keeping event logs in between checkpoints.
• Restoring the state of a given checkpoint and re-executing the events recorded in-between.

These tools induce a run-time overhead due to their constant capturing of both events and state. Some even require specific set-up or hardware. Extracted test cases are state based only, they do not contain information about events such as user interactions or network traffic. If a failure can only be triggered if an external event (such as network traffic or user input) is simulated, a state-based test case cannot reproduce this failure. Test extraction is easier to implement and more efficient because it does not record events, but because for the same reason it is potentially less effective. We conducted an empirical evaluation suggests, that suggests that in practice the lack of event-recording does not have a significant negative effect on the number of failures reproducible.

While state-based test extraction is more efficient because it does not capture events it does induce overhead by its frequent capturing of program state. During the execution of a program, at any point in time, a number of routines are being executed. Each frame of the program stack corresponds to one such routine invocation. This stack frame (among other things) gives access to the arguments with which the routine was invoked. The test case extractor produces one test case per routine invocation that is active at the time of the failure. Each such test case consists of two components: the routine to invoke and the state from which it is invoked. The state of the routine invocation must be taken from the moment when the routine was invoked. During the execution of a program, whenever a routine is entered, the test case extractor serializes both the routine arguments from the stack and parts of the heap reachable from the current stack frame. Upon normal termination of a routine (i.e. one without failure) the state copied at routine entrance is disposed. In the event of a failure however, the test extractor uses the saved states to generate test cases. It is this copying that imposes a significant execution overhead. A more optimistic test case extractor, is possible however. The present work introduces the so called failure-state extractor, which induces zero run-time overhead: the failure-state extractor does not copy any state when routines are invoked. Instead, at the time of a failure, it captures the state visible at that moment. This approach comes with a drawback however: a routine that reproduces a failure when invoked from its invocation-state might not reproduce this failure, when invoked from its failure-state. Long running routines in particular may change a significant amount of state. The empirical evaluation we conducted to evaluate test extraction showed however that using the failure state instead of the invocation state does not lead to fewer reproducible failures.

We evaluated our approach by conducting a user study. Developers were asked
to implement programs that contained both incoming events and long running routines (the two main sources of concern of our approach). Surprisingly, 90% of all failures triggered in the study were made reproducible by at least one extracted test case. Capturing states at method entrance time causes only marginally more test cases to reproduce their respective failures (94%), but introduces a prohibitive execution overhead. This suggests that failure-state based test extraction is effective despite the lack of event-replay and the execution from the failure-state instead of the invocation state. For a developer it is primarily important whether a failure is reproducible or not, and not how many test cases reproduce it. Most routines are short, do not depend on external events, and terminate quickly. Even if one test case fails to reproduce a failure (because the routine depends on an external event or it changes a lot of state) chances are that this failure is reproduced by another test case.

We have implemented a test case extractor into the EiffelStudio IDE\textsuperscript{4}. The test case extractor that we developed as part of this thesis can be downloaded in both binary and source form from http://dev.eiffel.com/CddBranch. This extractor was also used for the user study we used to evaluate test extraction.

Selecting and prioritizing tests

When developers produce more and more tests, either manually, or with the help of tools the size of test suites will grow too. Large and long lasting projects in particular will have large and long running test suites – even if developers and tools strive for small and fast executing tests. During development, after a small change, it is important to know immediately if this change broke something: the earlier the feedback, the easier it is to react. Given the frequency of changes, running the whole test suite will often be too time-consuming. Running the whole test suite after every change is often unnecessary: most tests exercise only a small part of the application and are not affected by a change to another part. This is true in particular for unit tests.

Our test case selection and prioritization algorithms reduces the execution-time of test suites by identifying those tests that are irrelevant with regards to a certain change. Such an algorithm has to be accurate, and in order to amortize its cost, it also has to be efficient. If it is not accurate it selects the wrong test cases, if it is not efficient it mitigates its purpose. This thesis presents a novel test case selection and prioritization algorithm, that is accurate and efficient. While many other algorithms rely on complex static analysis or executions, our algorithm uses only the association and the inheritance relation. Through an optimization it can even rely on lexical analysis of the source code only.

\textsuperscript{4}EiffelStudio is an integrated development environment for the Eiffel programming language. It is available from http://www.eiffel.com
1.5.2 Creating new tests

Due to the presence of executable specification in the form of contracts, test case synthesis can be fully automated: preconditions filter invalid input and postconditions serve as oracles. Even a random-based test case synthesizer is able to reveal many faults in production quality software. While random-based testing is simple, it is easy to apply, scales well, and can be used to test small as well as large systems.

Generated test cases that reveal failures, in particular test cases stemming from a random generator, tend to be large. Large test cases are more difficult to debug and take time to re-execute (which happens frequently as part of regression testing). In this thesis we present two novel techniques to minimize the sequence of failure-inducing routine calls: the first one is based on static slicing and the second one is based on a combination of static slicing and delta-debugging. We conducted a case study on the effect of our minimization algorithms. Both algorithms minimize failing unit test cases on average by 96%. This approach improves on the state of the art by being far more efficient: in contrast to the approach of Lei and Andrews [63], who use delta debugging alone, our case study found slicing alone to be $50 \times$ faster, while providing comparable results. The minimization cost of slicing is amortized already on the first run of the test suite. The combination of slicing and delta debugging gives the best results and is $11 \times$ faster. Figure 1.4 shows the basic principle of test synthesis and minimization.

As part of this thesis we have developed the AutoTest framework which is a platform for automated contract-based testing. The tool can be downloaded both in binary and source from http://se.ethz.ch/research/autotest/. Test generation is not hardwired, allowing researchers to try out new testing ideas. The best studied strategy is based on random testing [28, 29, 32], but others based
1.6. OUTLINE

on constraint solving [71] and adaptive random testing [31] have also been developed.

1.5.3 Running tests via reflection

Many testing tools, including the ones presented here, use reflection to invoke code from the system under test. While new programming languages often directly support reflection, many old ones feature only basic or no support at all. This thesis presents a preprocessor that compensates for a lacking reflection mechanism. It allows to add reflection capabilities to a language or compiler without the need to change either.

1.6 Outline

The rest of this thesis is organized as follows. Chapter 2 explains preliminaries for this thesis. Chapter 3 gives an overview of related work. Chapter 4 explains test case extraction. Chapter 5 shows how to select and prioritize tests. Chapter 6 shows how new test cases are created and made maintainable through minimization. Chapter 7 explains how reflection can be added a posteriori to a programming language. Finally, Chapter 8 presents future work and draws conclusions.

The following lists publications that led to this thesis. Section 4 is based on our publication to the 6th joint meeting of the European Software Engineering Conference. Section 4 is based on our publication to the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE’07) [67] and our publication in the proceedings of the 40th Hawaiian International Conference on System Sciences (HICSS-40) [66]. Section 6 is based on our publication in the proceedings of SOFSEM 2007, “Current Trends in Theory and Practice of Computer Science” [79], our publication in the proceedings of the “IEEE International Conference on Software Testing, Verification and Validation 2008” (ICST’08) [32], and our publication in the proceedings of the 22nd IEEE/ACM International Conference on “Automated Software Engineering” (ASE’07) [69]. Section 7 is based on our publication in the proceedings of the 45th International Conference TOOLS Europe 2007 [68].
CHAPTER 2

PRELIMINARIES

This chapter introduces several concepts that are required to understand the rest of this thesis and gives definitions used. Section 2.1 is a brief introduction to Eiffel and explains common Eiffel terminology. Section 2.2 explains Design By Contract a concept on which many of the approaches in this thesis build. Section 2.3 defines the meaning of fault, failure and mistake. Section 2.4 defines the possible outcomes of a test execution. Section 2.5 introduces specification-based testing and Section 2.6 contract-based testing. Section 2.7 explains how sandboxing makes the execution of generated test cases resilient. Section 2.8 describes slicing.

2.1 Eiffel & Eiffel terminology

Eiffel is an object oriented language, it is statically typed, supports multiple inheritance, constrained genericity, and Design By Contract [78].

For object-oriented concepts this thesis uses the Eiffel terminology, which occasionally diverges from the conventions used in languages such as C++, Java, or C#. This thesis uses the Eiffel terminology, because most examples are expressed in Eiffel. Loryn Jenkins wrote a terminology mapping from Eiffel to C++ [3]. Table 2.1 is based on this mapping and shows the entries relevant for this thesis. This table also appeared in [65].

Listing 2.1 shows how the popular Hello World application is written in Eiffel.

- The first line declares a new class is declared (class HELLO_WORLD).
- Lines 3 and 4 declare which routines of the class can also be used as creation procedures.
### Eiffel to C++ terminology mapping

<table>
<thead>
<tr>
<th>Eiffel</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecf file</td>
<td>Makefile</td>
</tr>
<tr>
<td>An ecf file is descriptive, while a makefile is imperative</td>
<td></td>
</tr>
<tr>
<td>ancestor class</td>
<td>superclass</td>
</tr>
<tr>
<td>attribute</td>
<td>data member</td>
</tr>
<tr>
<td>check</td>
<td>assert</td>
</tr>
<tr>
<td>child class</td>
<td>derived class</td>
</tr>
<tr>
<td>cluster</td>
<td>namespace</td>
</tr>
<tr>
<td>Clusters group classes. Two classes from two different cluster must not have the same name</td>
<td></td>
</tr>
<tr>
<td>command</td>
<td>virtual function</td>
</tr>
<tr>
<td>virtual function with void return type</td>
<td></td>
</tr>
<tr>
<td>create x.make</td>
<td>x = new X</td>
</tr>
<tr>
<td>creation feature</td>
<td>constructor</td>
</tr>
<tr>
<td>deferred feature</td>
<td>pure virtual function</td>
</tr>
<tr>
<td>deferred class</td>
<td>abstract base class</td>
</tr>
<tr>
<td>descendent class</td>
<td>subclass</td>
</tr>
<tr>
<td>feature</td>
<td>function</td>
</tr>
<tr>
<td>function = method</td>
<td></td>
</tr>
<tr>
<td>feature</td>
<td>data member</td>
</tr>
<tr>
<td>method</td>
<td>method = function</td>
</tr>
<tr>
<td>fully deferred class</td>
<td>subclass</td>
</tr>
<tr>
<td>function</td>
<td>virtual function</td>
</tr>
<tr>
<td>function = method with non-void return type</td>
<td></td>
</tr>
<tr>
<td>generic class</td>
<td>class template</td>
</tr>
<tr>
<td>manifest object</td>
<td>literal constant</td>
</tr>
<tr>
<td>parent class</td>
<td>base class</td>
</tr>
<tr>
<td>Precursor</td>
<td>super()</td>
</tr>
<tr>
<td>excepting that super() is regularly used to call any feature of the superclass so desired</td>
<td></td>
</tr>
<tr>
<td>Result</td>
<td>return</td>
</tr>
<tr>
<td>agent</td>
<td>callback</td>
</tr>
<tr>
<td>agents are similar to, but more powerful than C++ callbacks</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Eiffel to C++ terminology mapping
2.2. DESIGN BY CONTRACT

Listing 2.1: Hello World written in Eiffel

- The keyword **feature** in Line 6 indicates the beginning of routine definitions.

- The first and only routine in this class is called **make**. It takes no arguments and returns nothing. It contains a header comment, describing its purpose (Line 8 and 9).

- The body of routine **make** is captured between the keywords **do** and **end** (Line 10 and 12). Routine **make** only contains one statement (Line 11): **print** ("Hello World"), which prints the string *Hello World* to the standard output stream. The routine **print** is defined in the implicitly inherited ancestor class **ANY**.

### 2.2 Design By Contract

Design By Contract [75, 78], a method and notation that allows programs to be annotated with their specification, is exploited in many of the approaches introduced in this thesis. This section gives a brief introduction into this method.

Contracts state what conditions the software must meet at certain points of the execution. They include:

- The **precondition** of a routine, stating the conditions to be established (by the callers) before any of its executions.

- The **postcondition** of a routine, stating conditions to be established (by the routine) after execution.
• The invariant of a class, stating conditions that instances of the class must fulfill upon creation (after execution of a creation procedure) and then before and after executions of exported routines.

Design By Contract can in principle be applied to any imperative language and in fact many languages support it either directly, or through 3rd party extensions:

Direct support Eiffel [74], Spec# [13], SPARK [12], D\(^1\), Chrome\(^2\), Sather [70], Lisaac \(^3\),

Extension UML through the Object Constraint Language \(^4\), C and C++ through GNU Nana \(^5\), C# through Extensible C# \(^6\), Java through the Java Modelling Language [21], Contract4J \(^7\), or jContract \(^8\), JavaScript through ecmaDebug, Lisp through the CLOS metaobject protocol, Scheme via the PLT Scheme extension, Perl via Class::Contract, Python via PyDBC, and Ruby via DesignByContract or ruby-contract

Eiffel was the first language to support Design By Contract directly. Because of the large body of annotated programs from both industry and academia we chose Eiffel as the target language for our implementations and evaluations. Listing 2.2 shows an example of an Eiffel function equipped with pre- and postcondition. The function \(sqrt\) takes a real value called \(r\) as input and requires it to be non-negative. It returns a real value (implicitly called \(Result\)) and guarantees that the square of the result is near the input value \(r\). A few more notes on Eiffel syntax and conventions: double quotes (""--) start comments; the header comment of a function (the one immediately following the feature declaration) describes the result of the function and not the calculation performed by it; assertions can be labeled, with the label appearing in front of the assertion and the colon marking the end of the label (:) as in \(r_{\text{not negative}}: r \geq 0\) where \(r_{\text{not negative}}\) is the tag for the assertion \(r \geq 0\).

The Design by Contract approach does not require a fully formal specification; contracts can be partial. According to Chalin’s analysis [23] of both public-domain and commercial software, Eiffel programmers do use contracts (accounting overall for 4.4% of the code); this makes Eiffel a particularly interesting language to target and provides a significant advantage over approaches where specifications of the software’s intent must be added before testing can proceed.

\(^1\)http://www.digitalmars.com/d/
\(^2\)http://www.remobjects.com/
\(^3\)http://isaacproject.u-strasbg.fr/li.html
\(^4\)http://www.omg.org/docs/ptc/03-10-14.pdf
\(^5\)http://www.gnu.org/software/nana/
\(^6\)http://www.resolvecorp.com/
\(^7\)http://www.contract4j.org/
\(^8\)http://www.parasoft.com
2.3. Faults, Failures, and Mistakes

sqrt (r: REAL): REAL

-- Real square-root of 'r'

require

r_not_negative: r >= 0

do

-- Implementation omitted.

ensure

definition: abs ((Result * Result) - r) <= epsilon

end

Listing 2.2: Example of Eiffel function equipped with precondition and postcondition

Contracts take the form of Boolean expressions augmented, in the case of postconditions, by old expressions to express the relationship between the final state of a routine’s execution to its initial state, as in a postcondition clause of the form counter = old counter + 1. Because boolean expressions can rely on function calls, contracts can express arbitrarily complex properties.

Contracts can also be evaluated and monitored during execution. Monitoring is normally turned off in production systems due to the incurred performance overhead. Research is also underway that targets static verification of contracts [13, 21].

2.3 Faults, failures, and mistakes

This section defines common concepts related to problems in software as they are used throughout this thesis. The nomenclature in use to describe problems in software is not uniform. The IEEE Standard Glossary of Software Engineering Terminology [2] defines the following:

fault. (1) A defect in a hardware device or component; for example, a short circuit or broken wire. (2) An incorrect step, process, or data definition in a computer program. Note: This definition is used primarily by the fault tolerance discipline. In common usage, the terms “error” and “bug” are used to express this meaning.

mistake. A human action that produces an incorrect result. Note: The fault tolerance discipline distinguishes between the human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error).
20 \textbf{CHAPTER 2. PRELIMINARIES}

\textbf{failure.} The inability of a system or component to perform its required functions within specified performance requirements. Note: The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error).

This thesis follows the definitions from the IEEE standard: a mistake of the developer leads to a fault in the code, which can lead to a failure at run time. Occasionally the word “defect” is used instead of fault. As the word error is used to mean different things this thesis avoids it.

Within this thesis, a contract violation is considered to be a failure. When programs, for which no modular specification is available are run, specification violations cannot be detected. A routine without contracts that does not behave as intended and does not have any outside visible effect does not cause a failure; its internal state is not visible to the outside. With contracted programs this changes: a violation stemming from a routine not satisfying its contract is observable (through assertion monitoring) and classified according to the above definition as a failure.

\section{2.4 Test outcomes}

As mentioned, we assume that the purpose of testing is to reveal faults [16]. What can we learn from a test execution? A passing test does obviously not prove the absence of all faults, and a failing test only proves the presence of a particular fault. A correct test case does, however, provide a proof of a specific property: when stimulated with the inputs of the test case, the implementation under test satisfies the criteria specified in the oracle of the test case.

Given a correct test case, the outcome the outcome of its execution is either pass, or fail. A passing outcome means that the implementation under test has the property checked by the test case and a failing test case means that the implementation under test does not have this property. In practice, test cases can break: a test case might no longer compile, it might not terminate, or it might not satisfy the precondition (or invariant) of the routine under test. These are situations in which a test case itself is broken and must be either removed or fixed. Similarly to existing work [106, 107] we call such test outcomes unresolved. Hence, a test case is either passing, failing or unresolved:

\textbf{passing} If stimulated with the test inputs, the implementation has the property the oracle asserts.
2.4. TEST OUTCOMES

**failing** The implementation does not have the property the oracle asserts.

**unresolved** The test execution could neither prove or disprove that the implementation has the property the oracle asserts.

Exactly one of the three predicates is true for any given test outcome:

\[
\text{oneof}(\text{pass}, \text{fail}, \text{unresolved})
\]  

(2.1)

where `oneof` is defined as follows:

\[
\text{oneof}(a, b, c) \equiv (a \oplus b \oplus c) \land \neg(a \land b \land c)
\]

(2.2)

There are several reasons why a test case can be unresolved:

**bad code** The test case contains code that is no longer consistent with the implementation under test. For example it calls a routine whose name changed.

**bad state** The test case creates input state that no longer satisfies a precondition or invariant up invocation.

**unknown** An unknown error occurred. The test might not have terminated, or the execution of the test case rendered the testing process into a state where it no longer could output the test result. Since it is in general not decidable if a test case is stuck or just takes long to execute, test cases that execute for too long are classified as having an unknown outcome.

Any test outcome is unresolved either because of bad code, bad state, or some other reason:

\[
\text{unresolved} \implies \text{oneof(bad\_code, bad\_state, unknown)}
\]

(2.3)

Note that an unresolved outcome only implies either bad code, bad state or an unknown error. In some situations, even a broken test cases may lead to an outcome that is either pass or fail. This is the case, when for example, there is a problem during tear-down. In addition to the actual invocation of the routine under test, many test cases have a setup and tear-down routine. The setup routine prepares the environment and the tear-down routine performs cleanup operations. A test execution can go through setup and actual test execution normally but then raise an exception during tear-down. The oracle judges the test outcome before the tear-down. While it does decide whether the test outcome is pass or fail, the test case still needs looking into. Something is wrong with its tear-down and this
might affect subsequent test outcomes. Note that not all exceptions that are triggered during the execution of a test case imply that the test case is broken. Some exception only signal that the implementation under test is broken. Section 2.6 explains this distinction in depth.

We call a test outcome that implies that the developer needs to look at the test case a test case that requires maintenance. An unresolved outcome always indicates that the executed test requires maintenance:

\[
\text{requires\_maintenance} = \text{oneof(bad\_code, bad\_state, unknown)} \quad (2.4)
\]

Note that \(\text{unresolved} \implies \text{requires\_maintenance}\). The above is sufficient to describe testing outcomes relevant for this thesis.

### 2.5 Specification-based testing

Specification-based testing is a form of testing that takes advantage of the specification of a program. When it considers only the specification it is back box testing, i.e. it knows about the interface, but not about the implementation. Specification-based testing can be combined with other techniques that must know about the implementation. Such combinations are obviously no longer a form of black box testing.

The idea of using specification for testing is not new. Already in 1989 Richardson et al. give an overview and classification of it [86]. In short, specification-based testing can mean one or more of the following:

**Specification-based test selection:** The tester prepares test without knowing the implementation. He prepares tests looking at the specification only. Hybrid approaches select test cases from both the implementation and the specification.

**Specification as oracle:** The simplest form of an oracle is one that compares the actual test output with a precalculated one. Such an oracle has limitations: 1) it only works with deterministic implementation. The output of non-deterministic implementations cannot be compared to a single value. 2) It is likely to introduce an implementation bias. The precalculated output must be given in the implementation representation, which might be subject to change. These limitations can be addressed by making an oracle that uses a predicate over the output of the implementation under test to determine the test outcome. Using the specification as an oracle is a form of this where the predicate is taken from the specification of the implementation.
**Specification under test:** Here the specification itself is under test. Technically this is not verification, but validation. One case where this happens is when an abstract model of the implementation is simulated. Specification testing is also always involved when the specification is used as an oracle: if a discrepancy between the specification and the implementation is found it can also be the fault of the specification. Recent research found that automated testing where specification is used as an oracle does find a significant number of bugs in the specification [29].

### 2.6 Contract-based testing

Within this thesis we use the term contract-based testing to refer to a specialization of specification-based testing: it is a form of specification-based testing where the specification is given on the same level of abstraction as the implementation. For example, Korat [19] a testing tool for Java, depends on invariants provided as Java boolean expressions (or as expressions of a specification language, which is translatable to Java). Cheon et al. propose an approach for Java that depends on specifications written in JML. Eiffel’s Design By Contract can also be used for this purpose. Note that contract-based testing is a form of black-box testing. While contracts are interleaved with the implementation they are not a part of it – they are part of the interface. The following explains the concept of specification-based testing in more depth.

Test oracles decide whether a test case has passed or failed. Devising oracles can be one of the most delicate and time-consuming aspects of testing. Assertion monitoring is what enables the use of contracts as oracle for software testing. To understand this more precisely, it is necessary to consider the relationship of contract violations to bugs:

- Since establishing a precondition is the responsibility of a routine’s client (caller), its violation signals a possible bug in the client.

- Conversely, a postcondition or invariant violation signals a possible bug in the supplier.

In both cases the violation establishes a “possible” bug because the error can be due to an inappropriate contract rather than to an inappropriate implementation.

Employing contracts, testing can be turned into a game. The game is aimed at finding as many bugs as possible and we must distinguish between favorable and unfavorable situations:

1. A postcondition or invariant violation is a win for the tester: it has uncovered a possible bug in the routine being called.
2. A precondition violation, for a routine called directly by the tester as part of its strategy, is a loss: the object and argument generation strategy has failed to produce a legitimate call. The call will be aborted; the tester has wasted time.

3. Violated preconditions can also be a win for the tester: let \( r \) and \( q \) be routines. If the tester legitimately calls \( r \) (i.e. the tester satisfies \( r \)'s precondition) and then \( r \) calls \( q \) violating \( q \)'s precondition, the tester found a problem in \( r \).

Figure 2.1: Principle of contract-based testing. A single routine invocation is a complete test. The precondition serves to filter invalid input and the postcondition as an oracle. (Figure also shown in introduction)

The strategy of a tester must be, as a consequence, to minimize occurrences of direct precondition violations (Case 2), and to maximize the likelihood of Cases 1 and 3. All violations matching these cases will be logged, together with details about the context (i.e., the exact contract clause being violated and the full stack trace). This information is very useful for interpreting the results of a testing session: determining whether the violation signals a bug, and if it does (the most usual case), correcting the bug. Figure 2.1 shows the principle of contract-based testing in the form of a flow-chart diagram.

We call input that satisfies the precondition of the routine under test valid and input that violates the precondition invalid. We call the outcome of a test using invalid input unresolved.
Aichernig [6] has looked into the theoretical relation between contracts (in the sense of formal specification) and tests and came to interesting conclusions: test cases can be seen as piecewise specification. In fact Aichernig could show that test cases can be seen as an abstraction of specification. He introduces a refinement calculus which can be used to derive test cases from specification.

Using the idea of contract-based testing, much research has gone into algorithms and tools that generate test cases from specification automatically. The algorithms and tools are categorized and briefly described in Section 3.3.2.

2.6.1 Testing with incomplete specification

Contract-based testing can be applied in many settings, though it is easiest where specification annotations are well integrated and source is by default equipped with contracts. Extensions add support for contracts to some languages that do not directly support it. Even if for a certain language contracts are not supported (either directly or via an extension), the `assert` statement can be used as a replacement. Most modern languages feature this less expressive form of contracts and where lacking it is often supplied as a library routine. These `assert` statements contain a boolean expression that the programmer asserts to be always true when the program execution reaches the statement. Such assertions are not as expressive as Design By Contract assertions: they do not tell who is responsible in the case of a violation. Code with no specification annotation at all, must either be annotated or the annotations must be derived from some external (and possibly more abstract) specification or model.

The more complete the specification, the better contract-based testing works. The weaker the postcondition, the fewer failures it detects. For contract-based testing, too weak preconditions are a bigger problem than too weak postconditions. The weaker the precondition, the fewer failures and invalid inputs it detects. In other words, a weaker precondition increases the number of both false negatives (i.e. the number of failures that should have been detected but were not) and false positives (i.e. the number of inputs that should have been recognized as invalid, but have not been). A too weak postcondition, on the other hand, only has an effect on failure detection. Even in the extreme case, where no postconditions are given robustness related failures (e.g. routine call on void target, division by zero, etc.) or failures due to precondition violations can still be detected. A weaker postcondition increases the number of false negatives failures (i.e. failures that should have been detected, but were not).

Luckily, in practice developers tend to be more diligent with preconditions than postconditions [23]. One possible explanation is: a missing precondition may lead to failures if the routine is called with inputs it cannot handle. A missing postcondition does not lead to failure. This tradeoff — good preconditions,
weaker postconditions — appears acceptable for contract-based testing.

2.7 Resilient execution

Ill-behaved test cases can crash the tester. Generated test cases, in particular when stemming from a random-based strategy, can cause severe side effects. A defect in the system under test can for example harm the runtime system of the carrying process in a way so that it can no longer continue execution. The testing framework must be resilient towards such effects in order to not loose all testing results gathered already. It also must be able to abort individual test cases that are stuck or ill-behaved and continue with the execution of the remaining tests.

It is not uncommon that tests change the filesystem, damage the runtime of the process they are executing from, or call operating system routines that affect the running process. We have observed this phenomenon when applying random- and genetic-based testing to contracted programs. Attempts of random-based testing on the EiffelBase library without any execution protection failed badly. The test application crashed already while generating input data. Traditional unit testing frameworks have a single process architecture: the tester and testee live in the same program unprotected from each other. They depend on the test-developer to provide test cases that are well behaved: even if they fail they must not harm the testing framework. An automated test case generator cannot provide this assurance for the tests it creates.

In previous work [65] we created a simple but effective sandboxing technique that separates the tester from the testee: each process lives in a separate operating system process. The two processes communicate via the standard input and standard output channels. The two processes form a pair of master (the tester) and slave (the testee). The master knows what routines to execute with what arguments, but does not have the means to actually execute them. The testee on the other hand doesn’t have the knowledge on what to test but it can (given all the details from the tester) execute routines from the system under test. The functionality of the testee is similar to that of an interpreter, it receives statements from the standard input and executes them. The only difference to a full interpreter is that the testee does not interpret the system under test, instead it invokes it natively using reflection.

Consider the example depicted in Listing 2.3. It illustrates the kind of input that the interpreter receives from the tester. The interpreter parses the input line by line. The first line contains a creation instruction. After parsing, the interpreter locates the meta object for class\texttt{BANK\_ACCOUNT}. Using this meta object the interpreter reflectively invokes the creation instruction \texttt{make} (of class\texttt{BANK\_ACCOUNT}). The interpreter then associates the resulting object with the name
2.7. RESILIENT EXECUTION

Figure 2.2: Two process testing model to increase resilience. The master sends execution requests to the slave, who triggers the execution and returns whether exceptions occurred or not. When the slave crashes, the master can recover by starting a new slave.

ba in an internal table. Line 2 contains a normal routine invocation. First the interpreter retrieves the object from its internal table that is associated with the name ba, then it looks up the dynamic type of the object and gets its meta-object, which is used again to reflectively execute routine deposit. The interpreter catches all exceptions thrown by the system under test and prints them in a form digestible for the tester to the standard output.

```plaintext
1 create (BANK_ACCOUNT) ba.make
2 ba.deposit (40)
```

Listing 2.3: Example of input as sent from tester to testee.

Figure 2.2 (taken from previous work [65]) illustrates (via an UML sequence diagram) how tester and testee interact. Each lane represents a different operating system process.

The testing tools described in and implemented as part of this thesis feature such a two-process mechanism to ensure resilient execution of test case. The mechanism slows down test execution, because of the communication overhead, but was able to protect the tester from all ill behaved tests generated as part of the
experiments conducted.

If security or safety are a strong concern, even better separation can be achieved by putting each component on a different computer or by jailing the testee in a virtual machine. The advantage of only relying on operating system processes is that the testee is kept simple. It only reads and writes from the standard output. No additional libraries such as networking libraries are needed. The more external libraries the testee depends on, the more surface the implementation under test has to harm things.

2.8 Slicing

Slicing [101] is a technique that, given a program (or a program run) and a slicing criterion, separates the program into a relevant and irrelevant part. For example, given a program statement as slicing criterion, slicing can be used to determine which other statements from the same program can influence the execution of this statement. So called static slices are valid for all possible executions of a program and dynamic slices only for one particular run.

The following first explains the underlying notions of control flow graph, data, and control dependence. It then explains the two basic types of slices: forward slices and backward slices. The section ends with a brief discussion on dynamic slicing.

Control flow graph: a control flow graph (CF) represents all traversable paths of a program. Its nodes represent a statement block, its (directed) edges represent the flow of execution.

Data dependence: a statement $S_2$ is data dependent on another statement $S_1$, iff $S_1$ writes to some variable that $S_2$ reads and there is at least one path in the control flow graph that connects $S_1$ and $S_2$ on which this variable is not written to by statements other than $S_1$ and $S_2$. In the following example the statement in Line 3 is data dependent on the statement in Line 1.

\begin{verbatim}
1  x := 32
2  g := h + j
3  y := x + 3
\end{verbatim}

Control dependence: a statement $S_2$ is control dependent on statement $S_1$ iff the outcome of $S_1$ determines whether $S_2$ gets executed. In the following example the statement in Line 2 is control dependent on the statement in Line 1.
2.8. SLICING

1 if $x > 10$ then
2   $y := 32$
3 end

**Forward slice:** a forward slice of a statement $S$ from a program $P$ is the set of statements that (directly or indirectly) depend on $S$.

$$\text{forward_slice} (S, P) \triangleq \{ S' \in P | S \rightarrow^+ S' \}$$

where $A \rightarrow B$ is true iff $B$ is data or control dependant on $A$ and $\rightarrow^+$ is the transitive closure of $\rightarrow$.

**Backward slice:** a backward slice of a statement $S$ from program $P$ is the set of statements that $S$ (directly or indirectly) depends on.

$$\text{backward_slice} (S, P) \triangleq \{ S' \in P | S' \rightarrow^+ S \}$$

**Dynamic slice:** The slices explained above, called *static slices*, make assertions about all possible runs of a program. Static slices tend to be large, because static slicing is pessimistic. If it is not clear whether a statement is dependent on another statement, static slicing assumes it is dependent. *Dynamic slices* make assertions valid for a given trace (i.e., the ordered list of statements executed by the program) only. Dynamic slices are less general, but smaller. Dynamic slices must know which paths are taken by a program run. To calculate a dynamic slice, programs are instrumented to write out the paths executed. Dynamic slices are easier to compute because traces do not contain any control flow statements.
CHAPTER 3

RELATED WORK

This chapter lists work that is related to this thesis. The chapter’s structure is similar to that of the entire thesis. First, it describes related work on testing in general and automating test execution in particular, then on leveraging existing tests (which is divided into test extraction and test prioritization), creating new tests (which is divided into test case generation and test minimization), and finally on reflection generation.

3.1 Automating test execution

The xUnit family of testing frameworks\(^1\) has had a large impact on practitioners. Tests written for such frameworks are simple and execute automatically. They aid both, debugging and regression testing. Each unit test consists of up to three parts: a setup routine, a test routine, and a tear-down routine. The test routine contains the oracle in the form of `assert` statements. The xUnit framework is small and simple, but had a big impact on how people develop software. Martin Fowler describes it with these words:

> “never in the field of software development have so many owed so much to so few lines of code”

The practice of creating test cases that execute automatically predates the tools from the xUnit family. Most xUnit implementations are small and contain simple code. Nevertheless, something changed with their advent: xUnit tools have a low entrance barrier. It is not the case that developers do not test their applications, they do, but often they do not go the extra mile to automate their tests. Since

\(^1\)http://www.xprogramming.com/testfram.htm
xUnit is simple, it is easy to learn and master. Using xUnit, less effort is required to automate tests which makes test automation more attractive for developers.

While xUnit make it easy to write test cases that execute automatically, other challenges remain. It is difficult to test tightly coupled modules in isolation. For such modules, test developers have to create test stubs and drivers [16]. Several approaches that create these automatically from system level tests have been proposed [88, 43, 83].

To create an automatically executing system level tests it is often necessary to emulate environment such as users, networks, or databases. As a solution, capture and replay tools [39, 41, 80, 105], which observe the execution of a program and record the interactions of the program with its environment, have been proposed.

3.2 Leveraging existing tests

3.2.1 Reproducing failures

Observing the developer or user of a program and reproducing a failure if one happens is a relatively new field of research. Much research has gone into the direction of using capture and replay, which tries to replay program runs in general. Most capture and replay techniques are based on checkpointing the state of the system at certain intervals and recording a log of non-deterministic events in between the intervals [39, 41, 80, 105]. They aim to replay arbitrary executions. In contrast to these our test case extraction mechanism focuses on reproduction of failures.

Clause and Orso recently presented a system that in addition to the checkpointing and log-recording tries to minimize the log [34]. Test case extraction implicitly minimized the number of instructions to replay: each extracted test only executes one method. The current extractors do not yet try to minimize the amount of state that needs to be stored in the test case. We believe that existing minimization approaches like Delta Debugging [107] will yield promising results. First the extractor creates a test case with all state available in order to maximize the number of failures reproduced. Then a minimizer (based on e.g. Delta Debugging) tries to find the smallest subset of the state that still reproduces the failure.

A novel capture and replay approach that logs the interaction of software components at user-defined borders has been proposed independently by Orso et al. [83, 82] (Selective Capture and Replay), Ernst [88] (Test Factoring), and Elbaum [43] (Test Carving). The technique works as follows: a program is divided into two parts. During capture, the interactions (method calls etc.) that cross the boundary are logged. During replay one part of the program is replaced by mock objects which are hard-coded to behave according to the previous recording. The
3.2. LEVERAGING EXISTING TESTS

The advantage of this method lies in the freely definable border between the two parts. Most programs have parts that interact little and parts that interact frequently. By wisely choosing the border the amount to record can be reduced significantly. We believe that this work is orthogonal to ours. If sources of non-determinism are found to be a problem for a given application, they can be mocked out using Selective Capture and Replay, the rest of the state can be extracted as proposed by our method.

ReCrash [10], which is based on our work on test case extraction (as presented in Chapter 4), observes production runs of programs and extracts test cases in the case of a failure. The resulting unit test cases try to re-create the state of the program close to but before the failure. From this state the routines last executing are re-invoked. The authors of ReCrash experiment with a number of variations of the algorithm. Each variation differs in how much state it captures at method entrance and how much at the time of the failure. The more state captured at method entrance time, the more reliable, but also the more overhead the variant induced. In contrast to our approach, their method also works for systems that are not attached to a debugger. Furthermore, they introduce a mechanism called Second Chance, an extraction mechanism that requires a failure to occur at least two times. In such cases the mechanism is both efficient and effective.

3.2.2 Selecting and prioritizing tests

Test selection identifies tests that are relevant with regard to something (e.g., a change to the implementation under test). Test prioritization orders the tests of a test suite according to some criterion (the criterion might be “relevance to a change”). Note that the two are related – every test selection algorithm can be used as a binary test prioritization algorithm: tests that are relevant receive priority high, all other tests receive priority low.

Wallcott et al. [99] use a genetic algorithm to re-order the execution of the tests of a test suite. The algorithm needs to know how long it takes to execute each test as well as the fault finding capability of each test. Since the true fault finding capability is often not available, in practice it is replaced with a coverage measure. Their algorithm tries to minimize the time it takes until defects are found. Since the algorithm is time consuming itself, it only pays off for large and static test suites. In such settings it makes sense to re-order the tests of a test suite only every once in a while. Our approach, is less precise but can be run on the fly before each test-run.

Chen et al. [24] describe a system to select tests given a certain test scope or a recent change to the program. Their approach is based on a combination of dynamic and static analysis. They mention that a static analysis would be sufficient, but the performance overhead not acceptable. Our approach, on the other hand
does not need dynamic analysis, because (due to our lexical approximation of the inheritance and association relation) the static analysis used is cheap to compute.

Echelon [91] is a tool that prioritizes tests using programs in binary form. It is based on a binary comparison of the old and new version of the program in question and on coverage analysis.

Several researchers recently proposed slicing as replacement for coverage analysis to prioritize and select tests for regression testing [58, 59, 47].

Elbaum et al [44] present a comparative study of test case prioritization techniques. Their study considers both fine-grained techniques (e.g. ones working on the level of statements) and coarse-grained techniques (e.g. ones working on the level of functions). They find that prioritized test suites detect faults earlier. They also find that the fine-grained techniques yield better result, but only by a small margin. Do et al. [40] provide a similar study but focus on object-oriented programs (written in Java and tested via jUnit). They confirm that test prioritization improves the rate at which faults are found. They find that even a random ordering yields a better rate than the default ordering (as provided by the developer). The authors argue that this could be due to the fact that tests at the end of a test suite are more likely to reveal faults than others. (Developers add new tests at the end; new tests are likely to exercise new code; new code is more error prone). While they find that more analytical techniques outperform random ordering, they do not find that fine-grained prioritization techniques yield better results that coarse-grained techniques.

Our technique is based on the basic unit of object-oriented programs. The technique we present is coarse-grained, and cheap to compute.

RoseStudio, an in-house IDE developed at AXA Rosenberg, also implements test selection through analysis of the client relation.

3.3 Creating new tests

3.3.1 Test case generation

Cheon and Leavens [25] use the run-time assertion checker of a formal language (JML) to generate unit tests (JUnit). JML contracts are more expressive than Eiffel contracts. For example, they can contain quantifiers. Run-time assertion checking is limited to the executable subset of JML, however, where contracts are approximately as expressive as Eiffel contracts. Similarly to our approach for test case synthesis they use preconditions to filter invalid input and postconditions to flag failures. Only a subset of JML is executable (quantifiers, for example are not executable), but the assertion monitor needs to execute assertions to evaluate them. Hence their approach can only be applied when a program is annotated with the
executable subset of JML. They generate the code to invoke the methods under test, but contrary to our approach they depend on the user to provide the input data for testing.

Several researchers used random input generation as a basis for specification- or contract-based testing. Csallner et al. [35] show how random testing can be applied to Java programs. Their oracle uses a heuristic based on run-time assertions. For example, if a method invoked by the tester throws an IllegalArgumentException the oracle does not consider this to be a bug. If the tester invokes a method, which invokes another method, which throws an IllegalArgumentException, the oracle reports a bug. Andrews et al. [7] show how test fragment collections can help the developer apply random testing on Java programs. JTest [1] is a commercial random testing-based tool. Jartege [81] tests Java source code annotated with JML specification. It creates inputs not directly, but via constructor and method calls. Directed Adaptive Random Testing [48] is a variation of random testing that tries to spread test input evenly over the input domain. Ciupa et al. [31] extended this idea to work with object-oriented and contracted code.

Previous experiments on contract-based testing with a random test synthesis approach [28] indicate that failure-reproducing test cases for industrial code can be found and stored automatically. Generated test cases however, tend to be large which makes it hard for developers to understand them and subsequently to fix the corresponding fault. The minimization technique presented in this thesis addresses this problem.

Research has also gone into contract-based testing where the synthesis strategy is aided by symbolic execution, constraint solving and theorem proving [36, 37, 104, 20, 18, 98, 73].

With Korat [19], Boyapati et al. propose a contract-based testing method that efficiently iterates through valid test inputs. During the execution of invariants it monitors what fields get accessed. Assuming determinism, fields not accessed cannot affect the outcome. If for example an input is found to violate the invariant, inputs that differ only in fields not accessed also violate it.

Other approaches rely on the dynamic inferring of invariants [45] (e.g. Eclat [84]) or genetic algorithms [97].

Classen and Hughes [33] experimented with an automated tester for Haskell programs, called QuickCheck. QuickCheck does not rely on contracts (i.e. specification that is embedded in the source code). Instead, it requires the user to write laws about the implementation under test in the form of Haskell functions. These functions are similar to normal unit test cases except that they have arguments. QuickCheck takes the the provided functions and invokes them with random data. QuickCheck provides data generators for built-in types. The user is required to provide tailored generators for user defined types, though. In contrast, our approach relies on contracts, which specify properties that have to hold for all exe-
cutions of the implementation and is able to create input for both basic and user provided types fully automatic. The test functions for QuickCheck describe properties that have to be true only for runs of the test function. They are less general, but because of this, make it easier to describe complex and specific properties.

Similar in concept, Tillmann et al. describe parametrized unit tests [96]. A parametrized unit test is similar to a normal xUnit style test case, except that it accepts input arguments. Where QuickCheck uses generic data generators (which do not know about a particular implementation under test), Tillmann et al. show how a combination of symbolic execution and constraint solving can be used to automatically generate inputs tailored for a particular implementation under test: first a symbolic executor goes through the implementation to calculate path conditions, then a constraint solver is used to derive concrete inputs to cover all code paths. They state that full automation cannot be achieved however due to limitations in both the symbolic analyzer (because there can be infinitely many paths) and the constraint solver (solving of constraint can be computationally infeasible).

3.3.2 Test minimization

With Delta Debugging, Zeller et al. [107] present a general and automated solution that simplifies failure-inducing inputs to a 1-minimal input (a test input is 1-minimal iff it is not possible to remove any further element without having an input that does not fail). In the case of inputs being programs, and more generally in the case the input is strongly bound by a known semantic of the computation, other domain-specific techniques can be much more efficient. In the case of static slicing, the slices may not reproduce the error because of potentially hidden side-effects. When this is the case, it is possible to rely on Delta Debugging to compute a minimized test case that reproduces the failure. Our experiments indicate that slicing minimizes test cases nearly as effectively as Delta Debugging. When slicing is less effective it still reduces the number of executions needed by Delta Debugging and hence the overall minimization time.

Program slicing was first introduced in the testing area for improving regression testing [5, 55, 58, 59, 47]. In contrast to our approach, which minimizes individual test cases, their goal is to minimize test suites by removing redundant tests.

Combining Delta Debugging and program slicing for testing programs was already proposed by Gupta et al. [54] with a different purpose from what is described here. In that approach, the test cases are minimized with Delta Debugging and then the input is taken as the input for dynamic slices of the failing program. In the present work, slicing and Delta Debugging are both applied to a program only. In this case, Delta Debugging and slicing are then competing solutions rather than complementary techniques.
3.4. REFLECTION GENERATION

While many approaches concentrate on the generation of failure-producing test cases, most do not consider test case minimization. We are only aware of two approaches that minimize test cases, both using Delta Debugging. The work of Lei and Andrews [63] minimizes randomized unit tests, just as we do, whereas the work of Orso et al. [82] minimizes captured program executions. As discussed in Section 6.9, our approach is far more efficient than Delta Debugging alone; we use Delta Debugging as an optional step to further minimize sliced test cases and as a fall-back mechanism for when program slicing fails. By doing so, it is possible to speed up the minimization process by a factor of 11 comparing to a pure Delta Debugging minimization technique—which makes it applicable to large industrial code bases, such as the EiffelBase library.

3.4 Reflection generation

Recent research in the area of reflection has been strongly centered around separation of concerns and aspect-oriented programming (AOP). In the following we overview work on reflection closely related to ours, which for the most part predates AOP.

Inherent support for reflection

Reflection has traditionally been often either at the core of a language or absent from the language. CLOS [17, 60] and Smalltalk [87] for example integrate reflection as a core concept that guides the entire program execution. In both CLOS and Smalltalk, reflection is embodied by a meta-object protocol (MOP). Roughly, an MOP promotes meta-objects (and meta-classes) which allow the programmer to modify the behavior of the objects at runtime. To assist an MOP programmer, any single construct involved in the MOP is reified (put into an object form) and can be modified directly within the MOP. Depending on the type of modification allowed, there are different types of reflection that are available to programmers. Terminologies diverge in the literature. Generally, though, structural reflection implies a reification of the data handled by the running program; in that context, introspection refers to the ability of reading that data, and intercession allows for alterations thereof. Dynamic invocation facilities circumventing (static) type checking are commonly conceived as part of introspection mechanisms. Behavioral reflection reifies the semantics of the program and the execution, and is mostly interpreted as the ability to intercept/redirect calls. Both CLOS and Smalltalk support all these kinds of reflection by design. Smalltalk, for instance, provides for any given object a default routine doesNotUnderstand which is executed whenever the object receives a call for which no routine with matching signature exists – a likely event
given the dynamically typed nature of the Smalltalk language. Being interpreted, Smalltalk allows for strong support for behavioral reflection in the form of first class parse trees available to a program at runtime.

In a different setting (e.g. an existing language), it is not possible to change the core of the language and this is where our approach has added value.

Add-Ons

For languages that include only marginal support for reflection or for which no reflection support whatsoever is available, benefitting from (more advanced) reflective structures means either changing the runtime or instrumenting the code. C++ [93] and Objective-C [85] are examples of languages devoid of any reflective features. Efforts for providing reflection for C++ through an MOP include OpenC++ [26], Iguana [52] and MPC++ [57]. MPC++ and Open C++ claim to be meta-level, compile-time frameworks respectively, but bear more resemblance to complex preprocessing tools than to meta-object protocols, which by definition should allow run-time changes. Iguana generates the reflective infrastructure for each of given types of meta-data. While this infrastructure is the most complete one — combining introspection and behavioral reflection — it falls short in support for generics (i.e., templates in C++).

3.4.1 Extensions

Imperative languages like Eiffel [75], Java [8] or C# [56] only structural reflection. Even within these confines, there is usually a lack of fine-grained enough features: Java does not provide a satisfactory mechanism to introspect generics, C# enables reflection at the assembly level only, and Eiffel does not even support dynamic invocations.

Reflection frameworks for Java (e.g. Kava [103], Reflex [95] and Javassist [27]) complement the existing Java reflection APIs by adding further reflection support at load-time through byte-code instrumentation. This load-time approach has been popularized by Java thanks to its inherent dynamic code loading and linking facilities. A limited mechanism for behavioral reflection with a load-time flavor has been officially added to Java at 1.3 through dynamic proxies [94]. A dynamic proxy is a program-controlled typed proxy object implementing a set of types chosen by the program. It is created at runtime by generating a corresponding proxy class directly as byte-code and loading it. The approach is limited to interface types, however [46].

Neither the inherent introspection capabilities nor any of the above-mentioned frameworks provide support for generics due to their late addition to the Java language. Our infrastructure handles generics by default. It also differs from
the Java-related frameworks in that it generates the reflective code and allows programmers to modify it and apply further transformations to regular Eiffel code. On the downside, our approach does not yet cover full behavioral reflection.
CHAPTER 4

EXTRACTING TESTS

This chapter presents test case extraction. Section 4.1 gives an overview of the approach. Section 4.2 motivates it. Section 4.3 demonstrates the mechanism via an example. Section 4.4 explains the concept more formally. Section 4.5 shows how it can be implemented and Section 4.6 evaluates it.

The content of this section is based on our publication to the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE’07) [67].

4.1 Overview

For some programs, in particular interactive ones, it can be hard to write test cases. Developers may find it easier to test such programs manually, simply by invoking the program they develop. These tests are transient, cannot be used for regression testing, and are easily forgotten. The following presents a mechanism that observes these invocations and in the case of a failure, produces a persistent test case. In order to be usable during development such a mechanism must be unobtrusive. The mechanism described is completely automatic and hence does not disturb the developer.

Usually, failures are made reproducible via a combination of taking regular snapshots of the program state and recording of incoming events. This technique, called capture and replay comes with a performance overhead, may be tied to a particular platform, or require special hardware.

This section proposes a state-based test extraction mechanism, which induces no run-time overhead. Test extraction aims to reproduce a failure by invoking all routines on the stack from the state that originally led to the failure. Contracts help to exclude routines that are not of interest: no test case needs to be extracted
for a routine whose precondition was violated.

In principle test extraction induces run-time overhead, but there exists an effective approximation which does not. For precise (but slow) test extraction, throughout the execution of a program, whenever a routine is invoked, the extractor needs to copy the state visible from that routine. When a failure happens the extractor creates a test case for every routine on the stack at this moment. Each test case invokes one method from the stack using its previously saved invocation state. The extractor can dispose saved state when a routine terminates. The capturing of state for every for every routine invocation causes overhead. Artzi et al. [10] measured an overhead between 12,000% and >638,000%.

We propose an optimization which invokes routines from the state at the time of the failure instead. We refer to this variant as failure-state extraction (as opposed to invocation-state extraction). Failure-state extraction induces no run-time overhead, but is not able to reproduce some failures:

- it does not reproduce failures that require an external event between routine invocation and failure and

- the state between routine invocation and failure may have changed. A routine might reproduce a failure when invoked from its original invocation state, but not from the state of the failure.

We rely on the fact that most failures are detected close to their faults. Each extracted test cases invokes only one routine directly. We assume that:

- most routine executions are short and do not require external events;

- most routine executions alter only a small part of the state. Due to this the difference between the invocation state and the failure state is small and it is likely that the failure state can be used instead of the routine-entrance state to reproduce the failure.

The purpose of test extraction is to reproduce failures. It is secondary how many test cases reproduce a failure, as long as there is at least one test case that reproduces a failure. Every failure causes the extraction of a set of test cases, which increases the chances of success.

These assumptions need validation however. We conducted a user study to evaluate our test extraction method. Test cases extracted using the cheap failure-state reproduced 90% of all failures. Test cases extracted using the more expensive invocation state only reproduced 4% more.
The results suggest that failure-state extraction works well enough. The overhead of invocation-state extraction is not warranted. A combination of both methods, which applies failure-state extraction by default and invocation-state extraction only when necessary gives the best of both worlds. Artzi et al. [10] developed several variants of failure-state extraction, one of which, called Second Chance, extracts failures only on their second occurrence, but then with less overhead. When a failure occurs for the first time, no test cases are extracted. Instead the methods on the stack at this moment are marked as observed. For subsequent program execution, state is only saved when an observed method is entered. When a failure occurs a second time, the extractor uses the previously saved states to create test cases. The gain in performance varies depending on the program and failure under consideration, but gains of 1000% were not atypical. We have integrated their optimization into our approach in the following way: If a failure occurs, the failure-state extractor generates test cases (without imposing run-time overhead). The IDE automatically executes the new test cases. If the failure is reproduced by at least one test case nothing else happens. If the failure is not reproducible, the methods that were on the stack at the time of the failure are marked as observed. From now on, the invocation-state extractor is activated, but restricted to methods marked as observed. Whenever a method that is marked as observed is entered, its relevant state is captured. For other methods nothing happens. If the failure that was not reproducible with the failure-state extractor occurs a second time, the extractor uses the previously saved invocation states to generate test cases. The resulting test cases are identical to ones generated by a pure invocation-state extractor, but because of Second Chance the extractor requires much less overhead. As soon as a failure is reproduced via a test case, the methods previously marked as observed are unmarked. The developer may also unmark methods manually.

4.2 Motivation

Writing unit tests during the development of software systems brings obvious benefits. Although developers are aware of these benefits, they still write only very minimal unit tests. In order to find the reasons for this, we conducted a small scale study on Computer Science students from the ETH Zurich, asking them various questions about their unit-test-writing habits. Their degree of experience in writing software varied widely, from 6 months to 9 years. So did the number of software projects that they had worked on until then, from 1 to 5. 45% of the students said that they never wrote a unit test case before the implementation and 36% do it only very seldom. After implementation, only 18% of the students always write unit tests. 54% do it very often. We also asked the students to rank (on a scale from 1 to 5, where 1 represents total disagreement and 5 total agreement)
the causes that prevent them from writing unit tests. The causes and associated average ranks were:

- “Writing unit tests takes too much time”: 4.4
- “It takes too much effort to maintain the unit tests”: 3.6
- “Writing unit tests is too much effort for the provided benefits”: 3.2
- “It takes too much time to run the unit tests”: 2.1
- “Unit tests are not useful”: 1.8

The results of our study show that the time and effort involved in writing and maintaining unit tests are the most often occurring causes for the developers’ dislike of unit testing, as also indicated by other studies.

Executing ad-hoc test cases require less effort. They have a very serious drawback, however: they are implicit; usually they exist only for one or very few runs and cannot be kept for later automatic re-execution because:

- If the developer had to provide inputs, the test case cannot be rerun without him providing the same inputs again. This requires manual intervention and the developer needs to remember the exact inputs.

- If the developer changed parts of the program to force a certain path to be taken, then this change is unlikely to persist as it limits the generality of the application. Usually such a change is undone or altered yet again to create the next implicit test case.

It is of obvious benefit to capture these manual test cases so that they can be executed automatically. Capture and replay tools can be used to reproduce manual test cases, but induce run-time overhead. At regular checkpoints they store a snapshot of the state of the system. In between these snapshots they log all incoming events. In the event of a failure the user has to locate the closest checkpoint and can then debug the issue by replaying the log of external events.

### 4.3 A use case

To give an intuition of our proposed method, this section presents how it can be applied during the development of a program. In the following example, for the sake of simplicity we consider only one extracted test case per failure (the test case invoking the routine that immediately triggered the failure; the others, which invoke the routines further down the call stack, are ignored).
class BANK_ACCOUNT
inherit ANY
    redefine default_create
end

feature default_create
do
    balance := 300
end

balance: INTEGER

deposit (an_amount: INTEGER)
do
    balance_increased: balance > old
    balance deposited: balance = old balance + an_amount
end

withdraw (an_amount: INTEGER)
do
    balance := balance - an_amount
ensure
    balance_decreased: balance < old balance
    withdrawn: balance = old balance + an_amount
end

invariant
    balance_not_negative: balance >= 0
end

Listing 4.1: Main class of running example, representing a bank account.
Consider an application written in Eiffel providing the means to deposit and withdraw money from a bank account. Listing 4.1 shows the main class of this application and Figure 4.1 shows a screenshot of the running application. In addition to class `BANK_ACCOUNT` (shown in Listing 4.1), which implements the main business concept of the software system, the actual application also contains:

- Class `MAIN_WINDOW`, which represents a GUI window showing the current account balance and allowing the user to enter an amount that he can then deposit or withdraw.
- Class `INTERFACE_NAMES`, which contains some global application constants.
- Class `APPLICATION`, which serves as the entry point of the application. It creates a bank account and a main window, passes the account to the main window, displays the main window, and starts the event loop.

Note that this example represents an application under development; it contains both incorrect and unfinished code. However, the developer tests the application by launching it as it is. He starts out with an empty unit test suite. He
4.3. A USE CASE

```java
class TEST_CASE_2
feature
test
do
    local ba : BANK_ACCOUNT
    ba := new_object ("BANK_ACCOUNT")
    set_field (ba, "balance", 300)
    check_invariant (ba)
    ba.withdraw (20)
end
end
```

Listing 4.3: Class TEST_CASE_2, automatically extracted from run of application that withdrew money.

runs the application from within the debugger of his IDE (which supports test case extraction) and enters through the GUI that he wants to deposit 30 CHF. The GUI invokes the routine deposit of class BANK_ACCOUNT, which throws a post-condition error and stops the application. As usual, the debugger indicates the line of the violation, the current stack frame and the values of the variables in scope. Without the extraction mechanism, the developer would have to fix the bug immediately or the failure information would be lost. With extraction the test case extractor becomes active and extracts a test case that tries to reproduce this failure. This test case is automatically added to the previously empty test suite. The IDE detects the new test case, runs it and updates the testing window, which now looks as shown in Figure 4.2(a). The extracted test case is depicted in Listing 4.2. Note that the test case does not provide an oracle, since the postcondition of routine deposit takes over this responsibility.

The reason why the above failure occurs is that there is no implementation yet for routine deposit. This is similar in spirit to test first development, where tests are prepared before the implementation.

Since the failure is now reproducible via a test case, the developer no longer has to fix the fault immediately. He can go on and test another aspect of the application. For example, he can try to withdraw something from the bank account. If he does that, the debugger will once again stop the application and signal a fault in the postcondition. In this case the routine was implemented correctly, but the postcondition contains an error. The test case extractor again automatically creates a test case (depicted in Listing 4.3) for this failure. Then it adds this test case to the test suite, and compiles and runs it in the background. The test case status window is hence updated to show two failing test cases (Figure 4.2(b)).

Suppose the developer now adds a correct implementation for routine deposit and fixes the postcondition of routine withdraw. Since the IDE employs contin-
48 CHAPTER 4. EXTRACTING TESTS

(a) After one failure  
(b) After two failures  
(c) After fixes applied

Figure 4.2: Test-cases automatically extracted from failures (Screenshot of test-case-window as integrated into EifelStudio IDE)

uous testing [89], the test cases are flagged with PASS automatically, as shown in the test case status window (Figure 4.2(c)), indicating that the faults previously detected are fixed. Note that it does not indicate the absence of all bugs.

4.4 Model

The following describes test case extraction abstractly. In order to stay language agnostic and avoid technical details throughout this section uses mathematical functions to describe test cases and test case extraction. The following model is general enough to describe both invocation-state extraction and failure-state extraction.

4.4.1 The called-by tree

A test case extractor produces test cases that reproduce program executions. The following defines the called_by tree, which is an abstraction of program execu-
tions. The purpose of the tree is to capture the program flow of an execution. It does not retain information about primitive operations such as integer addition, for example. It only retains information about routine invocations and object initialization. Since creation procedures (which initialize objects) can be looked at as regular routines the following does not make a distinction between normal routine calls and creation procedures. The called_by tree captures

- which routines where invoked (and from which state and with what arguments),
- in which sequence they were executed, and
- which routines called which other routines.

Every node in the called_by tree represents a given invocation of a routine: the routine invoked and its context. An example graph can be seen in Figure 4.3. This graph is based on a trace of the bank account example introduced in Section 4.3.

Listing 4.4 provides some more details of the bank account application introduced previously. We use this example throughout this section. Class APPLICATION contains the main event loop (which we consider to be the application’s entry point for our discussion) in routine event_loop. This routine is responsible for calling the corresponding subscribers for each observed event. The routine uses the Eiffel agent mechanism (which is similar in intent to the C# delegate mechanism), where each event keeps a list of its subscribers. A subscriber is just another routine (that must have been registered before with the event). The event_loop routine consists of two nested loops: the outer loop is executed once for every event, while the inner loop iterates over the subscribers of each event and calls these subscribers. Routines deposit_amount and withdraw_amount respectively subscribe to the events associated to the two buttons of the application (“deposit” and “withdraw”, as seen in Figure 4.1). These routine in turn read the amount entered via a text entry box and then call the corresponding routine from class BANK_ACCOUNT.

The context of a routine invocation consists of the objects accessible during its execution. These are the object on which it is executed, the arguments passed to it, and whatever objects are reachable from those. Consider routine invocation < withdraw, c11 > from Figure 4.3. The context c11 consists of an object of type BANK_ACCOUNT (i.e. the account to which money is deposited) and an object of type integer (i.e. the amount of money deposited). Let $R$ be the set of routines and $C$ the set of contexts, then the set of routine invocations is $R \times C$. A routine invocation is said to be well formed if its context is such that the routine can be executed from it without any syntax or type errors:

\[
\text{wellformed} : R \times C \rightarrow \mathbb{B}
\]
class APPLICATION
feature
... event_loop
  do
    ...
    from
    until
      should_quit
    loop
      wait_for_event
      from
        ev.subscribers.start
      until
        ev.subscribers.after
      loop
        ev.subscribers.item.call
        ev.subscribers.forth
      end
    end
  end
...
end

class MAIN_WINDOW
feature
... amount: TEXT_FIELD

account: BANK_ACCOUNT

deposit_amount
  do
    account.deposit (amount.to_integer)
  end

withdraw_amount
  do
    account.withdraw (amount.to_integer)
  end
...
end

Listing 4.4: Partial source for example application. It shows the root class (class APPLICATION) and the class representing the main window (class MAIN_WINDOW)
Every routine invocation executes on a target object. For example, the target object of the routine invocation `account.deposit(30)` is `account`. In the case of a creation procedure the object being created serves as the target object. Let $O$ be the set of all possible objects. Then the signature of `target` is:

$$
target : C \rightarrow O
$$

The edges of the called_by-tree indicate which routine invocation was called by which other routine invocation. Let $< r_1, c_1 >$ and $< r_2, c_2 >$ be two invocations. Then $< r_2, c_2 >$ is called by $< r_1, c_1 >$ if and only if $r_1$, while executing in context $c_1$, directly (i.e. not indirectly via some other routine invocation) invokes $r_2$ in $C_2$:

$$
called_by : R \times C \rightarrow R \times C
$$

Any node in the tree can potentially trigger a failure. Failures occur due to a contract violation, routine call on void target, operating system signals or other kind of exceptions. Each programming language will have its own set of causes. For Eiffel the list is given in the Eiffel ECMA standard Section 8.26.1 [42].

![Diagram showing called_by-tree](image-url)
In the presence of contracts, every failure not only has an origin (the routine invocation that immediately triggered the failure), but also a recipient. Intuitively the recipient is the routine responsible for the failure. In most cases the recipient and origin are the same routine invocation. The recipient of a failure that is not due to a contract violation (e.g., a division by zero) is the routine from which the failure originates (e.g., the routine containing the division). The recipient of a postcondition violation is the routine that contains the postcondition. The recipient of a precondition violation, on the other hand, is the routine that invoked the routine which has its precondition violated. The recipient of invariant violations depends on when the invariant is broken: if the invariant is violated on routine entrance, the recipient is the caller. If it is violated on routine exit, the recipient is the called routine.

For Eiffel, a precise definition of recipients is defined in the ECMA Standard Section 8.26.10. The semantics of recipient is extended to not only mean the receiving routine, but also its context. With \( F \) being the set of failures, the signature of recipient becomes:

\[
\text{recipient} : F \rightarrow R \times C
\]

### 4.4.2 Test cases

For the purpose of test extraction, a test case is a routine invocation. The unit under test is the routine called, and the context is the test input. The oracle of such a test case consists of the contracts of the routine and checks of the implementation (e.g. division by zero checks, or assert statements). Let \( < r, c > \) be a test case, then it is executed in the following way:

1. Recreate state from context \( c \).

2. Check invariant of \( c \) (if violated \( \rightarrow \) unresolved test case).

3. Check precondition of \( r \) in \( c \) (if violated \( \rightarrow \) unresolved test case).

4. Invoke \( r \) in recreated state using arguments from \( c \) (if normal termination and postcondition is satisfied \( \rightarrow \) pass, otherwise \( \rightarrow \) fail).
More formally the oracle of a test case can be defined in the following way:

\[
\text{tc}_{\text{valid}} : R \times C \to \mathbb{B} \\
\text{tc}_{\text{valid}}(r, c) \triangleq \text{wellformed}(r, c) \land \text{inv}(r, c) \land \text{pre}(r, c)
\]

\[
\text{tc}_{\text{passing}} : R \times C \to \mathbb{B} \\
\text{tc}_{\text{passing}}(r, c) \triangleq \text{tc}_{\text{valid}}(r, c) \land \text{n-terminates}(r, c) \land \text{post}(r, c)
\]

\[
\text{tc}_{\text{failing}} : R \times C \to \mathbb{B} \\
\text{tc}_{\text{failing}}(r, c) \triangleq \text{tc}_{\text{valid}}(r, c) \land \neg \text{tc}_{\text{passing}}
\]

Intuitively, \text{pre}(r, c), \text{post}(r, c), and \text{inv}(r, c) represent the contracts of the routine invocation \(<r, c>\). Since contracts are only defined for routines and not invocations of routines, the following more precisely defines their meaning.

The predicate \text{pre}(r, c) is defined as the result of the evaluation of the precondition of routine \(r\) in the context \(c\).

The predicate \text{post}(r, c) is defined as the evaluation of the postcondition of a routine. As with \text{pre}(r, c), this is equivalent, to the evaluation of the postcondition of the called routine in the context \(c\). Note that the postcondition will not be evaluated if the routine does not terminate, hence \text{post} is only defined for terminating routines.

The predicate \text{n-terminates}(r, c) is true if and only if the routine terminates normally. In the case of Eiffel (and Spec#) an abnormal termination (such as a null pointer dereference, division by zero, operating system signal, etc.) does not guarantee any postcondition, which is the case described by the formalism above.

In JML there is a pair of pre- and postconditions for normal termination and separate pairs for different kinds of abnormal termination. The above formalism can be easily adapted to this case. Similarly to the way one big pre- and postcondition pair is formed for theorem proving JML annotated programs, a big pre- and postcondition pair can be used for the oracle predicates above.

The predicate \text{inv}(r, c) intuitively tells whether the context \(c\) is broken or not. It might be confusing that \text{inv} has two arguments, a routine and a context. At first, one might be tempted to require the invariant to hold for all objects in the scope, in which case it does not matter which routine gets executed. Things are more complicated, however, because there is a need to temporarily break the invariant in order to allow for object state to change. The exact way in which this is implemented depends on the contract-enabled language. In Eiffel the following rule fulfills this purpose: the object which is the target of the currently executing routine is allowed to have its invariant temporarily violated. A routine can trigger
the execution of another routine. Consequently, more than one object at any given point in time can have a violated invariant.

In Eiffel, the assertion monitor checks the invariant of an object before and after a routine is executed on it only. This approach is too optimistic when routine invocations are considered to be test cases. In our setting, this would imply that we only need to check the invariant of the target object to determine whether the context is broken or not. An invariant breach due to a routine call in the middle of a routine invocation operates not necessarily on the initial heap, but on a potentially modified one. When the invariant is broken we cannot know whether this is due to the context being broken already before the test execution started or whether it was broken due to the execution. This distinction is important. For example, a test case might be extracted at first with a context containing a set of objects that satisfy invariants, but then, as a result of changes in the program, the invariant of a class is strengthened and the extracted context may contain objects that do not satisfy the new invariant.

Neither complete invariant checking, nor the checking employed by the Eiffel assertion monitor is appropriate. Instead, the scope of the invariant check must be broadened to include the information of the executing routine and their called_by information. A first intuition is to check the invariant of all those objects in the scope, except those which are target to any of the currently executing routines (e.g. the targets of the routines in the transitive closure of the called_by relation to the current invocation). In order to express this, the notion of the target set of a routine invocation is handy. It is the set of all objects serving as target to any of the currently executing routines (where called_by* is the reflexive transitive closure of called_by):

\[
\text{target\_set} : R \times C \rightarrow \mathcal{P}(O) \\
\text{target\_set}(r, c) \triangleq \{ \text{target}(c') | \langle r', c' \rangle \text{ called\_by}^* \langle r, c \rangle \}
\]

However, requiring all objects not in the target-set to have a valid invariant is overly protective for the purpose of test case execution. The context might perfectly well contain objects with broken invariants that are not needed for the execution of a routine invocation. In such a case (caused by natural program evolution) one should not be required to throw away the test case. We hence use a notion of necessary objects of a routine invocation: an object is necessary for an routine invocation if and only if the execution accesses the object.

\[
\text{necessary} : R \times C \times O \rightarrow \mathbb{B}
\]
Based on these notions, the final definition of the invariant check is:

\[ \text{inv} : R \times C \rightarrow \mathbb{B} \]

\[ \text{inv}(r, c) \triangleq \forall o \in O | (\text{necessary}(r, c, o) \land \neg (o \in \text{target\_set}(r, c))) \rightarrow \text{inv}_{\text{obj}}(o) \]

where \( \text{inv}_{\text{obj}} \) is the invariant of an object as defined in the class’s contracts.

Given these definitions, extracting a test case from a failure becomes very simple:

\[ \text{testcase}_{\text{failure}} : F \rightarrow R \times C \]

\[ \text{testcase}_{\text{failure}}(f) \triangleq \text{recipient}(f) \]

The definition \( \text{testcase}_{\text{failure}} \) gives a test case for the routine that immediately was responsible for the failure. This assumes that the contracts are sufficient and correct. What happens, as can be the case in practice when they are not? For example, imagine that the invariant of an involved object or the precondition of the recipient of the failure is too permissive. In this case the test case as defined by \( \text{testcase}_{\text{failure}} \) still highlights a problem (the incorrect or insufficient assertion), but the actual defect is located in code executed earlier. In order to increase the chances of extracting a test case that executes the code that has the defect it is necessary to extract more than one test case per failure. The test suite of a failure \( (\text{testsuite}_{\text{failure}}) \) is defined via all the routine invocations directly or indirectly called by the recipient of the failure:

\[ \text{testsuite}_{\text{failure}} : F \rightarrow \mathbb{P}(R \times C) \]

\[ \text{testsuite}_{\text{failure}}(f) \triangleq \{ \text{tc}(r', c') | \text{recipient}(f) \text{ called by }^* \langle r', c' \rangle \} \]

Extracting a failure suite instead of just one test case is also beneficial in the extreme case of a too permissive contracts: when no contracts are given the developer can choose the test case most useful for further debugging.

### 4.4.3 Further applications of test case extraction

This section gives two further applications for test case extraction: debugging and test factoring.
CHAPTER 4. EXTRACTING TESTS

**From testing to debugging** One of the advantages of automatic test case extraction is that a developer observing a failure has the choice of fixing the fault either immediately or later, since the failure is automatically reproducible. Furthermore, test case extraction can also provide a benefit that extends into the process of fixing the fault. Most non-trivial faults need to be fixed in several places, not just in one. The test suite extracted for a failure can provide guidance for this process. For example, an observed failure stemming from a null pointer dereference in a routine \( m \) can be fixed by either changing the implementation of \( m \) in a way so that the null case is treated via a special path, or by strengthening the precondition of \( m \) to exclude the case in which the reference is null to begin with. Note that, if the developer chooses this latter fix, now the calling site of \( m \) violates \( m \)'s precondition. Again the failure test suite helps. The test case extracted for the routine receiving the failure turns unresolved. The test case extracted for the routine calling the immediate routine will still fail and must be fixed. In this way the test outcomes guide the developer.

**Test factoring** So far test case extraction is a means to reproduce failures. It can, however, also be used to factor large and time-consuming system level tests into many fast and focused unit level tests. Test factoring was first introduced by Saff and Ernst [88]. They propose a mechanism that factors large system level tests into small unit tests via a capture and replay mechanism: There the system under test is divided into two parts: the unit under test and the rest of the system. They then record all interactions between these two parts and during replay replace the rest of the system with mock objects. The mock objects behavior is determined via the previous capturing. This test factoring method removes the part of the system that is deemed not relevant by the developer.

We propose an orthogonal way to factor test cases: instead of cutting out a part of the system, test extraction can be used to cut out the non relevant part of the test by providing targeted test cases that only invoke the routines relevant for the failure. This is achieved by generalizing the test suite notion from above from failures to routine invocations:

\[
\text{testsuite}_{\text{invoc}} : R \times C \rightarrow \mathcal{P}(R \times C) \\
\text{testsuite}_{\text{invoc}}(r, c) \triangleq \{ tc(r', c') \mid \langle r, c \rangle \text{ called by } \star \langle r', c' \rangle \}
\]

With the presented test case extraction mechanism it is possible to extract short running unit level tests automatically from existing system level tests in just one step. When the IDE signals that the developer is about to change a certain module (in the sense of class or package), relevant system level tests can be executed.
automatically and for each routine invocation of the targeted module a test case is extracted.

While the developer changes the module, he gets instant feedback about whether he broke anything, without the added overhead of unit test suite maintenance.

**Deep invariant checks for traditional debugging**  The traditional approach to runtime monitoring of invariants (checking the invariant at routine entry and exit) is a compromise between performance and correctness. It captures many invariant violations, but routines accidentally violating the invariant of objects other than the target object can lead to an infected state that is not discovered at the time of the infection.

The deep invariant check proposed earlier can be used in such cases, to selectively check the invariant in situations where one suspects an infected state.

4.4.4 Invocation-state extraction

A precise test extractor produces test cases that invoke routines from the state of their original invocation. Tests extraction occurs at the time of a failure. A precise test extractor must remember the invocation states for those routines for which it is producing test cases. Whenever a routine is invoked, the extractor must capture the relevant part of the state. When a failure occurs during the execution of a routine, the extractor uses the state captured previously to produce a test case. When the routine terminates, the extractor can forget the saved state. At any given time, the extractor remembers as many states as there are routines on the stack.

We call such a precise test extractor *invocation-state extractor*. Artzi et al. describe in detail how such an extractor can be implemented. Invocation-state extraction is expensive, because state needs to be serialized throughout normal program execution.

4.4.5 Failure-state extraction

We propose a more efficient extractor, the *failure-state extractor*. To avoid the repeated capturing of state, it invokes routines from the failure-state instead of the invocation state. During normal program execution, no state is captured. When a failure occurs, the state from this moment is captured. In the extracted test cases, each routine is invoked from the same state: the state from the moment of the failure. Failure-state extraction is potentially less effective than invocation-state extraction, because the routine invoked from the failure state may not reproduce the failure, but more efficient.

The following gives an example (based on one used previously in this chapter) of when the failure state extraction does not work. Consider class `BANK_ACCOUNT`
depicted in Listing 4.5. What happens if the developer were to run the application so that on an account with a zero balance, he is to withdraw 20 CHF? With an initial balance of zero and an \textit{an\_amount} of 20 the invariant check at routine entrance is OK, so is the precondition check. Then the implementation executes the subtraction and the \textit{balance} chances to -20. Because the postcondition is erroneous it is violated and a test case gets extracted using the default failure-as-prestate strategy. The actual problem in this scenario is that the precondition is too weak, but this fact is not of any concern for the pre-state extraction. The resulting test case (class \texttt{TEST\_CASE\_3} is seen below class \texttt{BANK\_ACCOUNT} in Figure 4.5).
Notice that the field \textit{balance} is initialized with -20, because this was the value of this field at the time of the failure. When this test case gets executed, it does not reproduce the original failure, because the invariant check that occurs before the routine \texttt{withdraw} gets executed is violated. The result is an unresolved test case. One where it could not be determined whether the implementation passes or fails the intended check.

\section*{4.5 Implementation}

We have extended EiffelStudio IDE\textsuperscript{1} with a failure-state extractor. Our modified version is called \textit{CDD EiffelStudio}\textsuperscript{2} and can be download from http://eiffelsoftware.origo.ethz.ch/index.php/CddBranch/ in both binary and source form. The code is distributed under an open source license. A screenshot of CDD EiffelStudio can be seen in Figure 4.4. Our implementation targets Eiffel code, because Eiffel natively supports contracts and real world source code equipped with contracts is broadly available. The implementation consists of approximately 50 new classes totaling to around 6000 lines of code. The addition is relatively small compared to EiffelStudio itself (approximately 1400 classes and 2 million lines of code) and, to keep maintenance efforts reasonable, we kept the number of existing classes that we modified to a minimum. The classes of the extension can be roughly divided into groups achieving the following:

- Model (internal representation of test cases)
- State and code extraction using the interface of the debugger
- Test case serialization
- Compilation and execution of the serialized test cases

\footnotetext[1]{http://www.eiffel.com}
\footnotetext[2]{CDD stands for Contract Driven Development}
class BANK_ACCOUNT
feature balance: INTEGER

withdraw (an_amount: INTEGER)
do
  balance := balance - an_amount
ensure
  balance_decreased: balance < old balance
  withdrawn: balance = old balance + an_amount
end
...

invariant
  balance_not_negative: balance >= 0
end

class TEST_CASE_3
feature
test
  local ba: BANK_ACCOUNT
  do
    ba := new_object ("BANK_ACCOUNT")
    set_field (ba, "balance", -20)
    check_invariant (ba)
    ba.withdraw (-20)
  end
end

Listing 4.5: Example where test case extraction fails. Executing withdraw with a balance of zero and a an_amount of -20 results in test case TEST_CASE_3, which when executed raises a invariant violation before executing the routine under test.
Test harness (unit testing framework)

Visualization and user interface

Example code

Our implementation has received extensive testing, ships with documentation, installer and a tutorial video. EiffelSoftware, the company developing EiffelStudio is at the time of this writing integrating our implementation into their mainline distribution. Test case extraction as proposed in this thesis will hence become available via the standard EiffelStudio release to all Eiffel developers.

The main supplier of the test extractor is the debugger. The test case extractor registers a callback with the debugger to get notified in the event of an exception. When it does get notified it uses the interface that the debugger normally uses to visualize the current state to the user to extract the test case. The debugger keeps track of both the stack and the heap. Both things are essential for the test extractor.
Figure 4.5: Architecture of test case extractor. The system under test is run in the debugger, which notifies the extractor in the event of an exception. The extractor serializes the current state and sends it to the test serializer which creates and executable test case out of the state and the stack information. The extracted test case is then sent to the compiler, which yields an executable. The executable is run and the test outcome is visualized in the developer’s IDE.

Figure 4.5 shows the basic control flow. The debugger is in control of the system under test (the application that the developer is working on). When a failure occurs, the state extractor retrieves information from the debugger about the current state of the application. The test case serializer saves this state into a compilable unit test case. The resulting test case is then compiled and executed and finally the results are displayed.

As already described, the main motivation of the model developed in Section 4.4 is to extract test cases in a completely automated way while developers program applications. Our approach relies on the presence of contracts. Contracts are first level citizens in this language and practitioners using the language are known to provide contracts in real world settings. This provides for a good setting in which the tool can be validated.

Our implementation poses no run-time overhead during debugging, since the extractor becomes active only at the time of an observed failure. At this point the debugger stops the application anyway, so extraction takes place when the application is not running. The application under test is not instrumented or altered in any way. This has the clear advantage of interfering as little with the developer’s working habits as possible.
Test case visualization  The extracted test cases are displayed in a tree where they are grouped by the class and the routine that they are testing (see Figure 4.2). The developer can choose to see all test cases or only failing ones. It is also possible to disable the background extraction and execution of test cases, or to select an individual test case for debugging (the latter is described in more detail below). Each test case node displays the status of its last execution and the assertion violation raised, if any. The developer can use each test case node to navigate to the source code of the test case or to the receiving routine.

Test case execution  Whenever the IDE successfully finished compiling the system, all extracted test cases for that system get compiled and executed. For each test case, the tool first recreates the context, then checks the invariant, and finally invokes the routine under test with the created context. If the invariant was found to be violated, the test case is flagged as unresolved and this test case is not executed. Otherwise, the routine under test is executed with the recreated target object and arguments.

During execution, assertion monitoring is enabled and violated assertions are reported in the form of exceptions back to the test case executor which determines the test outcomes.

Background execution of test cases allows the developer to always see the latest state of the test cases. However, in addition to this, the developer may want to more clearly understand why a particular test case fails. For this case our implementation allows him to execute a test case in the regular debugger. When doing this on a failing test case, the debugger will automatically stop at the exception being raised and thus allow the developer to inspect the concrete values. Additionally the developer can set breakpoints and thus step through the test case (including the routine under test) line by line and inspect the state at routine entry point and how this state evolves.

Maintenance issues  Users of early versions of our tool noticed that some failures happen over and over again, causing many redundant test case to be extracted. They reported that these duplicates makes it difficult to keep an overview of the test suite. In response to that, we have added a duplicate test case remover to later versions of our tool. Whenever a new test case is extracted it is compared against the existing ones. Only if no comparable test case exists, will the extracted test be added to the test suite.

Technical considerations

Our implementation of the test case extractor is solid and ready to be used by developers. There are however still technical limitations. While these limitations
exist in our current implementation, they are not inherent to the approach and should be fixed in a future version.

1. **Global State** – Test cases use the reflection mechanism of Eiffel to re-create state. The reflection mechanism in the version of EiffelStudio used (6.1) does not support writing to global state. Hence test cases are not able to properly initialize global state.

2. **Agents** – The test extractor is not able to serialize delegates, instead it will serialize the default value (`Void`). Test extraction may fail because of this.

3. **Bounded State** – The debugger is not optimized to retrieve large amounts of state from the debuggee. The communication line has been developed to show few variables at a time and not to transfer many objects at once. Extraction of large states takes a significant amount of time. To avoid large delays during extraction the extractor bounds the state both in the number of objects to retrieve and how deep it will traverse in the object graph. Having a faster serialization mechanism would allow us to remove this boundary.

### 4.6 Evaluation

We set out to answer the question of how many failures can be reproduced by test case extracted from a failure state. If a large proportion can be shown to be reproducible, this would be evidence that the run-time overhead of maintaining a shadow stack is not necessary.

#### 4.6.1 Experimental setup

We conducted a study with 59 6th semester students from ETH Zurich who took the *Software Engineering* class in 2008. The students were divided into 19 groups of 2-4 students. 10 groups had to complete an assignment involving a genealogy tree, and 9 different groups had to complete an assignment involving v-cards.\(^3\)

**genealogy** The goal of the genealogy assignment is to implement a genealogy database. The user must be able to search the database for various properties such as closest common ancestor. The students were required to write a library, and they were provided with an application that used this library.

v-card  The goal of the v-card assignment is to implement the v-card standard for electronic addresses. For this assignment, students were required to write both a library and an application that uses this library.

On average, solutions to the genealogy assignment consisted of 3284 lines of code and 11 classes while solutions to the v-card assignment consisted on average of 3995 lines of code and 34 classes. Table 4.1 conveys relevant statistics. Students had one month to complete their assignment.

Both applications made use of external events as discussed in Section 4.1 through user interfaces and file access as follows. Both assignments involved a text-based user interface where the user interactively inputs text. Moreover, both assignments contained features that required access to the file system: the v-card assignment included functionality to read and write v-cards to the file system, and the genealogy assignment included the feature to load commands from a batch file.

To work on their assignments the students were given an integrated development environment, CDD EiffelStudio. This IDE is equipped with a failure-state extractor, and we included a logging mechanism for information relevant to the experiment, including failures, extracted test cases, and information on whether or not tests were reproducing (the IDE ran the extracted test cases in the background directly after they had been generated). Students were required and agreed to commit these logs together with their program to a version control system.

Overall, 714 individual failures were logged. Many times similar failures appeared repeatedly. We assume this is because students executed their program several times, providing the same input. Such redundant failures add a bias: failures produced repeatedly are likely to be easier to reproduce. To remove this bias, we grouped similar failures into categories called unique failures. Two failures are unique if and only if they were triggered due to the same kind of exception and

\[\text{v-card} \] The goal of the v-card assignment is to implement the v-card standard for electronic addresses. For this assignment, students were required to write both a library and an application that uses this library.

On average, solutions to the genealogy assignment consisted of 3284 lines of code and 11 classes while solutions to the v-card assignment consisted on average of 3995 lines of code and 34 classes. Table 4.1 conveys relevant statistics. Students had one month to complete their assignment.

Both applications made use of external events as discussed in Section 4.1 through user interfaces and file access as follows. Both assignments involved a text-based user interface where the user interactively inputs text. Moreover, both assignments contained features that required access to the file system: the v-card assignment included functionality to read and write v-cards to the file system, and the genealogy assignment included the feature to load commands from a batch file.

To work on their assignments the students were given an integrated development environment, CDD EiffelStudio. This IDE is equipped with a failure-state extractor, and we included a logging mechanism for information relevant to the experiment, including failures, extracted test cases, and information on whether or not tests were reproducing (the IDE ran the extracted test cases in the background directly after they had been generated). Students were required and agreed to commit these logs together with their program to a version control system.

Overall, 714 individual failures were logged. Many times similar failures appeared repeatedly. We assume this is because students executed their program several times, providing the same input. Such redundant failures add a bias: failures produced repeatedly are likely to be easier to reproduce. To remove this bias, we grouped similar failures into categories called unique failures. Two failures are unique if and only if they were triggered due to the same kind of exception and

\[\text{v-card} \] The goal of the v-card assignment is to implement the v-card standard for electronic addresses. For this assignment, students were required to write both a library and an application that uses this library.

On average, solutions to the genealogy assignment consisted of 3284 lines of code and 11 classes while solutions to the v-card assignment consisted on average of 3995 lines of code and 34 classes. Table 4.1 conveys relevant statistics. Students had one month to complete their assignment.

Both applications made use of external events as discussed in Section 4.1 through user interfaces and file access as follows. Both assignments involved a text-based user interface where the user interactively inputs text. Moreover, both assignments contained features that required access to the file system: the v-card assignment included functionality to read and write v-cards to the file system, and the genealogy assignment included the feature to load commands from a batch file.

To work on their assignments the students were given an integrated development environment, CDD EiffelStudio. This IDE is equipped with a failure-state extractor, and we included a logging mechanism for information relevant to the experiment, including failures, extracted test cases, and information on whether or not tests were reproducing (the IDE ran the extracted test cases in the background directly after they had been generated). Students were required and agreed to commit these logs together with their program to a version control system.

Overall, 714 individual failures were logged. Many times similar failures appeared repeatedly. We assume this is because students executed their program several times, providing the same input. Such redundant failures add a bias: failures produced repeatedly are likely to be easier to reproduce. To remove this bias, we grouped similar failures into categories called unique failures. Two failures are unique if and only if they were triggered due to the same kind of exception and
Table 4.2: Statistics for original (top) data and its post-processed subset (bottom)

<table>
<thead>
<tr>
<th>criterion</th>
<th>original data</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>avg</td>
<td>med</td>
<td>max</td>
<td>stdev</td>
</tr>
<tr>
<td>per individual failure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># extracted tests</td>
<td>0.0</td>
<td>2.3</td>
<td>2.0</td>
<td>12.0</td>
<td>2.3</td>
</tr>
<tr>
<td>failure stack size</td>
<td>2.0</td>
<td>8.1</td>
<td>6.0</td>
<td>25.0</td>
<td>3.8</td>
</tr>
<tr>
<td>per unique failure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># individual failures</td>
<td>1.0</td>
<td>2.2</td>
<td>1.0</td>
<td>17.0</td>
<td>2.3</td>
</tr>
<tr>
<td>extracted tests</td>
<td>0.0</td>
<td>5.2</td>
<td>4.0</td>
<td>40.0</td>
<td>5.6</td>
</tr>
<tr>
<td>post-processed data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>per individual failure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># extracted tests</td>
<td>0.0</td>
<td>2.5</td>
<td>3.0</td>
<td>12.0</td>
<td>2.2</td>
</tr>
<tr>
<td>failure stack size</td>
<td>2.0</td>
<td>8.4</td>
<td>6.0</td>
<td>18.0</td>
<td>3.0</td>
</tr>
<tr>
<td>per unique failure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># individual failures</td>
<td>1.0</td>
<td>2.2</td>
<td>1.0</td>
<td>17.0</td>
<td>2.3</td>
</tr>
<tr>
<td>extracted tests</td>
<td>0.0</td>
<td>5.4</td>
<td>4.0</td>
<td>40.0</td>
<td>5.8</td>
</tr>
</tbody>
</table>

the recipient was the same routine. Categorization of the 714 individual failures resulted in 319 unique failures.

1664 tests were extracted with an average 2.3 tests per individual failure. The failures aggregated as one unique failure caused on average the extraction of 5.2 test cases. Relevant statistics are shown in Table 4.2, top. We show the numbers for both individual and aggregated (i.e., unique) failures. The stack size is not the same as the number of extracted test cases because the test case extractor ignored three classes of stack frames.

1. **External routines** – It ignored stack frames of external routine invocations. Such routines call C functions from within Eiffel. Since the implemented test extractor is not aware of and therefore cannot serialize the memory of these C functions, the extracted test cases would not be of much use.

2. **Agents** – It ignored stack frames of agents, because our implementation cannot handle them.

3. **Library routines** – It ignored stack frames of routines from the standard Eiffel libraries, since we did not expect developers to be interested in such test cases.

4. **Redundant test cases** – It did not add redundant test cases. If, due to a
Table 4.3: Reproduction results for original data (depth-limited extractor)

<table>
<thead>
<tr>
<th></th>
<th>original data</th>
<th>post-proc. subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>ind. failures</td>
<td>714</td>
<td>411</td>
</tr>
<tr>
<td>unique failures</td>
<td>319</td>
<td>189</td>
</tr>
<tr>
<td>– contract viol.</td>
<td>209 (66%)</td>
<td>125 (66%)</td>
</tr>
<tr>
<td>– other exc.</td>
<td>110 (35%)</td>
<td>64 (34%)</td>
</tr>
<tr>
<td>extracted tests</td>
<td>1664</td>
<td>1024</td>
</tr>
<tr>
<td>reproducing tests</td>
<td>390 (23%)</td>
<td>271 (27%)</td>
</tr>
<tr>
<td>repr. unique failures</td>
<td>139 (44%)</td>
<td>92 (49%)</td>
</tr>
<tr>
<td>– contract viol.</td>
<td>85 (41%)</td>
<td>58 (46%)</td>
</tr>
<tr>
<td>– other exc.</td>
<td>54 (49%)</td>
<td>34 (53%)</td>
</tr>
</tbody>
</table>

previous extraction an identical test case already existed, no new test case was generated.

An initial analysis of the logging data revealed that 139 (or 44%) of the unique failures were reproducible and that 390 (or 23%) of the test cases were reproducing. The middle column of Table 4.3 summarizes these numbers; the rows relating to contracts as well as the rightmost column will be discussed below.

These rather low numbers made us analyze the results. We realized that we had been too optimistic in terms of the necessary depth of serializing heap structures when the failure occurred. We had worked with a maximum depth of five consecutive links, and this turned out to be too low. In the genealogy assignment, for instance, students were happy to implement genealogies with more than four generations. Our analysis also indicated that we could expect much better results with more deeply serialized structures.

### 4.6.2 Failure-state extraction without depth limit

As a consequence, we decided to manually regenerate the failures and have the IDE re-extract the test cases, this time with a sufficient depth of serialized heap structures. Note that like the limited-depth extractor, this unlimited extractor does not introduce any overhead during regular program execution, but that it does of course introduce more overhead at extraction time. Since the manual regeneration turned out to be extremely labor-intensive, we could not post process all failures. We restricted ourselves to 7 out of the 19 groups and picked those groups that exhibited the most unique failures. From the 7 groups, 4 had implemented the genealogy assignment and 3 had implemented the v-card assignment. These groups had triggered 411 out of the entirety of 714 individual failures. Correspondingly,
Table 4.4: Statistics for manually regenerated failures (limited & unlimited depth)

<table>
<thead>
<tr>
<th>criterion</th>
<th>min</th>
<th>avg</th>
<th>med</th>
<th>max</th>
<th>stdev</th>
</tr>
</thead>
<tbody>
<tr>
<td>per individual failure (total 209)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># extracted tests</td>
<td>0</td>
<td>2.5</td>
<td>3</td>
<td>11</td>
<td>1.8</td>
</tr>
<tr>
<td>failure stack size</td>
<td>2</td>
<td>6.8</td>
<td>6</td>
<td>17</td>
<td>2.2</td>
</tr>
<tr>
<td>per unique failure (total 97)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># individual failures extracted tests</td>
<td>0</td>
<td>2.1</td>
<td>2</td>
<td>13</td>
<td>1.8</td>
</tr>
<tr>
<td>extracted tests</td>
<td>0</td>
<td>5.5</td>
<td>5</td>
<td>17</td>
<td>3.9</td>
</tr>
</tbody>
</table>

they had triggered 189 out of all 319 unique failures.

There is some evidence that the 7 groups are representative of all 19 (Table 4.3, rightmost column). In terms of reproducibility, using the depth-limited test case extractor, 49% of all unique failures from the 7 groups are reproducible. This is comparable to the 44% of reproducible failures from all groups. Conversely, the percentage of reproducing tests, 27%, seems sufficiently close to the original 23%. Moreover, the statistics for stack sizes, number of extracted tests, and the number of individual failures that correspond to one unique failure are very close to the statistics of the original data set, as shown in Table 4.2, bottom.

Our primary interest is in determining how many failures can be reproduced rather than in how many test cases reproduce the failure from which they were extracted. This allowed us to reduce the number of failures we had to manually regenerate: it is safe to assume that all failures reproducible with the depth-limited extractor (92 out of 189) are also reproducible with the unlimited extractor. As a consequence, we only needed to manually regenerate those failures that were not reproducible with the depth-limited extractor—because of our assumption we already knew that they were reproducible with the unlimited extractor. This left only 97 unique failures that we had to manually regenerate. These correspond to 210 individual failures. We proceeded as follows.

1. For every unique failure, we determined its first occurrence in the log files.

2. We retrieved the version of the program that was committed to the version control system as close as possible to the date and time of the occurrence of this failure.

3. We ran the program and provided input that we thought would reproduce the failure. This task was rendered feasible with the information from the stack trace of the failure that we were trying to regenerate.

4. When we were able to manually regenerate a failure, we made sure the failure was similar in nature to the one we were after by first applying it to
the depth-limited failure-state extractor. We considered the failures equal only if the depth-limited failure-state extractor indeed failed to extract reproducing test cases for the regenerated failure. Otherwise, we searched for different inputs until we could regenerate the exception.

Each previously non-reproducible failure was regenerated (at least) two times: once applying the limited extractor, and once applying the unlimited extractor. The regeneration of non-reproducible failures resulted in a total of 530 test cases out of which 148, or 28%, reproduced the failure they were extracted from, with an average 2.5 tests extracted per individual failure and an average 5.5 tests available per unique failure. Table 4.4 shows relevant statistics for the manually regenerated failures and the corresponding tests. Note that it does not include the failures we did not have to regenerate because there was no problem with the depth limit when copying heap structures (see Table 4.5 below). Also note that the minimum number of failures per unique failure is zero. This captures the fact that we did not succeed in regenerating all unique failures, which was the case for 12 of them. We counted these unique failures as non-reproducible, thus introducing a slight negative bias against failure-state extraction.

Recall again that we have argued that for developers, it is important whether or not a failure is reproduced, not so much by how many test cases it is reproduced. Accordingly, we found that out of the 97 failures that we had to regenerate manually because of the limited extractor, 77, or 80%, were reproducible with the failure-state extractor, i.e., there was at least one extracted test case that could reproduce it. The second column of Table 4.5 shows relevant data.

Figure 4.6 shows how the unique failures relate to each other. Numbers on the left relate to the original data obtained with the limited-depth test case extractor. Numbers to the right relate to the subset that we post-processed with the unlimited depth extractor.

Note that the above considers only those failures that were irreproducible with the depth limited failure-state extractor. Since we assume that any failure repro-
Table 4.5: Reproduction results for post-processed data

<table>
<thead>
<tr>
<th></th>
<th>manually regenerated</th>
<th>no regen.</th>
<th>both</th>
</tr>
</thead>
<tbody>
<tr>
<td>ind. failures</td>
<td>209</td>
<td>201</td>
<td>410</td>
</tr>
<tr>
<td>unique failures</td>
<td>97</td>
<td>92</td>
<td>189</td>
</tr>
<tr>
<td>extracted tests</td>
<td>530</td>
<td>602</td>
<td>1132</td>
</tr>
<tr>
<td>reproducing tests</td>
<td>148</td>
<td>271</td>
<td>419</td>
</tr>
<tr>
<td>% repr. tests</td>
<td>28</td>
<td>45</td>
<td>37</td>
</tr>
<tr>
<td>repr. unique failures</td>
<td>78</td>
<td>92</td>
<td>170</td>
</tr>
<tr>
<td>% repr. unique f.</td>
<td>80</td>
<td>100</td>
<td>90</td>
</tr>
</tbody>
</table>

produced by the limited failure-state extractor is also reproduced by the unlimited failure-state extractor, we can combine the numbers to learn how the unlimited failure-state extractor performs on all failures of the selected 7 groups: 92 failures were reproduced by the limited extractor and 78 by the unlimited extractor, leaving 19 non-reproduced by either (Table 4.5, third column). Hence, in our experiment that is restricted to 7 out of the 19 groups of student programmers, the unlimited failure-state extractor reproduced 170 out of 189 unique failures, or 90%.

4.6.3 Invocation-state extraction

In order to find out how much more effective invocation-state test extraction is when compared to the far more efficient failure-state extraction, we iterated on the approach described in Section 4.6.2. As discussed above, we manually regenerated the unique failures for which the limited depth extractor did not extract a reproducing test (making sure the exception is similar in nature to the original one) and then used an invocation-state extractor to produce new test cases. For fair comparison, we considered the same 7 out of 19 groups of student programmers. We assumed (1) that any failure reproducible by the unlimited failure-state extractor was also reproducible by the invocation-state extractor and, once more, (2) that any failure reproducible by the limited failure-state extractor was also reproducible by the unlimited failure state extractor. As a consequence, we used the invocation-state extractor to reproduce only those 19 unique failures that were not reproducible with the unlimited failure-state extractor. The right column of Table 4.6 shows the results. Applying the invocation-state extractor to these remaining 19 unique failures rendered another 7 reproducible, or 3.7 percent points. Overall, test case extraction could hence reproduce 177 unique failures, or 94%. Incidentally, the 12 missing unique failures are exactly those that we could not reproduce manually, and for which the invocation-state extractor could equally
Table 4.6: Regeneration results by exception kind (post-processed)

<table>
<thead>
<tr>
<th></th>
<th>failure-state extraction</th>
<th>invocation-state extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique failures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>– contract violations</td>
<td>125 (66%)</td>
<td>125 (66%)</td>
</tr>
<tr>
<td>– other exceptions</td>
<td>64 (34%)</td>
<td>64 (34%)</td>
</tr>
<tr>
<td>repr. unique failures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>– contract violations</td>
<td>112 (90%)</td>
<td>118 (94%)</td>
</tr>
<tr>
<td>– other exceptions</td>
<td>58 (91%)</td>
<td>59 (92%)</td>
</tr>
</tbody>
</table>

not be applied. For these 12 unique failures, we do not know if invocation-state extraction or failure state extraction or both would be able to generate reproducing tests.

4.6.4 Contract vs. other runtime failures

The failures considered in this study stem from programs that are annotated with contracts. We were interested in understanding to what extent our results can be generalized to programs that do not contain specification annotations. To answer this question, we divided failures into two categories. **Contract violations** correspond to failures that stem from contract violations (i.e. precondition violations). Such failures only occur in programs with assertion annotations. **Other violations** correspond to failures that stem from other runtime-violations (i.e. null pointer dereferences). Such failures occur also in programs without assertion annotations.

Table 4.6 shows the reproducibility of contract violations and violations due to normal exception separately. The 189 unique failures processed with the unlimited failure-state extractor can be divided into 125 contract violations and 64 other violations. The failure-state extractor reproduced 90% of the contract violations and 91% of the other violations (see Table 4.3 for respective statistics of the original data set). The invocation-state extractor reproduced 94% of all contract violations and also 92% of all other violations. For both test case extractors the difference between contract failures and other failures is less than 3%. **In our experiment, there is no difference between failures that are a result of contract violations and other exceptions.**

4.6.5 Unique failures

We have introduced the notion of unique failures to avoid introducing a bias. If a failure is easy to reproduce and occurs frequently, it would add a positive bias. If a failure is hard to reproduce and occurs frequently it would add a negative bias. Unique failures group exceptions that
1. indicate the same runtime problem (e.g., precondition violation or division-by-zero) and

2. have been triggered by the same location in the source code.

The idea is that unique failures group failures with similar properties. Two failures stemming from different faults are likely to have different properties. When grouped by kind and location it is technically possible that two failures that stem from distinct faults are grouped together. For example, assume a routine \( a \) which takes an argument, that must not be null. Two distinct routines may each invoke \( a \) with null as argument. There are two faults, one in each calling routine, but the two failures are put into the same group. The fact that the source code from our experiment was annotated with contracts in the form of preconditions, postconditions and invariants lowers the chances of such a scenario. Routines that do not accept, say, null arguments often state this in their precondition. Since the recipient for a precondition violation is the calling routine no test case is generated for the called one. Hence the two failures from the above example would end up in two distinct groups.

While contracts help they do not remove the chance that two failures with dissimilar properties are put into the same group. We studied the relevance of this potential issue by considering the following criteria. Unique exceptions are equivalence classes. In our context, two members of the same equivalence class should lead to qualitatively similar test cases. To capture this fuzzy notion of qualitative similarity, we considered the following criteria. We consider two tests to be qualitatively similar if

1. the failure stacks of the corresponding individual exceptions have the same height; and

2. two frames of the failure stacks of the corresponding individual exceptions at the same vertical position correspond to the same routine—routine arguments are not considered; and

3. the tests generated from two frames at the same vertical position in the failure stack of the corresponding individual exceptions either both pass or fail.

These criteria capture the intuition of similar tests. Criteria (1) and (2) approximate that the control flow of the program execution until the failures are similar, and criterion (3) very roughly approximates that the data flow problems in the two executions are similar.

In our experiments, a large majority—277 out of 319, or 87%—of the unique exceptions satisfies these three criteria (the first and second criterion together are
satisfied by 89%). In other words, there is some evidence that in our experiment, the kind and location of a raised exception are good determinants for equivalence classes because these two criteria relate to the nature of the generated test cases rather than the exceptions themselves.

4.6.6 Interpretation and consequences

Out of the 189 considered unique failures, we could not manually reproduce 12. The unlimited failure-state extractor generated 170 reproducing tests. Invocation-state extraction led to another 7 reproducing tests. Due to the problem with limited depth of copying heap structures, we had to group individual failures into classes, so-called unique failures, and we have argued that the simple definition of these classes can be justified. Failure-state test case extraction is both effective and efficient.

We conjecture but do not know for sure if the 12 unique failures that we could not regenerate—and for which we do not know if any of the extractors would be able to generate reproducing tests—were not reproducible because of external events.

Modulo the threats to validity of a generalization of our results that we discuss below, our experiments very clearly indicate that the huge runtime overhead of invocation-state extraction cannot be justified. A mere additional 4 percent points of reproducible failures will, in most situations, not be worth several orders of magnitude of slow-down.

CDD EiffelStudio now contains a hybrid text extraction approach that combines failure-state test extraction with Second Chance. If a failure occurs, the failure-state extractor generates test cases (without imposing run-time overhead). The IDE automatically executes the new test cases. If the failure is reproduced by at least one test case nothing else happens. If the failure is not reproducible, the routines that were on the stack at the time of the failure are marked as observed. From now on, the invocation-state extractor is activated, but restricted to routines marked as observed. Whenever a routine marked as observed is entered, its relevant state is captured. For other routines nothing happens. If the failure that was not reproducible with the failure-state extractor occurs a second time, the extractor uses the previously saved invocation states to generate test cases. The resulting test cases are identical to the ones generated by a pure invocation-state extractor, but because of Second Chance the extractor requires much less overhead. As soon as a failure is reproduced via a test case, the routines previously marked as observed are unmarked. To reduce run-time overhead, the developer may also unmark routines manually.
4.6. EVALUATION

4.6.7 Threats to validity

The most obvious threat to the validity of a generalization of our results lies in the restricted class of programs, both in terms of their application domain and the language that we used. In particular, state-based test case extraction is naturally restricted by external events, as discussed in Section 4.1. We do not know if our results generalize to programs with more frequently occurring external events. Another apparent threat is the size of the programs used in the study. The two programs were medium sized, and while they are not trivial, they are not comparable to large multi-million line industrial projects. The number and kind of failures occurring during development of a small project may not be representative for large projects.

Two threats stem from our manual regeneration of failures. First, we limited manual regeneration to the 7 (out of 19) groups with the most failures. The failures of these groups might not be representative of the failures of all 19. We assume that they are, because (as shown in Table 4.3) the properties of the unprocessed data of the 7 groups is similar to the properties of all 19 groups. Second, the failures we manually regenerated might be different from the original ones. To address this concern, we thoroughly compared the failures we manually regenerated to the original ones, as described in Section 4.6.2.

The conclusions drawn in this study assume that for a developer it is sufficient that a failure is reproduced by even a single test case. Not all test cases are equally useful to the developer, however. Failures reproduced only by tests useless to the developer should not be counted as reproduced. For example, a test case might reproduce the failure, but not execute the code containing the fault. A test case could invoke a routine with a null argument, but this routine is not supposed to handle null arguments. Since the present study considers contracted programs, it is not affected by this problem as much as if it would consider programs without contracts. Routines not accepting null arguments often state this in their precondition, which is checked before the routine is executed. As described in Section 4.4 the proposed extractor does not create test cases for routines that have their precondition violated.
CHAPTER 5

SELECTING AND PRIORITIZING TESTS

This chapter presents a novel change-based test case selection and prioritization algorithm. Section 5.1 explains how the inheritance and association relation help to detect relevant test cases. Section 5.2 gives the test selection algorithm and explains how it can be implemented efficiently. Section 5.3 shows how the algorithm can be extended to prioritize tests. Section 5.4 presents an experimental evaluation of the performance of the proposed test selection algorithms.

Modern development methods depend on frequent test execution. The extreme case of frequent testing is continuous testing, where test execution happens automatically in the background. Starting automatically after every change in the source code. Continuous testing was found to increase development productivity significantly when compared to the traditional case where tests execution is triggered manually [89].

When tests are executed frequently or even all the time they must be fast executing, or they will not give the developer feedback early enough. As a test suite grows, its execution time increases. Eventually it will take longer for the test suite to execute than it will take the developer to make a change to the program. Then the programmer either has to wait until all tests are executed, or he risks not being notified of a regression while he is still in the right mindset. Not all test cases must be executed all the time however. Test cases who’s outcome is not affected by a given change don’t need to be re-run after this change has been made. This applies in particular to unit tests, where each test is targeted only towards a particular unit of the program. Excluding irrelevant test cases decreases the execution time of a test suite and increases the productivity of the developer.

An algorithm identifying relevant test cases for a given change in a program should be:
effective – Identify the relevant test cases to re-run. This is the set of test cases that yield a different outcome when run against the program with the change and without the change applied.

efficient – Find the relevant test cases fast enough. If it takes longer to find the relevant test cases than it takes to execute the irrelevant ones no time is saved and applying the algorithm useless.

In general it is not possible to automatically and precisely determine which test cases must be re-run and which test cases can be excluded safely. Knowing which test cases will yield a different outcome implies knowing the outcomes without running the tests then. It would be unnecessary to even run the tests. Since it is necessary to run them, the precise set of relevant tests cannot be known in advance.

We present a test case selection algorithm that approximates the relevant test cases in a very efficient manner. The proposed algorithm is based on an approximation of the inheritance and association relation and requires only lexical analysis of the source code. Lexical analysis of a program is much faster than a full parse of it, which is the main reason why our proposed algorithm is efficient.

The content of this section is based on our publication in the proceedings of the 40th Hawaiian International Conference on System Sciences (HICSS-40) [66]. The section first describes the algorithm intuitively via an example, then provides a more formal description, and finally it shows how the proposed algorithm can be implemented efficiently.

5.1 Intuition of test selection

This section describes how the proposed test selection works via the use of an example. Figure 5.1 shows the UML diagram of the example used throughout this section. The classes PERSON, CURRENCY, BANK and BANK_ACCOUNT make up the system under test. The classes TEST_BANK_ACCOUNT, TEST_CURRENCY and TEST_PERSON form the corresponding test suite. They are all descendants of class TEST_CASE. Class TEST_BANK_ACCOUNT is a client of BANK_ACCOUNT, class TEST_PERSON is a client of PERSON, and class TEST_CURRENCY is a client of CURRENCY.

The tests are normal unit test cases as produced when using an xUnit framework. Whether the tests are written manually or have been created or extracted automatically is not of concern for test case selection.

What test cases are relevant for a change and what test cases are not? Suppose the developer has changed both class PERSON and class CURRENCY. The relevant
5.1. INTUITION OF TEST SELECTION

5.1. INTUITION OF TEST SELECTION

Figure 5.1: UML class diagram showing relation of classes under test and test classes. Test classes inherit from common root class and use the system under test.

test cases are detected through the two fundamental relations of object-oriented programming as follows:

- Inheritance is used to mark which classes are unit tests
- Association (client-of) is used to determine which unit tests apply to the classes in the test scope.

The inheritance relation is commonly used in manual unit testing frameworks. Unit test cases are required to inherit from a common ancestor, `TEST_CASE` in our example. This common ancestor provides convenience routines and prototypes for setup and tear-down hooks. Our use of the client-relationship to reduce the number of relevant manual unit tests is novel.

We detect the complete set of test cases (in the example `TEST_BANK_ACCOUNT`, `TEST_CURRENCY` and `TEST_PERSON`) using the inheritance relation. To reduce the number of test cases we only select those test cases that are clients of a class in the scope: `TEST_CURRENCY` and `TEST_PERSON`.

In most unit test suites a given test case class is dedicated to testing one class from the system under test. Any such test case class is obviously a direct client of its class under test; we will call it immediately relevant. The notion of immediate relevance is naturally extended by including tests that are directly or indirectly clients of a class under test. This extension, called transitive relevance, can capture subtle interactions between classes that may not be caught by examining only immediate clients. In our example the set of transitively relevant test
cases would also include class `TEST_BANK_ACCOUNT` because it is a client of class `BANK_ACCOUNT`, which is a client of class `CURRENCY`, which is in the test scope. Test case selection based on transitive relevance ensures that all test cases that may exercise a class from the scope are selected.

5.2 Test selection algorithm

This section formalizes the notions immediate and transitively relevant and our strategy for test case selection and shows how they can be implemented efficiently.

Let $C$ be the set of classes making up the system under test and $tc \in C$ be the abstract test case class from which every manual test case class inherits. We need notations for the inheritance and client-of relations:

**Definition 1 (inheritance).** Let $\text{inh}$ be the inheritance relation such that for any two classes $a, b \in C$:

$$a \text{ inh } b \iff a \text{ is a direct descendant (i.e. subclass) of } b.$$ 

The set of manual test cases $T$ is defined as $T \triangleq \{ tc \in C \mid tc \text{ inh}^+ tc \}$ where $\text{inh}^+$ denotes transitive closure of the relation $\text{inh}$. Hence, $T$ is the set of all manual test cases from the system under test.

**Definition 2 (client-of).** Let $\text{co}$ be the client relation such that for any two classes $a, b \in C$:

$$a \text{ co } b \iff a \text{ is a direct client of } b.$$ 

Furthermore, $a$ is an indirect client of $b \iff a \text{ co}^* b$ (where $\text{co}^*$ denotes the reflexive, transitive closure of the relation $\text{co}$).

We denote the inverse of the client-of relation, called supplier-of, by $\text{so}$.

Let $S \subseteq C$ be the given test scope, i.e. the set of classes that were changed by the developer. The sets of immediately and transitively relevant test cases are defined as:

**Definition 3 (immediately relevant)** $T_{\text{immediate}} \triangleq \{ tc \in T \mid \exists s \in S : tc \text{ co } s \}$

**Definition 4 (transitively relevant)** $T_{\text{transitive}} \triangleq \{ tc \in T \mid \exists s \in S : tc \text{ co}^* s \}$

Given a class $s$ in the test scope, computing all test cases that are clients of $s$ requires the traversal of all classes in $C$. However, to compute the suppliers of a test case $tc$ that are in the test scope, it is sufficient to traverse $tc$ and all its direct and indirect parents. In order to be more efficient, in our implementation we use the supplier-of relation to compute $T_{\text{immediate}}$. 
While set \( T_{\text{immediate}} \) can be calculated efficiently using the supplier-of relation, to obtain \( T_{\text{transitive}} \) we need to compute the transitive closure of the suppliers of all test cases, which involves finding all indirect suppliers of these test cases. In large and highly interconnected systems this can lead to a significant overhead as it requires the whole system to be parsed.

To improve performance when computing \( T_{\text{transitive}} \), we use a reasonable approximation of the relation \( so^* \). We define this approximation based on some observations about constants:

- Constants have types that are defined in a core library of the language.
- Core libraries are self-contained, meaning they do not depend on classes external to the library.
- Core libraries do not need testing except from the compiler provider.

Thus, when computing \( so^* \) it is sufficient to consider classes outside of the core library.

Let \( so_{\text{nc}} \) be the relation between classes such that for \( a, b \in C \), \( a so_{\text{nc}} b \iff a \) is a supplier of \( b \) that does not occur in a core library. Given a class \( b \), the direct and indirect suppliers of \( b \) that do not occur in a core library are the classes \( a \) such that \( a so_{\text{nc}}^* b \).

The computation of the relation \( so_{\text{nc}}^* \) is also expensive, since it requires information available only after type checking. Note that this is also true for the relations introduced above. There exists an over-approximation of \( so_{\text{nc}}^* \), based on purely lexical analysis, which makes it cheap to compute:

- Given a class a mark all type names occurring in the text of a as belonging to the result set.
- Process all type names occurring in the text of a recursively.

This over-approximation can be used to compute \( T_{\text{transitive}} \) more efficiently as it requires only lexical analysis. Lexical analysis is not dependent on the size of a program, only on the size of the class to parse, which allows our algorithm to scale well.

### 5.3 From test selection to test prioritization

As described, both \( T_{\text{immediate}} \) and \( T_{\text{transitive}} \) are algorithms that select test cases relevant to a certain change in the system (or test scope). Our approach can however also be used to prioritize tests. The intuition is that the further away a test from the test scope the lower its priority.
A coarse grained way to do this is to give all tests in $T_{\text{immediate}}$ priority 1 (here a lower priority number means higher priority) and tests that are in $T_{\text{transitive}}$ (but not also in $T_{\text{immediate}}$) priority 2. All remaining tests receive priority 3. As a result first, tests which immediately exercise classes under test are executed, followed by tests that indirectly exercise classes under test, followed by all other tests.

Finer grained prioritization can be achieved by further dividing the tests that are in $T_{\text{transitive}}$: the more indirections between a test and the scope, the lower its priority.

### 5.4 Evaluation

This section presents a performance evaluation of the proposed test selection algorithms. We have implemented both $T_{\text{immediate}}$ and $T_{\text{transitive}}$ in the AutoTest framework. AutoTest is available both in binary and source form under an open source license.\(^1\)

#### 5.4.1 Setup

To evaluate the algorithms we measured their running time when applied an Eiffel library and a test suite. We used the EiffelBase library, a library employed by many Eiffel applications, covering fundamental data structures and algorithms. It contains 297 classes. We created a test suite consisting of 571 tests and 10726 lines of code using a random test synthesizer. The synthesizer is described in Section 6. Parsing of EiffelBase and its test suite takes approximately 4 seconds. A full compilation (including binary code generation) takes a few minutes. Subsequent compilation, which are incremental, are then in the range of a few seconds again. The 571 tests execute in approximately 98 seconds. Table 5.1 gives an overview of the setup.

#### 5.4.2 Results

The results of the evaluation are given in Table 5.2. Every row indicates one experimental run of both $T_{\text{immediate}}$ and $T_{\text{transitive}}$. The leftmost column shows for each experimental run, which classes where assumed to have changed. As expected $T_{\text{immediate}}$ selects fewer test cases than $T_{\text{transitive}}$. Basic classes, like $\text{STRING}$ are used indirectly by almost every class. This explains why $T_{\text{transitive}}$ of class $\text{STRING}$ selects almost all tests (562 of 571).

\(^1\)http://se.ethz.ch/research/autotest/
5.4. EVALUATION

| Number of classes in SUT | 297  
|--------------------------|------|
| Parsing time             | 4.4 sec  
| Number of tests          | 571  
| Number of lines of test code | 10726  
| Execution time of tests  | 97.7 sec  
| Avg. execution time of one test | 0.17 sec  

Table 5.1: Experimental setup

<table>
<thead>
<tr>
<th>Classes changed</th>
<th>$T_{immediate}$</th>
<th>$T_{transitive}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#tests selected</td>
<td>time (sec)</td>
</tr>
<tr>
<td>STRING</td>
<td>40</td>
<td>1.0</td>
</tr>
<tr>
<td>DIRECTORY</td>
<td>9</td>
<td>1.0</td>
</tr>
<tr>
<td>FIXED_LIST</td>
<td>39</td>
<td>1.0</td>
</tr>
<tr>
<td>LINKED_LIST</td>
<td>2</td>
<td>0.9</td>
</tr>
<tr>
<td>STRING, DIRECTORY</td>
<td>42</td>
<td>0.9</td>
</tr>
<tr>
<td>FIXED_LIST, LINKED_LIST</td>
<td>41</td>
<td>1.0</td>
</tr>
<tr>
<td>all 4 classes</td>
<td>83</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 5.2: Results of evaluation of test selection

The running time of both $T_{immediate}$ and $T_{transitive}$ does not seem to be influenced by the number of classes assumed changed. It takes approximately twice as long to execute $T_{transitive}$ as $T_{immediate}$. The time to calculate either is lower than the time for a full parse of the system and much lower than the time to execute all tests. After a change to the highly connected class STRING, calculating $T_{immediate}$ and executing all 40 selected tests takes 7.8 seconds (assuming an average test execution time of 0.17 seconds). This is a huge time saving over the normal running time of 97.7 seconds. Calculating $T_{transitive}$ and executing the 562 selected tests takes 97.64, which is almost no gain. For less interconnected classes both $T_{immediate}$ and $T_{transitive}$ provide savings. If class DIRECTORY changes, the time to calculate $T_{immediate}$ and to execute all 9 selected tests is 2.53 seconds. Calculating $T_{transitive}$ and executing the same 9 tests takes 3.63 seconds.
CHAPTER 6

CREATING NEW TESTS

This chapter presents how new tests are synthesized with the intent to reveal failures. It also shows how they are then minimized to reduce their cost of maintenance.

It is organized as follows: Section 6.1 introduces AutoTest, a fully automated contract-based testing tool. The section introduces the tool and shows how it is used. It shows the AutoTest architecture, which allows adding new test synthesis strategies. Section 6.2 motivates the notion of unit test used in AutoTest. Section 6.3 explains the default synthesis strategy, which is based on random testing. Section 6.4 explains how reusing objects for several tests increases the diversity of inputs. Section 6.5 gives a summary of the main findings of an evaluation of random-testing with AutoTest, which was executed as part of the thesis of Ilinca Ciupa [30].

Section 6.6 motives the need for test case minimization. Section 6.7 explains how slicing can be used to minimize test cases. It proposes an unsound but efficient algorithm. Section 6.8 evaluates the proposed algorithm. Section 6.9 explains how delta debugging can be used to minimize test cases. In an evaluation, delta debugging proves to be more effective than slicing, but much less efficient. Section 6.10 proposes a joint approach of both slicing and delta debugging, that is as effective as delta debugging, but much more efficient.

The content of this chapter is based on our publication in the proceedings of SOFSEM 2007, “Current Trends in Theory and Practice of Computer Science” [79], our publication in the proceedings of the “IEEE International Conference on Software Testing, Verification and Validation 2008” (ICST’08) [32], and our publication in the proceedings of the 22nd IEEE/ACM International Conference on “Automated Software Engineering” (ASE’07) [69].
6.1 AutoTest framework

The AutoTest tool is, at its core, an automated testing framework that produces systematic tests from contracts of object-oriented programs. AutoTest is the last in a series of tools we built, that all implement contract-based testing. Many aspects (most importantly the actual testing strategy) can be plugged in as needed. The complete source of AutoTest can be downloaded from the tool homepage\footnote{http://se.ethz.ch/research/autotest/} where it is available under an open source license. “Automatic”, when applied to AutoTest, should be understood in the full “push-button” sense of the term: all a user must specify is the set of classes that he wants to test; then AutoTest will test these classes automatically, without requiring any intervention of the user, such as preparing test cases.

6.1.1 A tour of AutoTest

The following shows an example invocation of AutoTest using the command line front-end:

```bash
auto_test time_limit compilation_options BANK_ACCOUNT STRING LINKED_LIST
```

where `time_limit` is the maximum time given to AutoTest for exercising the software, `compilation_options` is the control file governing normal compilation of a system (it does not need to be changed for testing), and the remaining arguments are names of classes to be tested. AutoTest will also exercise any other classes on which these classes depend directly or indirectly, for example library components.

This information suffices to specify an AutoTest session which will, in the prescribed time, test the given classes and their features, using various heuristics to maximize testing effectiveness; it will evaluate the contracts and log any contract violations or other failures. At the end of the session, AutoTest reports all failures, using by default the HTML format illustrated in Figure 6.1. Most failures reflect bugs in the software.

In the example results, all three classes under test are marked in red. Such a mark indicates that a test case for one or more features triggered a failure. (Classes marked with a green ellipse indicates that no bugs have been found in that class and classes mark in yellow indicate that no bugs have been found, but that there were unresolved tests.) Expanding the tree node of a class displays the routines of a class as child nodes: for `BANK_ACCOUNT` routine `make` was OK (green), but `deposit` had failures. Clicking the name of this feature displays, on the right
Figure 6.1: Results of an AutoTest testing session. The left panel allows navigation through the system under test and the main panel shows the test results of the selected class, cluster, or routine.
side of the figure, the details of these failures. For each failed test, this includes a witness: a test scenario, automatically generated by AutoTest, which triggered the failure. Listing 6.1 shows the witness for BANK_ACCOUNT.deposit.

```
feature
  test is
    do
      create (PERSON) v_66.make_default
      create (BANK_ACCOUNT) v_67.make(v_66)
      v_67.deposit(-1)
    end
```

Listing 6.1: Automatically generated test witness leading to a failure in class BANK_ACCOUNT.

This generated code creates an instance of PERSON and another of BANK_ACCOUNT (whose creation refers to the person, v_66, as the owner of the account), then deposits a negative amount – causing a precondition violation in a called routine. Figure 6.2 depicts the failure trace of the witness when run against the system under test.

Figure 6.2: Failure trace of the generated test case. It depicts the stack trace at the time of the failure. (Stack grows upwards)

Failure witnesses, as in Listing 6.1, appear in a minimized form. AutoTest may originally have detected the failure through a longer sequence of calls, but will then compute a minimal sequence that leads to the same result. This helps AutoTest users understand and correct the bug by reducing it to a simpler case. Minimization is also critical for regression testing: AutoTest records all failure scenarios and re-tests them in all subsequent test runs; but since the path to a failure often goes through many irrelevant detours, it is essential to remove these detours, keeping only the instructions that take part in causing the failure. Minimized tests make regression testing faster and the tests more resilient to code
6.1. AUTOTEST FRAMEWORK

changes in the system under test. AutoTest features minimizers based on static slicing, delta debugging and a combination of both. Minimization is further described in Section 6.6.

In the case of deposit in class `BANK_ACCOUNT` the bug was planted, for purposes of illustration. But AutoTest routinely finds real, unexpected bugs in production software. This is indeed apparent in the example where `STRING` and `LINKED_LIST`, basic library classes, appear in red. The corresponding bugs were unknown until AutoTest was applied to these classes. In each case they affected a single feature. For `STRING` the feature is `adapt`. Figure 6.3 shows the test case and the failure trace.

**Bug explanation:** `adapt` is a little-used operation, applicable when a programmer has defined a descendant `MY_STRING` of the basic library class `STRING`; it is invalid to assign a manifest string written in the usual notation “Some Explicit Characters”, of type `STRING`, to a variable of type `MY_STRING`, but `adapt` will yield an equivalent object of type `MY_STRING`. Routine `adapt` should include a precondition requiring a non-void argument. Without it, `adapt` accepts a void argument but passes it on to a routine `share` that demands non-void.

While AutoTest, as illustrated, provides extensive mechanisms for automated test generation, the framework also supports manual tests. The two approaches...
are complementary, one providing breadth, the other depth: automatic tests are
good at exercising components much more extensively than a human tester would
ever accomplish; manual tests can take advantage of domain knowledge to test
specific scenarios that an automatic mechanism would not have the time to reach.

6.1.2 AutoTest architecture

AutoTest is a framework for fully automated software testing. It allows for arbi-
trary testing strategies to be plugged in and is not hard coded to a certain testing
strategy. The pluggable testing strategy is only concerned with determining ex-
actly how and with what inputs the system under test should be invoked. The
actual execution and outcome evaluation is a task of the framework. Figure 6.4
shows the coarse grained architecture of AutoTest. The following describes the
illustrated components in more detail:

- Testing strategy: pluggable component that determines what instructions
  should be executed on the system under test. A testing strategy receives the
  AST of the system under test and the test scope, i.e., the set of classes that
  should be tested. It then uses this information to synthesize test cases which
  it sends to the interpreter for execution. The strategy provided by default
  creates test cases that use random input to exercise the classes under test.
  In addition to this default strategy two other strategies were developed: A
  strategy based on constraint solving and proof techniques [72], a strategy
  based on Adaptive Random Testing [31]. Section ?? describes the random
  testing strategy in detail.

- Interpreter: Executes instructions on the system under test. The interpreter
  lives in a separate process to increase robustness (see Chapter 7). Typical
  instructions for the interpreter are: create object and invoke routine.

- Object Pool: The object pool is a store for objects. All objects that were
  used for testing are put there. When the testing strategy creates a new test,
  it can either create a new object or re-use an object already used for test-
  ing from the pool. This gives the testing strategy access to both new and
  unmodified and old and modified object.

- Oracle: The oracle is based on the idea of contract-based testing as de-
  scribed in the preliminaries. It receives execution results and determines
  the outcome of the execution. The oracle then writes the testing results in
  the form of HTML documents to the hard disk. For every failing test it also
  produces a unit test in the form expected by the manual strategy. This way
  failing test cases can be kept and maintained just like regular test cases.
Figure 6.4: Architecture of AutoTest: The testing strategy analyzes the program under test and comes up with testing instructions, which it sends to the interpreter for execution. The interpreter employs a store for objects, in which it puts all objects used for testing, called the object pool. Using this pool, new tests can re-use objects from earlier tests. The interpreter sends information about the execution back to both the testing strategy (to allow for on-the-fly testing strategies) and to the oracle. The oracle uses this information to decide if a test passed or failed and produces the testing results.
6.2 Unit tests

Having an intuitive understanding of unit testing is rather straightforward: the purpose of unit testing is to verify the run of a certain program unit. We hence have to first discuss the notion of units in O-O software, including the notions of input and oracles.

1. In practice the purpose of a unit test (xUnit-style) is most often to verify the run of a particular method. Typically this is reflected in the names of manually written unit tests, when, for instance, a method \( x \) is tested by the test method named \( \text{test}_x \). What happens however if method \( x \) itself calls other methods (e.g. method \( y \))? In a strict interpretation of unit testing these other methods are not under test and must be replaced by simple stubs that are hard-coded to work with the given test only. In practice, not all called methods are replaced by stubs though. The called method \( (y) \) is stubbed only if it is non-trivial, if it conceptually belongs to another “unit,” or if it depends on an environment that is difficult to set up automatically. These criteria cannot be judged automatically and require the knowledge of the developer.

2. A unit test needs test input. Test input can be created either by directly writing bits to memory (similarly to how previously saved objects are loaded into memory through deserialization) or by issuing a series of creation procedure calls that create and modify data through normal means. These two techniques have different advantages: creating the state directly gives complete control of the input creation, while creating the data through creation procedure calls yields data that is more likely to be relevant.

3. The oracle, of a test case can be provided at different levels of abstraction: it can be as concrete as specifying exactly the expected output, it can specify a condition that the output should fulfill for a particular input, it can specify conditions that the output should fulfill for any input, or it can be as abstract as specifying “no exception” as expected output. The more concrete the oracle, the easier it is to express complex scenarios. The more abstract the oracle, the easier it is to re-use for other tests and the less likely it is to introduce an implementation bias.

The following describes the choices on which the AutoTest framework is based:

**Unit.** In accordance with common practice we choose the method as the unit under test. The test synthesizer creates test cases and the goal of each test
6.3. RANDOM-BASED TEST SYNTHESIS ALGORITHM

Case is to test one particular method. AutoTest does not replace any methods by stubs, because it cannot infer automatically what methods to replace and how the stubs should behave.

**Input.** AutoTest creates input objects through regular creation procedure and routine calls, because it is much more likely to end up with valid objects (ones that satisfy their invariant) when creating them through creation procedure calls than by writing arbitrary bits to memory. Hence creation through execution promises to be more efficient. AutoTest uses both new and unmodified objects as well as old and modified ones. If only new objects were used, the code under test would be invoked only with objects in their initial state. For efficiency reasons, when creating new tests, AutoTest reuses objects from previous ones.

**Oracle.** In order to achieve full automation we chose a solely contract-based oracle. A contracted program contains its specification in the form of preconditions, postconditions and invariants. Preconditions are used as filters for invalid input and postconditions are used to detect failures: if a method is called by the test and its precondition is violated, the test is invalid and hence worthless. If the test calls a method and satisfies its precondition, the method body gets executed. If after the execution of the method body the postcondition is violated, the test found a fault in the method.

### 6.3 Random-based test synthesis algorithm

This section presents the default synthesis strategy of AutoTest. It is based on the idea of random testing. Random testing is:

- **Fast.** For a fully automated tester it is important to find as many faults and failures in as little time as possible. The number of generated test cases is secondary. With a random-based strategy time is mostly spent executing the system under test. Synthesis and other bookkeeping tasks are simple and quick executing. Note that for settings where for example manual post processing of test cases is necessary the number of generated test cases might be more important than the time spent generating the tests. For our (fully automated testing) scenario time is of the essence and the random strategy of advantage.

- **Unbiased.** Random testing is unbiased (to the extent that the pseudo random number generator used is unbiased), because it does not try to infer meaning from either code or specification. Lacking a fault or failure model for the
system under test any input is as good a try as any other. This speaks for random testing. Only testing the same routine with the same input twice is obviously useless (assuming that the system under test is deterministic). Random testing with duplicate testing can still perform worse than random testing without, however. If the time to compare a new test case against the already executed ones takes longer than the execution of the test itself, pure random testing is more effective. In AutoTest every test only invokes one routine and most routines do not execute for very long.

The default testing strategy in AutoTest is not a pure random tester. It deviates to increase effectiveness, but tries to not to decrease efficiency. While the deviations are not based on a formal failure model, they have proved to find faults more effectively in a large experiment [28]. The strategy deviates from pure random testing as follows:

- Primitive input values (such as numeric values) are sometimes selected from a set of predefined values (containing maxima, minima and other distinguished values). This is because these special values are likely to uncover a set of often found bugs [28]. AutoTest chooses either a distinguished value or a random one with an adjustable probability.

- Tests use both objects in their initial state and in a state modified through routine calls. For efficiency reasons AutoTest does not create (and modify) new objects for every test. Instead sometimes (given a selectable probability) objects are re-used from earlier tests. These objects were modified as a side effect of prior testing.

In AutoTest, the language used to express test cases is a simplified, dynamically typed variant of Eiffel. Figure 6.2 shows an example of a generated test case. In this example, variables \( v_1, v_2, v_3, \) and \( v_4 \) represent the object pool, which is where objects are stored so that they can be re-used for later testing. The following explains the building blocks of a generated test case. A generated test case is a sequence of instructions; there are four kinds of instructions:

**Object creation.** This instruction consists of a creation procedure name, type name, a sequence of variables and constants serving as arguments and a variable (which will be assigned a reference to the object being created).

An example is Line 1: \texttt{create \{STRING\} v_1.make_empty}. The Eiffel instruction \texttt{create \{class\} object.procedure} creates an object \texttt{object} of class \texttt{class} using the \texttt{procedure} creation procedure. Thus, Line 1 creates an object named \( v_1 \) of type \texttt{STRING} using the \texttt{make} creation procedure with no arguments.
6.3. RANDOM-BASED TEST SYNTHESIS ALGORITHM

The creation procedure can be omitted, in which case an implicit default creation procedure is used.

**Routine invocation.** These instructions are built from a routine, a sequence of variables and constants serving as arguments and a target. Line 67: \( v2.\ deposit (v3) \), for instance, invokes the deposit routine on the \( v2 \) object using \( v_3 \) as an argument.

**Routine invocation with assignment.** This is the same as the normal invocation, as described above, plus an additional target variable being assigned the result of the routine invocation.

**Assignment.** Finally, assignments are built from a source variable or constant and a target variable. Only object pool variables can sever as target variables. In particular, attributes cannot be assigned to. For instance, Line 3: \( v3 := 452719 \) stores an integer into variable \( v3 \).

The above language is as complete as required by the test synthesis strategy and as simple as possible. Control structures, for instance, are not needed as tests are generated on the fly. Likewise, composite expressions can be transformed into a sequence of instructions using only atomic expressions.

Figures 6.5, 6.7 and 6.6 show the random-based synthesis algorithm used in pseudo code.

The following describes how the generation algorithm produces the test case depicted in Listing 6.2.

```
1 create {STRING} v1.make_empty
2 create {BANK_ACCOUNT} v2.make (v1)
3 v3 := 452719
4 v2.deposit (v3)
5 v4 := Void
6 v2.transfer (v4, v3)
```

Listing 6.2: Test case generated by algorithm serving as example.

The example assumes that the AutoTest has been invoked to test all routines of class BANK_ACCOUNT, which is shown in Figure 6.3. At every step, routine write_tests of AutoTest chooses one of the routines which have been tested the least up to that point to be tested. When the testing session starts, no routines have yet been tested, so one of the routines of class BANK_ACCOUNT is chosen at random. Let’s assume routine deposit is chosen. In order to execute this routine one needs an object of type BANK_ACCOUNT and an integer representing the amount of money to deposit.
class BANK_ACCOUNT
create
make (an_owner: STRING) is
    -- Create new bank account object with owner ‘an_owner’.
    require
        an_owner_not_void: a_owner /= Void
    ensure
        owner_set: owner = an_owner
        balance_zero: balance = 0
end
feature (ANY) -- Access
owner: STRING
    -- Name of owner of bank account
balance: INTEGER
    -- Balance of bank account
feature (ANY) -- Money transfer
deposit (v: INTEGER)
    -- Deposit ‘v’ to bank account.
    require
        v_big_enough: v >= 0
    ensure
        balance_increased: balance := old balance + v
end
withdraw (v: INTEGER)
    -- Withdraw ‘v’ from bank account.
    require
        v_big_enough: v >= 0
        v_small_enough: v <= balance
    ensure
        balance_increased: balance := old balance - v
end
transfer (target: BANK_ACCOUNT; amount: INTEGER)
    -- Deposit ‘amount’ from this bank account to ‘target’.
    require
        target_not_void: target /= Void
        amount_big_enough: amount >= 0
        amount_small_enough: amount <= balance
    ensure
        balance_decreased: balance := old balance - amount
        target_balance_increased: target.balance = old target.balance + amount
end
invariant
    balance_big_enough: balance >= 0
end

Listing 6.3: Class interface of BANK_ACCOUNT as used to generated test case depicted in Figure 6.2.
Routine \texttt{write_tests} depicted below contains the main loop of the testing strategy. At each step it selects a routine for testing and writes out code that is ready to be sent to the interpreter for execution.

\begin{verbatim}
write_tests (timeout):
  from
  initialize_pool
  until timeout loop
    m := choose (routines_under_test ())
    write_tests_for_routine (t, m)
end
\end{verbatim}

Routine \texttt{initialize_pool} creates an empty pool of objects to be used for testing (both target and parameters). The routine \texttt{routines_under_test} returns the set of routines under test. The non-deterministic routine \texttt{choose} selects an arbitrary element of a set or a list. The routine \texttt{write_tests_for_routine} is described in Figure 6.7.

Figure 6.5: Testing loop in a nutshell. Algorithm \texttt{write_tests()} first initializes the pool and then tests the class(es) under test routine by routine until a timeout is reached.

The creation and selection of these values happens in routine \texttt{write_test_for_routine}. The test generator chooses randomly from the pool inputs with the required types, but before it makes this choice it might also create new instances for the required types (with probability $P(gen\_new)$) and add them to the pool. In the example test case the generator decides at this point to create a new object of type \texttt{BANKACCOUNT}. Therefore it chooses a creation procedure (\texttt{make}) and now works on the sub-task of acquiring objects serving as parameters for this creation procedure. The creation procedure \texttt{make} requires only one argument which is of type \texttt{STRING}. Hence \texttt{write_creation} calls itself recursively now with the task of creating a string object. 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 \texttt{make_empty} which does not take any arguments (Line 1). The object pool now is: $\{v1 : STRING\}$. The recursive call of \texttt{write_creation} returns. Routine \texttt{write_creation} itself synthesizes the creation instruction for the bank account object (Line 2) using the newly created string object. This updates the object pool to: $\{v1 : STRING, v2 : BANKACCOUNT\}$. At this point \texttt{write_creation} terminates having created a target object, but is invoked once more in order to find an integer (the argument to \texttt{deposit}). Integers are basic objects and in the present example a random integer (452719) is chosen and is assigned to a fresh pool variable (Line 3), changing the object pool to $\{v1 : STRING, v2 : BANKACCOUNT, v3 =
The routine \texttt{write_tests_for_routine} is responsible for writing code that calls a routine \texttt{m}.

Notations: \(<x_1, x_2, x_3>\) creates a list (ordered set) with the elements \texttt{x1}, \texttt{x2}, and \texttt{x3}. Given two lists \((\texttt{list1} \texttt{and list2}), \texttt{list1} \ldots \texttt{list2}\) builds their concatenation. Routine \(P(x)\) non-deterministically evaluates to True with a probability of \(x\). Function \texttt{type (m)} yields the type in which routine \texttt{m} is contained. Function \texttt{choose} picks an arbitrary element from a list. The keyword \texttt{foreach} applies a block of statements to every element of a list. The function \texttt{arguments} gives the list of arguments of a routine. The procedure \texttt{write_invoke_instruction (r, ops)} writes code representing a call to routine \texttt{r} using \texttt{ops} as operands. This code is ready to be sent to the interpreter for execution.

\begin{verbatim}
write_tests_for_routine (r):
  ops := <>
  foreach ot from (<type (r)> ... arguments (r))
    do
      if P (gen_new) then write_creation (ot)
    end
    ops := ops ... choose (conforming_objects (ot))
  write_invoke_instruction (r, ops)
end
\end{verbatim}

For the target object of the routine call and every argument to the routine call, \texttt{write_tests_for_routine} does the following:

1. With a probability of \(P(gen\_new)\) it creates a new object and puts it into the pool.

2. It selects an arbitrary object conforming to the required type from the pool.

Using the selected objects, \texttt{write_tests_for_routine} then writes out code that invokes routine \texttt{r}. The first element of list \texttt{ops} is used as the target object of the routine call, subsequent elements as arguments. Note that object creation and selection are decoupled. As a result, routines are sometimes invoked with newly created objects and sometimes with existing and probably modified ones.

Figure 6.6: Routine \texttt{write_test_for_routine} which generates calls to invoke routines under test.
The routine \texttt{write_creation} \((t)\) prints code that creates a object of type \(t\).
Notations: \(<x_1, x_2, x_3>\) creates a list (ordered set) with the elements \(x_1, x_2,\) and \(x_3\).
Given two lists \((\text{list1 and list2}), \text{list1 ... list2}\) builds their concatenation.
Routine \(P(x)\) non-deterministically evaluates to True with a probability of \(x\). Function \texttt{choose} picks an arbitrary element from a list. The keyword \texttt{foreach} applies a block of statements to every element of a list. The function \texttt{creators} returns the list of creation procedures of a type. The function \texttt{arguments} gives the list of arguments of a creation procedure. The function \texttt{write_creation_instruction} prints the code that creates an object of type \(t\) using the creation procedure \(c\) and the arguments \(ops\). The created object is assigned to a new variable from the object pool. The generated code is ready to be sent to the interpreter for execution. Similarly, \texttt{write_assignment_instruction} \((x)\) prints the code that assigns \(x\) to a new variable in the object pool.

\begin{verbatim}
write_creation (t):
if is_basic_type (t) then
    if P (gen_basic_rand) then
        write_assignment_instruction (random_basic_object (t))
    else
        write_assignment_instruction (choose (predefined_objects (t)))
    end
else
    c := choose (creators (t))
    ops := <>
    foreach ot from (arguments (c)) do
        if P(gen_new) then
            write_creation (ot)
        end
        ops := ops ... choose (conforming_objects (ot))
    end
    write_creation_instruction (c, ops)
end
\end{verbatim}

The routine \texttt{write_creation} \((t)\) treats basic types (e.g. \texttt{INTEGER, DOUBLE, BOOLEAN, ...}) and reference types differently. For basic types with a probability of \(P(\text{gen_basic_rand})\) an arbitrary random object of the required type is created using routine \texttt{random_basic_object}. Since objects of all basic types are finite state such a routine is easy to devise using a random number generator. The alternative is to select an object from a predefined list that includes common values (e.g. \(0, 1\) for integers) and boundary values.

If the type to be created is not basic then the algorithm chooses randomly one of the available creation procedures for the required type. This creation procedure might also need arguments, so a similar algorithm to the one implemented in \texttt{write_test_for_routine} is applied, with the difference that a creation procedure call does not need a target, only arguments.

Figure 6.7: Routines \texttt{write_creation} which generates calls to create objects.
452719}. Execution returns to write_test_for_routine which is now able to synthesize what it was asked for: a call to BANK_ACCOUNT.deposit. It uses the newly created bank account and the randomly chosen integer (Line 4).

At this point execution returns to write_tests which selects another routine for testing; let’s assume BANK_ACCOUNT.transfer is chosen. This routine transfers the amount it is given from the bank account on which it is called to the bank account it receives as argument. Again, a target object and 2 arguments are necessary. For each of these 3 inputs, an object of the corresponding type might be created and added to the pool (with the probability $P(\text{gen\_new})$). In the case of the call to BANK_ACCOUNT.transfer the decision is to create a new 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 $\{v_1 : \text{STRING}, v_2 : \text{BANK\_ACCOUNT}, v_3 = 452719, v_4 = \text{Void}\}$. 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. Test case generation would continue after this point, but for brevity the example stops here.

### 6.4 Object pool

As explained above the random-based algorithm in AutoTest uses an object pool. Any object either created as object under test, serving as input for a test or that has been returned from a routine under test is put in the pool. The primary reason for the pool is to enable testing with objects of different age: i.e. objects that are fresh and unmodified and objects that have been used for many modifications.

Besides the diversification aspect the object pool has additional benefits:

- Object that are of particular interest to a certain SUT can be added to the test input data without any additional mechanism. Adding a factory that returns the objects of interest as a class under test is sufficient. When the class is tested its return values will be put in the object pool and used for subsequent tests. To increase the frequency with which those values should be used one can increase the test priority of the factory class.

- Similarly, objects created by the system under test itself get to be used as test inputs as well. Objects created by the system under test might be of special interest, as they are likely to be realistic.

- The object pool enables integration testing. When an object is needed either as input for a test, one object from all conforming objects of the pool is randomly selected. Note that an object does not have to be an instance of a
6.4. OBJECT POOL

Figure 6.8: Class diagram of class \textit{SET} and descendants. AutoTest revealed a bug that can only be found when using both objects of type \textit{LINKED\_SET} and \textit{ARRAYED\_SET} together.

given type to conform to it. It can be an instance of a descendant type or a generic derivation.

The following example illustrates how the object pool enables integration testing. The test case described is real and was found by AutoTest when trying to test the mathematical modeling library (MML) [90]. The MML library is a step towards bridging the gap between theorem provers and Eiffel assertions. It is a library of side-effect free mathematical models. Each class has a correspondent in the theorem prover. For brevity the example is simplified.

Listing 6.8 depicts three classes from the MML library: \texttt{SET, ARRAYED\_SET, LINKED\_SET}. Note that both \texttt{subset} and \texttt{superset} are deferred in class \texttt{SET} (Listing 6.8). Also, the postconditions of \texttt{subset} mentions \texttt{superset} and vice-versa. Recursive assertions are allowed in Eiffel. To break it, the run-time assertion monitor does not evaluate the assertions of an assertion. For example, the evaluation of the postcondition of \texttt{superset}, triggers the execution of \texttt{subset}. Since the execution of \texttt{subset} is due to an assertion evaluation, the contracts of \texttt{subset} are not evaluated themselves.

The two descendants of class \texttt{SET} each make a different choice in how to implement \texttt{superset} and \texttt{subset}:

- Class \texttt{ARRAYED\_SET} implements \texttt{superset} only. Routine \texttt{subset} is defined in terms of \texttt{superset}. (See Listing 6.5)

- Class \texttt{LINKED\_SET} implements \texttt{subset} only. Routine \texttt{superset} is defined in terms of \texttt{subset}. (See Listing 6.6)

The generated test case that revealed a bug in this library can be seen in Listing 6.7. Invoking \texttt{subset} on a \texttt{LINKED\_SET} using an \texttt{ARRAYED\_SET} as argument does not terminate. \texttt{subset} invokes \texttt{superset}. A similar bug is found when invoking routine \texttt{superset} if target object and argument are switched.
Listing 6.4: Source code of class SET. Ancestor to class ARRAYED_SET and class LINKED_SET. The class defines two routines each of which are deferred and have to be implemented by its subclasses constraint only by the contracts.

Listing 6.5: Source code of class ARRAYED_SET. It inherits from the deferred class SET. The routine subset is implemented with the help of superset.
deferred class LINKED_SET
  inherit
  SET
  feature
    subset (o: SET): BOOLEAN
      do
        ... implementation omitted ...
      end

    superset (o: SET): BOOLEAN
      do
        Result := o.subset (Current)
      end
  end

Listing 6.6: Source code of class LINKED_SET. It inherits from the deferred class SET. The routine superset is implemented with the help of subset.

create {LINKED_SET} s1
create {ARRAYED_SET} s2
s1.subset (s2)

Listing 6.7: Generated test case that reveals bug by using both LINKED_SET and ARRAYED_SET together.
Note that implementing one routine in term of another is perfectly legitimate and in this case it appears like a reasonable choice of code reuse. There is only a problem if both classes are used together.

While AutoTest primarily creates unit level tests, this particular test has properties of integration level tests: it combines two components that are not tightly coupled.

Why did AutoTest choose to combine those two components? It does not rely on a special tactic to cover certain combinations of classes or dynamic binding. Instead the combination is a consequence of using an object pool. When testing `ARRAYED_SET.subset`, AutoTest needed an object conforming to type `SET` to serve as an argument. It looked at the object pool and found an object of type `LINKED_SET`, which was there because it was used for testing routines of that type before.

### 6.5 Evaluation of random contract-based testing

As part of her thesis [30], Ilinca Ciupa provides a thorough evaluation of random-based testing using contracts. She uses a variety of algorithms that are similar to the one presented here. The variations are all implemented as strategies in AutoTest. The following is a summary of her main findings:

- 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.
6.6 Minimizing tests

Randomized unit test cases can be very effective in detecting defects. In practice, however, failing test cases often comprise long sequences of routine calls that are tiresome to reproduce and debug. We present a combination of static slicing and delta debugging that automatically minimizes the sequence of failure-inducing routine calls. In a case study on the EiffelBase library, the strategy minimizes failing unit test cases on average by 96%.

This approach improves on the state of the art by being far more efficient: in contrast to the approach of Lei and Andrews [63], who use delta debugging alone, our case study found slicing to be $50 \times$ faster, while providing comparable results. The combination of slicing and delta debugging gives the best results and is $11 \times$ faster.

Automated testing solutions are more attractive than ever. While automated execution of test cases has been around for a long time (and recently become popular in practice with automated unit testing tools), automated generation of test cases is now also finding its way into practice. Commercial tools such as Agitar’s Agitator [18] or ParaSoft’s JTest [1] are clear examples of this trend.

Unfortunately, randomly generated unit test cases can become very large in size of inputs and the number of routine invocations. While large unit test cases can be useful for triggering specific defects, they make the subsequent diagnosis hard, and take resources to execute. This can be a serious problem in light of the principle that suggests keeping failing test cases into the regression test suite.

The problem of minimizing failing test cases was first generally addressed by Zeller and Hildebrandt [107], who introduced delta debugging to minimize failure-inducing inputs, automatically repeating failing test cases with reduced inputs until the input could no longer be reduced. In 2005, Lei and Andrews [63] applied delta debugging on randomized unit tests (i.e. randomly generated sequences of routine calls). Their experiments show that delta debugging significantly reduces the number of failure-inducing calls (reporting an average reduction by 75%)—and thus end up in a smaller unit test case.

The downside of delta debugging, though, is that it requires running the test repeatedly to assess the outcome of reductions. The time complexity of the algorithm is $O(n^2)$, where $n$ is the size of the input—or, as in the approach of Lei and Andrews, the number of routine calls. For large numbers of failing tests, or for tests that consume significant resources, this cost is prohibitive. We introduce an alternative test case minimization routine based on static program slicing.

The key idea of using slicing for minimization is straightforward: we start from the oracle in the failing test case—that is, the failing assertion or another exception. By following back data dependencies, we establish the subset of the code—the slice—that possibly could have influenced the oracle outcome; any
instructions that are not in the slice can therefore be removed. An additional delta debugging run takes minimization further, as summarized in Figure 6.9.

6.7 Minimizing through slicing

Listing 6.8 shows a test generated by AutoTest. It tests the interplay of various classes in the EiffelBase library—the library, used by almost all Eiffel applications, covering fundamental data structures and algorithms. This test case has successfully detected a defect in the EiffelBase library. The invocation in Line 149: \texttt{v\_136 := v\_135.real\_item} results in a precondition being violated (of the \texttt{RANDOM.i\_th} routine), raising an exception.

How did this violation come to be? In general, a violated precondition indicates a defect in the caller or further upstream in the execution. In Listing 6.8, though, “upstream” means 148 lines of code, not counting all the code that is executed in routine calls. This is a common problem of randomly generated test cases: the generated test cases can be very large. This makes it hard to understand how the failure came to be, which in turn takes time in debugging. It also slows

![Diagram](image)

Figure 6.9: Unit test generation and minimization. An Eiffel class file (a) is fed into the AutoTest tool (b), which generates a randomized unit test case (c) as a sequence of routine calls. Failing unit test cases (d) are automatically minimized—first by static analysis (e), and then by delta debugging (f).
Listing 6.8: A generated random test case. Line 149 fails with a nested precondition violation.

down the execution of the test case, which may be acceptable during debugging, but brings a recurrent penalty as soon as the test is included in the regression test suite.

A means to simplify failing test cases to the bare minimum of instructions required to reproduce the failure is therefore needed. This helps the programmer understand the failure conditions and narrow down the defect—and it makes test execution more efficient. A simple test case will also be easier to explain and communicate. Finally, simplifying promises for easy identification of duplicate failures (multiple failing test cases that relate to the same defect), as the simplified test cases all contain the same sequence of instructions.

A common diagnosis strategy for debugging works as follows: starting with the failing instruction, proceed backwards (“upstream”) to identify those instructions that might have influenced the failure. This can be done dynamically (e.g. in a debugger) or statically (by analyzing the code). In the case of Listing 6.8, this backward reasoning could be as follows:

1. Line 149 fails when invoking \texttt{v\_135.real\_item}.

2. Object \texttt{v\_135} was created in Line 148 as a \texttt{RANDOM} object, setting the seed from \texttt{v\_63}.

3. Object \texttt{v\_63}, in turn, was created from \texttt{v\_62} in Line 69.

4. Object \texttt{v\_62} was created from a constant as a \texttt{PRIMES} object in Line 68.
Not only does this backwards reasoning help us understand the chain of events that led to the failure, but even more important is the fact that no other objects were involved in the computation. In other words, the above backward sequence of events could be reversed to a forward subset of instructions, as shown in Listing 6.9.

```
create {PRIMES} v_62
v_63 := v_62.lower_prime {{INTEGER_32} 2)
create {RANDOM} v_135.set_seed (v_63)
v_136 := v_135.real_item
```

Listing 6.9: Example test case, minimized

The reason for the failure is a fault in the creation procedure \texttt{RANDOM.set_seed}. It is not properly initializing the pseudo random number generator. Accessing the first random number via \texttt{v_135.real_item} signals the fault through a failure. The other \texttt{RANDOM} creation procedure, far more frequently used, works fine.

Do these four out of 149 lines suffice to reproduce the failure? Since, as part of random unit test generation, we already have a full-fledged infrastructure for testing, we can simply try. Indeed, Listing 6.9 produces the same failure as Listing 6.8: the same exception is raised from the same object. We thus have successfully minimized the original test case from 149 to only 4 lines.

The general technique of reasoning backwards and identifying those statements that could have influenced a statement at hand is known as slicing, and the set of influencing statements is known as a slice [102]. In our example, the statements in Listing 6.9 constitute the backward slice of Line 149. Put in other words, Line 149 depends on the statements in the backward slice, because they all potentially influence the input to Line 149. Furthermore, the slice is a static slice, as it is obtained from program code alone—in contrast to a dynamic slice [4, 62], which would be extracted from a program run, tracking all concrete variable accesses. Note that slicing as sketched above, despite being static is nevertheless also similar to dynamic slicing, because it does not consider control flow. The test case itself does not contain any control flow statements, and the source of the system under test (which does) is not considered.

Relying on static program analysis to compute a slice is hard, both in terms of implementation effort and efficiency. The reason is that there are many ways in which objects can influence each other. A full-fledged static analysis therefore should be

- \textit{interprocedural}: take into account what happens in a function or routine call;
6.7. MINIMIZING THROUGH SLICING

- rely on *points-to analysis*: track where references might point to; and
- *flow-sensitive*: discriminate between multiple paths of execution.

All these techniques make static analysis slow and hard, especially when applied to large industrial programs.

Dynamic slicing, which works on program runs also has a set of problems. It requires the program to be instrumented so that the exact path through the program can be recorded. Such instrumentation makes the program to be sliced execute more slowly and requires the complete source of the program. Some part of the code may not always be available. Third party libraries are often distributed in binary form only. Programs may call out to libraries written in a different language through a foreign function interface, which makes instrumentation much more difficult. A program may make calls to the operating system or to the run time system, both of which are often only available in binary form. When the full source code is not available, dynamic slicing becomes less precise. Slices may exclude relevant statements and/or include irrelevant ones.

An imprecise analysis may be acceptable if there is a separate way to verify its result. This is exactly the case in our application, since we can always verify the analysis result by a repeated test run (which may actually be faster than expensive additional analysis).

These ideas are the basis of our approach to minimize test cases: a straightforward static analysis that follows data dependencies backwards from the failing routine call, and returns a slice within the generated test case; this slice then serves as a minimized test case as well. Our approach is not interprocedural, and does not implement fancy static analysis techniques. Since it is not interprocedural and test cases do not contain control-flow the algorithm becomes very fast and simple. While being static and not needing instrumentation, it does retain some of the characteristics of dynamic slicing. The technique is not precise, however. Slices may fail to include relevant statements may include irrelevant ones; but, as said before, we always verify the result using an additional test run.

This following presents the slicing-based minimization algorithm in detail. The main function \( \text{ssmin} \) takes a test case as a sequence of instructions and returns a minimized test case, keeping only those instructions in the test case that might affect the failing instruction. As mentioned, the \( \text{ssmin} \) algorithm does not guarantee to produce the smallest possible test case.

Let \( tc \) be a test case, represented as a sequence of instructions \( tc = [i_1, \ldots, i_n] \). Our *static slicing minimization algorithm* \( \text{ssmin}(tc) \) returns a minimized sequence \( tc' = \text{ssmin}(tc) = [i'_1, \ldots, i'_m] \) with \( m < n \) and \( i'_1, \ldots, i'_m \in tc \).

\( \text{ssmin} \) is defined from the function \( \text{slice} \). \( \text{slice} \) returns all instructions in the original test case on which the last instruction \( i_n \) depends. In other words, \( \text{slice} \)
returns only those instructions that can actually affect the execution of the last instruction, the one that makes the test fail:

$$\text{slice}([i_1, \ldots, i_n]) \triangleq \{ i \in [i_1, \ldots, i_n] \mid i_n \rightarrow^* i \}$$

In this definition, \([a \in A \mid b]\) is the subsequence of \(A\) including only those elements for which \(b\) holds, and \(i_n \rightarrow^* i\) is the transitive reflexive closure over the dependency relationship. An instruction \(i_2\) is dependent on an instruction \(i_1\), written \(i_2 \rightarrow i_1\), if there is some variable \(v\) that is written by \(i_1\), read by \(i_2\), and not written in-between:

$$i_2 \rightarrow i_1 \triangleq \exists v \in (\text{WRITE}(i_1) \cap \text{READ}(i_2)) \setminus \text{WRITTEN\_BETWEEN}(i_1, i_2)$$

(Note that we ignore control dependencies, as the generated test cases do not contain any control instructions.)

The set of variables read and written depend on the individual type of instruction:

$$\text{READ}(i) \triangleq \begin{cases} 
\{ \text{source}(i) \} & \text{if } i \text{ is an assignment} \\
\{ \text{arguments}(i) \} & \text{if } i \text{ is an object creation} \\
\{ \text{target}(i) \} \cup \text{arguments}(i) & \text{if } i \text{ is a routine invocation} \\
\{ \text{target}(i) \} \cup \text{arguments}(i) & \text{if } i \text{ is a routine invocation with assignment}
\end{cases}$$

$$\text{WRITE}(i) \triangleq \begin{cases} 
\{ \text{receiver}(i) \} & \text{if } i \text{ is an assignment} \\
\{ \text{target}(i) \} & \text{if } i \text{ is an object creation} \\
\{ \text{target}(i) \} & \text{if } i \text{ is a routine invocation} \\
\{ \text{receiver}(i), \text{target}(i) \} & \text{if } i \text{ is a routine invocation with assignment}
\end{cases}$$

In these definitions, \(\text{source}(i)\) is the right-hand side of instruction \(i\), \(\text{receiver}(i)\) is the left-hand side of an assignment, \(\text{target}(i)\) the variable on which a routine is called on, and \(\text{arguments}(i)\) is the set of actual variables (constants are ignored).

Finally, the set of variables that may be written by some other instruction executed between \(i_1\) and \(i_2\) is defined as

$$\text{WRITTEN\_BETWEEN}(i_1, i_2) \triangleq \{ v \mid i \in i_1 \leftrightarrow i_2 \land v \in \text{WRITE}(i) \}$$

Here, \(i_1 \leftrightarrow i_2\) denotes the sequence of instructions that can be executed between \(i_1\) and \(i_2\), but excluding \(i_1\) and \(i_2\).

Finally, we define \(ssmin\) on top of \(\text{slice}\). Since \(\text{slice}\) is unsound, we check whether the minimized test case returned by \(\text{slice}\) produces the same failure as the
original:

\[
ssmin(tc) \triangleq \begin{cases} 
slice(tc) & \text{if } failure(tc) = failure(slice(tc)) \\
tc & \text{otherwise}
\end{cases}
\]

where \( failure \) returns the contract that failed the test.

Example

The core of the algorithm is the definition of the \( ssmin \) function, which gets a sequence of instructions to be minimized. This is how \( ssmin \) works, applied to the 149 instructions in Listing 6.8:

1. \( ssmin \) first invokes \( slice \), which establishes backward dependencies, starting with the failing instruction \( i_{149} \) and unfolding dependencies until a fix-point is reached.

2. In our case, \( slice \) finds \( i_{148} \) as an instruction on which \( i_{149} \) depends (written \( i_{149} \rightarrow i_{148} \)). \( i_{149} \) depends on \( i_{148} \) because it creates an object (\( v_{135} \)) that is read by \( i_{149} \).
   (Formally, \( v_{135} \in WRITE(i_{148}) \land v_{135} \in READ(i_{149}).\)
   Furthermore, \( v_{135} \) is not in the set of variables written between the two instructions; actually, this set is empty.
   (Formally, \( v_{135} \notin WRITTEN\_BETWEEN(i_{148},i_{149}) = \emptyset.\))

3. \( i_{148} \) depends on \( i_{69} \) because \( v_{63} \in WRITE(i_{63}) \land v_{63} \in READ(i_{148}) \).
   Again, \( v_{63} \) is not written between \( i_{69} \) and \( i_{148} \).

4. Likewise, \( i_{69} \) depends on \( i_{68} \) because of variable \( v_{62} \) and
   \( WRITTEN\_BETWEEN(i_{68},i_{69}) = \emptyset.\)

5. Instruction \( i_{68} \) does not depend on anything else. Therefore, the slice eventually contains the four instructions as listed above;
   formally, \( slice([i_1,\ldots,i_{149}]) = [i_{68},i_{69},i_{148},i_{149}] \), resulting in the minimized test case in Listing 6.9.

6. Finally, \( ssmin \) validates the \( slice \) result: since Listing 6.9 and Listing 6.8 produce the same failure, the result is correct.

The slicing approach to simplify test cases is surprisingly simple but highly effective. In our test setting, executing the full test case (Listing 6.8) took 2344 ms. Executing the minimized test case takes only 105 ms—that is, the test case is now 22 times faster. The whole run for \( ssmin \) took 105 ms as well, due to the verifying
test run at the end. The slicing step itself (slice) took less than 1 ms, which means virtually no cost.

The complexity of $ssmin$ is linear over the number of instructions in the test case (i.e. $O(n)$ for $n$ instructions to be examined) and constant in the number of test executions (i.e. $O(1)$ because one test run is required). The question remains open whether the unsound approach works in practice, and how efficient and effective it actually is when applied on a large number of programs. This is explored in the following section.

### 6.8 Case study: EiffelBase library

#### Setup

In order to evaluate the efficiency and effectiveness of the $ssmin$ algorithm we have generated over 1300 failing test cases for the EiffelBase library with AutoTest. EiffelBase is the standard library for the EiffelStudio compiler and IDE containing basic classes (such as string and integer), data structures (such as linked lists and trees), basic I/O classes (such as file and console access), and a variety of utility classes (such as random number generation). The exact setup used to conduct the experiment was the following:

**Computer:** AMD dual core, 1 GHz, 2 GB RAM

**Operating System:** Linux (Ubuntu 7.04)

**Eiffel compiler:** 6.0.6.8510

**EiffelBase:** 6.0.6.8510

**AutoTest (and Erl-G):** Subversion revision 656

**Minimization algorithms:** included in AutoTest revision 656

**AutoTest testing options:** ise.ace -just-test -disable-minimize -disable-manual

**AutoTest minimization options:** ise.ace -benchmark -just-test -time-out=0 -disable-manual -disable-auto

**Scope of EiffelBase test:** all classes in EiffelBase

---

2Several patches to fix bugs were applied, the precise instance of EiffelStudio used is available from [http://se.ethz.ch/people/leitner/ase_min/](http://se.ethz.ch/people/leitner/ase_min/)
Scope of data structure test: **LINKED_LIST, HEAP_PRIORITY_QUEUE, HASH_TABLE, and BINARY_SEARCH_TREE**

We ran tests both on EiffelBase as a whole and on subsets of it. The reason for this was the intuition that the scope of the tests can make a difference in the degree of connectivity between the test calls generated by AutoTest: it may be that the wider the scopes, the better bug-reproducing examples can be reduced. We also varied the duration of the testing process in the experiment, as the size of the bug-reproducing examples is likely to increase with the duration of the test run. Hence we ran AutoTest twice using different scopes and time-outs to produce two separate sets of failing test cases:

- The first run tested all of the EiffelBase library for 15 minutes. This included interactive as well as I/O-dependent classes.
- The second run tested four data structure classes for 5 minutes. This did not test interactive or I/O dependent classes.

6.8.1 Results

Table 6.1 summarizes the results of the case study in terms of test case size. The number of instructions per test case is averaged over all processed test cases. There was no test case that slicing failed to minimize or where slicing generated a test case that did not reproduce the same failure as the failure produced by the original test case. This result indicates that even though the proposed slicing algorithm is unsound it works well in practice.

<table>
<thead>
<tr>
<th>Test scope</th>
<th>Avg. no. original inst.</th>
<th>Avg. no. min. inst. (slicing)</th>
<th>Avg. reduction factor (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EiffelBase</td>
<td>35.29</td>
<td>1.62</td>
<td>95.39</td>
</tr>
<tr>
<td>Data struct.</td>
<td>100.24</td>
<td>2.90</td>
<td>97.10</td>
</tr>
</tbody>
</table>

Table 6.1: Reduction in number of lines of code of test cases by slicing. Slicing reduces the size of test cases by over 90%.

Slicing produced test cases reduced by 95% (EiffelBase) and 97% (data structures). This clearly indicates that slicing is a very effective minimization technique despite its simplicity. Test cases were minimized on average from 35.29 down to 1.62 instructions and from 100.24 down to 2.9 instructions. While developers might be lost in test cases spanning several pages, bug reproducing test cases of 1 to 3 lines should be easy to understand and likely to give hints towards how further debugging should take place.
The overall overhead of minimization compared to random testing (as summarized in Table 6.2) is small: the first run of the random tester lasted for 900 seconds, subsequent minimization only 9.4 seconds; the second run lasted for 300 seconds and minimization took 1.9 seconds.

Table 6.3 shows how much time is spent slicing and how much time executing the resulting slice. Values are again averaged over all processed test cases. Most of the minimization time is spent verifying the slice (via execution). The slicing step in particular, takes negligible time; this seems to confirm the advantage of a shallow but fast slicing. Note that the sum of slicing and execution time is slightly smaller than the total minimization time, because the algorithm also performs some extra bookkeeping and benchmarking.

Even though slicing has to work on the original, large test cases, the time in this phase is negligible (0.13 ms resp. 0.18 ms). The overhead to execute the (minimized) test cases takes up most of the total execution time. The high cost of execution is due to two reasons:

**Sandboxing.** As explained in Chapter 7, for random testing to be practical, some form of sandboxing of the system under test is essential. AutoTest guards itself from severe malfunctions in the system under test through a process-based sandbox. To verify that they reproduce the original failure, slices are executed in a new interpreter process to be immune to side effects from previous tests. Process start-up and the interpreter initialization (even though implemented in a lazy fashion) costs time.

**Oracle checks.** The oracle used by AutoTest is based on run-time assertion monitoring. Unlike traditional unit tests where the oracle takes the form of assert statements at the level of the test case, the oracle or rather the contracts are interwoven throughout all of the source code. As a result there are oracle checks not only at the end of the test case execution, but also in during the execution of the various routines executed directly or indirectly by the test case. These additional checks provide for more error detection capabilities, but slow down execution.
Table 6.3: Performance of different minimization sub-tasks. *Avg. min. time* is the average time it took to minimize a test case. *Avg. slicing time* is the average time it took to slice a test case. *Avg. exec. time* is the average time it took to execute the sliced test case. *Avg. orig. exec.* is the average time it took to execute the original test case. Note that the minimization time is not precisely equal to the sum of the slicing time and the execution time. The difference is due to some extra bookkeeping work done during minimization.

<table>
<thead>
<tr>
<th>Test scope</th>
<th>Avg. min. time (ms)</th>
<th>Avg. slicing time (ms)</th>
<th>Avg. exec. time (ms)</th>
<th>Avg. orig. exec. (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EiffelBase</td>
<td>71.40</td>
<td>0.13</td>
<td>71.25</td>
<td>898.37</td>
</tr>
<tr>
<td>Data struct.</td>
<td>110.42</td>
<td>0.18</td>
<td>110.18</td>
<td>5930.41</td>
</tr>
</tbody>
</table>

The last column of Table 6.3 gives the time to execute the unminimized test case. Note that this is not the time it took originally to generate and execute the test case:

- Test generation has the added overhead of the synthesizer (although the random synthesizer does not add much).
- Test generation is incremental, test replay is not.
- Test replay does not need to execute those tests that contain no failure.

Note that the time it takes to execute the original test case is much bigger than the time it takes to slice the test case and then execute the slice. This means that for regression testing minimization already pays off the first time.

### 6.8.2 Non-determinism

Several test cases, generated via random testing, proved to be non-deterministic: running the test case on the same system under test did not repeatedly produce the pass or fail result. We found two reasons for this behavior:

**Garbage collection.** Finalization routines can be called by the garbage collector at different times. Availability of memory influences the behavior of garbage collection most, but depending on the type of garbage collector used there can be other factors too. To remove this kind of non-determinism the garbage collector can be turned off — assuming that there is enough memory to do so. All finalization routines can be called in a predetermined...
fashion at program exit. Test cases that produce a failure only when finaliza-
tion routines are called during program execution still cannot be minimized 
this way.

**File and directory accesses.** Random testing and minimization involved routine 
calls responsible for creating, deleting, writing and reading files and direc-
tories. Tests that depend on the existence or non-existence of certain files 
will sometimes fail or pass depending on whether random testing previously 
created the corresponding files.

Failing test cases are expected to prove the presence of bugs. If they do not 
prove this deterministically, they are therefore of little use. To avoid skewing test results, we removed such test cases (16 for EiffelBase, 3 for the data struc-
ture tests) by hand. Further work would be necessary to either remove such test cases automatically; for instance, using repeated executions, taint analysis, or by 
removing the non-determinism using a capture-replay mechanism [82, 83].

### 6.8.3 Discussion

Test minimization provides two benefits: tests become easier to understand and regression testing takes less time. The results of applying `ssmin` on the EiffelBase library indicate its practicability. It pays to minimize a test even when re-executing it only once. Despite the theoretical possibility of test cases that could not be minimized at all (due to the unsoundness of slicing), the data suggests that this is not a problem in practice. The extra time spent minimizing failure reproducing test cases is very small compared to the time spent finding those test cases. The gain of having to understand a test case consisting of 1 to 3 lines instead of one consisting of several dozens of instructions seems compelling.

### 6.9 Minimizing through delta debugging

Our work is not the first one that minimizes randomized unit tests. In 2005, Lei and Andrews [63] presented an approach based on *delta debugging*. The delta de-
bugging minimization algorithm (abbreviated as *ddmin*) takes a set of factors that might influence a test outcome, and repeats the test with subsets of these factors. By keeping only those factors that are relevant for the test outcome, it systemat-
ically reduces the set of factors until a minimized set is obtained containing only relevant factors [107].

The unique feature of delta debugging is that it is purely *experimental*, mean-
ing that it gathers its knowledge solely by running experiments. In particular, it requires no knowledge about the dependencies or other semantics or constraints
on the factors—only an automated test and a means to split the set of factors into subsets.

### 6.9.1 Algorithm

The main function of the delta debugging algorithm is \( \text{ddmin} \). It takes a test case as a sequence of instructions and minimizes the test case using repeated test runs of selected subsets.

Let \( tc \) be a test case, represented as a sequence of instructions \( tc = [x_1, \ldots, x_n] \). Like static slicing minimization, delta debugging minimization returns a minimized sequence \( tc' = \text{ddmin}(tc) = [x'_1, \ldots, x'_m] \) with \( m < n \) and \( x'_1, \ldots, x'_m \in tc \).

The algorithm \( \text{ddmin} \) relies on a testing function \( \text{test}([x_c, \ldots x_d]) \) which returns \( \times \) if the given subsequence fails, and \( \checkmark \) if it passes; \( \text{test}(tc) = \times \) and \( \text{test}([]) = \checkmark \) is assumed. The algorithm uses repeated tests to reduce the subsequence to the minimum, starting with splitting the test case in two halves, and continuing with a larger number of smaller subsets:

\[
\text{ddmin}(tc) \triangleq \text{ddmin}_2(tc; 2) \quad \text{where}
\]

\[
\text{ddmin}_2(tc, q) \triangleq \begin{cases} 
\text{ddmin}_2(tc^q_i, 2) & \text{if } \exists i \in \{1, \ldots, q\} \cdot \text{test}(tc^q_i) = \times \text{ ("reduce to subset")}
\text{ddmin}_2(tc \setminus tc^q_i, \max(q - 1, 2)) & \text{else if } \exists i \in \{1, \ldots, q\} \cdot \text{test}(tc \setminus tc^q_i) = \times \text{ ("reduce to complement")}
\text{ddmin}_2(tc, \min(|tc|, 2q)) & \text{else if } q < |tc| \text{ ("increase granularity")}
tc & \text{otherwise ("done")}
\end{cases}
\]

where \( tc^q_d \) is its \( d \)-th out of \( q \) subsequence of approximately equal size (with \( d \leq q \)). The concatenation of the subsequences should yield the original sequence \( (tc = tc^q_1 \circ \cdots \circ tc^q_q) \). For example, \( tc^q_d \) can be defined as \( [i_{[\frac{q}{2} n] + 1}, \ldots, i_{\frac{q}{2} n}] \).

Also, \( tc \setminus tc^q_d \) is the subsequence of \( tc \) without \( tc^q_d \). I.e., \( tc \setminus tc^q_d = tc^q_1 \circ tc^q_2 \circ \cdots \circ tc^q_{d-1} \circ tc^q_{d+1} \circ \cdots \circ tc^q_q \). The number of instructions in a sequence is denoted via \( |tc| \).
6.9.2 Example

Let us illustrate how \textit{ddmin} works by applying it to minimize our running test case in Listing 6.8, following the formal definition of the \textit{ddmin} algorithm given above:

1. \textit{ddmin} splits the entire test case \textit{tc} into two subsequences \textit{tc}_1 = [i_1, i_{75}] and \textit{tc}_2 = [i_{76}, i_{149}] (we write \textit{tc} = \textit{tc}_1 \circ \textit{tc}_2, where \circ stands for the concatenation of these sequences).

2. Testing \textit{tc}_1 passes the test (test(\textit{tc}_1) = \checkmark).

3. Since \textit{tc}_2 accesses variables that are initialized in \textit{tc}_1, strictly spoken, \textit{tc}_2 is not a legal program. Our \textit{test} function is set up to catch such illegal accesses and ignore the respective instructions, but to proceed with the legal ones. No failure is reported (test(\textit{tc}_2) = \checkmark).

4. Since neither test fails, \textit{ddmin} increases the granularity by doubling the number of subsets and splitting \textit{tc} into \textit{tc} = \textit{tc}_1 \circ \textit{tc}_2 \circ \textit{tc}_3 \circ \textit{tc}_4.

5. In the next round, all subsets pass (with \textit{tc}_2, \textit{tc}_3, and \textit{tc}_4 again reporting illegal accesses).

6. However, \textit{ddmin} also tests complements: that is, it removes the appropriate subsequences from the original. Removing \textit{tc}_1 results in a test run test(\textit{tc} \setminus \textit{tc}_1) = test(\textit{tc}_2 \circ \textit{tc}_3 \circ \textit{tc}_4) = test([i_{37}, \ldots, i_{149}]) = \times. This test fails, since all the statements required for the failure were actually executed. Therefore, \textit{ddmin} can remove \textit{tc}_1 from the working set.

7. The test also fails for test(\textit{tc} \setminus \textit{tc}_3) when \textit{tc}_3 = [i_{76}, i_{113}] is removed. The remaining working set is now \textit{tc} = \textit{tc}_2 \circ \textit{tc}_4, and the number of subsets is again two (n = 2).

8. In subsequent steps, \textit{ddmin} keeps on splitting and removing, until after an overall of 52 tests, \textit{tc} is minimized to \textit{tc} = [i_{68}, i_{69}, i_{148}, i_{149}]. Neither instruction can be removed without causing the test to fail; thus, \textit{ddmin} is done.

The result of \textit{ddmin} thus is the same as \textit{ssmin}, discussed in Section 6.7. But \textit{ddmin} takes vastly more time. The overall minimization with \textit{ddmin} takes 22300 ms, which is 212 times slower than the minimization with \textit{ssmin} (105 ms). This does not come as a surprise, considering that \textit{ssmin}’s time essentially comes from the single test execution.
6.10. A JOINT APPROACH

6.9.3 Delta debugging compared

The experiences of \textit{ddmin} running on our example easily generalize to larger systems. To compare against the state of the art, we have repeated our case study (Section 6.8) with \textit{ddmin} instead of \textit{ssmin} on the EiffelBase library.

The overall overhead of minimization with \textit{ddmin} compared to random testing (as summarized in Table 6.4) is quite high: for the first run of the random test generator, which lasted for 15 minutes, subsequent minimization required additional 78 minutes; the second run required 5 minutes and minimization with \textit{ddmin} took 30 minutes.

<table>
<thead>
<tr>
<th>Test scope</th>
<th>Total testing time (min)</th>
<th>Total minimiz. time \textit{ddmin} (min)</th>
<th>\textit{ssmin} (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EiffelBase</td>
<td>15</td>
<td>78.11</td>
<td>0.0012</td>
</tr>
<tr>
<td>Data struct.</td>
<td>5</td>
<td>30.80</td>
<td>0.0018</td>
</tr>
</tbody>
</table>

Table 6.4: Performance of testing compared to minimization with \textit{ddmin}.

Table 6.5 compares the effectiveness of \textit{ssmin} versus \textit{ddmin}. It turns out that on average, replacing \textit{ssmin} by \textit{ddmin} results in an improvement of only about 1\% of the reduction. In practice, this means that on average, \textit{ddmin} is able to remove one additional instruction.

Why is \textit{ddmin} able to minimize further than \textit{ssmin}? The \textit{ssmin} algorithm bases its reasoning on an approximation of the actual dependencies between statements, whereas \textit{ddmin} experimentally verifies causality.

The small improvement of \textit{ddmin} over \textit{ssmin} however comes at a huge cost. As shown in Table 6.6, \textit{ddmin} is much slower; in our settings, the penalty factor ranges from a factor 50 to a factor 240.

6.10 A joint approach

The two approaches discussed here need not be exclusive. We may pass the test case minimized by \textit{ssmin} over to \textit{ddmin}, which may then attempt to experimentally squeeze out the last bits. This combination of \textit{ssmin} and \textit{ddmin} is called \textit{sdmin}, and it is easily defined as

\[
\textit{sdmin}(tc) \triangleq \textit{ddmin}(\textit{ssmin}(tc))
\]

As shown in Table 6.7, the combined \textit{sdmin} approach yields the same reduction factor as \textit{ddmin} alone—that is, it is just as effective. However, it is still
Table 6.5: Average sizes of minimized test cases for \textit{ddmin}

<table>
<thead>
<tr>
<th>Test scope</th>
<th>LOC original</th>
<th>LOC minimized \textit{ssmin}</th>
<th>LOC minimized \textit{ddmin}</th>
<th>Reduction factor (%) \textit{ssmin}</th>
<th>Reduction factor (%) \textit{ddmin}</th>
</tr>
</thead>
<tbody>
<tr>
<td>EiffelBase</td>
<td>35.29</td>
<td>1.62</td>
<td>1.42</td>
<td>95.39</td>
<td>95.97</td>
</tr>
<tr>
<td>Data struct.</td>
<td>100.24</td>
<td>2.90</td>
<td>2.05</td>
<td>97.10</td>
<td>97.95</td>
</tr>
</tbody>
</table>

Table 6.6: Minimization times for \textit{ddmin}

<table>
<thead>
<tr>
<th>Test scope</th>
<th>Avg. time to minimize one test case (ms) \textit{ssmin}</th>
<th>\textit{ddmin} time / \textit{ssmin} time</th>
<th>Avg. no. of required executions \textit{ssmin}</th>
<th>\textit{ddmin}</th>
</tr>
</thead>
<tbody>
<tr>
<td>EiffelBase</td>
<td>71.40</td>
<td>3570.25</td>
<td>50.00</td>
<td>1</td>
</tr>
<tr>
<td>Data struct.</td>
<td>110.42</td>
<td>11554.51</td>
<td>104.64</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test scope</th>
<th>Average time per execution (ms) \textit{ssmin} \textit{ddmin}</th>
</tr>
</thead>
<tbody>
<tr>
<td>EiffelBase</td>
<td>71.25</td>
</tr>
<tr>
<td>Data struct.</td>
<td>110.92</td>
</tr>
</tbody>
</table>

vastly more efficient than \textit{ddmin} alone: between 11 times and 15 times faster (Table 6.8). The algorithms are dominated by execution time and thus the number of executions required for minimization, as can be seen in Table 6.9.

The \textit{sdmin} algorithm is between 4 and 7 times slower than \textit{ssmin}, but only reduces tests by less than 1% point. Is the gain in effectiveness of \textit{sdmin} over \textit{ssmin} warranted, despite its lower efficiency? If due to regression testing, tests are re-executed often it can. The following answers this question quantitatively based on a rough estimation. For each test case, minimization with \textit{sdmin} takes approximately half a seconds longer than with \textit{sdmin}. Test cases minimized by

<table>
<thead>
<tr>
<th>Test scope</th>
<th>LOC original</th>
<th>LOC minimized \textit{ssmin}</th>
<th>LOC minimized \textit{ddmin}</th>
<th>LOC minimized \textit{sdmin}</th>
<th>Reduction factor (%) \textit{ssmin}</th>
<th>Reduction factor (%) \textit{ddmin}</th>
<th>Reduction factor (%) \textit{sdmin}</th>
</tr>
</thead>
<tbody>
<tr>
<td>EiffelBase</td>
<td>35.29</td>
<td>1.62</td>
<td>1.42</td>
<td>1.42</td>
<td>95.39</td>
<td>95.97</td>
<td>95.97</td>
</tr>
<tr>
<td>Data struct.</td>
<td>100.24</td>
<td>2.90</td>
<td>2.05</td>
<td>2.05</td>
<td>97.10</td>
<td>97.95</td>
<td>97.95</td>
</tr>
</tbody>
</table>

Table 6.7: Average sizes of minimized test cases for \textit{ssmin}, \textit{ddmin}, and \textit{sdmin} compared
6.10. A JOINT APPROACH

<table>
<thead>
<tr>
<th>Test scope</th>
<th>Avg. time to minimize one test case (ms)</th>
<th>ssmin</th>
<th>ddmin</th>
<th>sdmin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EiffelBase</td>
<td></td>
<td>71.40</td>
<td>3570.25</td>
<td>316.68</td>
</tr>
<tr>
<td>Data struct.</td>
<td></td>
<td>110.42</td>
<td>11554.51</td>
<td>738.88</td>
</tr>
</tbody>
</table>

Table 6.8: Performance of \(ssmin\), \(ddmin\), and \(sdmin\) compared

<table>
<thead>
<tr>
<th>Test scope</th>
<th>Avg. no executions required</th>
<th>Average time per execution (ms)</th>
<th>ssmin</th>
<th>ddmin</th>
<th>sdmin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EiffelBase</td>
<td>1</td>
<td>14.01</td>
<td>2.10</td>
<td>71.25</td>
<td>3551.42</td>
</tr>
<tr>
<td>Data struct.</td>
<td>1</td>
<td>29.35</td>
<td>6.05</td>
<td>110.92</td>
<td>11510.41</td>
</tr>
</tbody>
</table>

Table 6.9: Execution times of \(ssmin\), \(ddmin\), and \(sdmin\) compared

\(sdmin\) execute approximately 10ms faster than test cases minimized by \(ssmin\). Minimization with \(sdmin\) pays off if tests are expected to be re-executed at least 30 times.

6.10.1 Discussion

When it comes to minimizing randomly generated unit tests, we found that \(ddmin\) works reliably, as reported by Lei and Andrews [63]. For practical applications, we found our static slicing approach to be as reliable and almost as effective, but orders of magnitude faster, which is a significant improvement over the state of the art. If tests have to be re-executed often, the added overhead of \(sdmin\) is warranted, otherwise \(ssmin\) produces almost as small tests, in a much shorter time.

The \(ssmin\) strategy, as noted, requires a test run to compensate for its unsoundness. Could we avoid this step? In principle, this is doable, but would then require a sound, reliable, and full-fledged static analysis. For the present application of simplifying test cases, we think that the additional cost of a single test run is more than outweighed by the speed of the unsound analysis as well as by

---

Unfortunately we did not measure how much time it takes to execute \(sdmin\) minimized test cases. Instead, we assume that one test case instruction is executed on average in 40ms in the interpreter and calculate the running time based on the number of instructions. This approximation is based the average size of sliced test cases from EiffelBase (1.62 lines, see Table 6.1) and the time it takes to execute the slice (71.25ms, see Table 6.3).
the additional safety obtained this way. We are unsure whether the effort required to ensure the soundness of a full-fledged static analysis would pay off in terms of saved execution time of minimized test cases. What the present study does show, however, is that even an unsound analysis can result in large performance gains over experimental approaches like delta debugging, and maybe even fully replace them.

On a larger scale, our approach also demonstrates what can be achieved by combining different techniques in quality assurance—in our case, random test generation, static program analysis, and experimental assessment.

It is not clear to what extent the presented results can be generalized to non-random based synthesis strategies. To encourage further research in these and other directions, everything needed to replicate our experiments is publicly available. A package including all data, intermediate results, the source of the tester and minimizer, as well as the compilers and tools used to compile the source is available at

http://se.ethz.ch/people/leitner/ase_min/
CHAPTER 7

RUNNING TESTS VIA REFLECTION

The system under test needs to provide an interface to be accessible for the tester. To allow for on the fly testing some kind of facility to dynamically call arbitrary code from the system under test must be available. While many new languages come with sufficient reflection support, older languages often do not (e.g. Eiffel, C++ [93], and Objective-C [85]). The present chapter explains how such languages can be made reflective through a reflection generator. It first describes an efficient language agnostic technique to build a pre-processor to alleviate the lack of sufficient reflection. It further introduces Erl-G a reflection library generator for Eiffel, that is utilized by the AutoTest synthesizer also presented in this thesis. The content of this chapter is based on our publication in the proceedings of the 45th International Conference TOOLS Europe 2007 [68].

7.1 Introduction

Reflection [60] has made its way into many programming languages in one form or another. By enabling the reification of the structure of a program and its abstract data types, structural reflection enables the program to reason about the very data it manipulates. Introspection is a subset of structural reflection focusing mainly on representing the runtime structure of a program without supporting its alteration. Behavioral reflection consists in reifying the very programming language semantics and the data used to execute a program, thus providing the possibility to manipulate operational semantics at runtime. Dynamic invocations are a subset of behavioral reflection that enable the invocation of code based on the results given by structural reflection.

In the realm of object-oriented programming languages, reflection has become especially popular, as it further supports adaptability and extensibility of pro-
grams, two of the inherent driving forces behind the object paradigm. Reflection comes in very handy for performing tests dynamically. Furthermore, introspection and dynamic invocations allow a program to adapt its behavior dynamically, depending on its internal state or on external stimuli, thus circumventing some of the limits present in statically typed languages.

The incentives and thus mechanisms for reflection vary strongly between its incarnations, depending on the semantics of the considered programming language, the architecture of the runtime environment, safety and security concerns, etc. Another strongly influential factor is legacy, i.e., the point at which the corresponding reflection mechanisms are added to the language. In the dynamically typed Smalltalk language [49], (structural) reflection appeared from the start as an integral part of the computation model. In Java [51], introspection was added gradually with hooks into the virtual machine and only a limited flavor of behavioral reflection eventually appeared, without any support from the runtime environment, in the form of dynamic proxies [94]. In the AspectJ [61] extension of Java, the interface to behavioral reflection mechanisms (the “meta-object-protocol”) has made its way into the language syntax like in 3-Lisp (level-shifting processor) [38]. The new viewpoint offered thereby on program design and development has culminated in the coining of a term denoting an intriguing new development methodology – “aspect-oriented software development”.

This chapter describes a more pragmatic approach to reflection, consisting in adding introspection to an existing object-oriented programming language a posteriori and without extending the programming language, compiler, or runtime environment in any specific manner. To that end, we introduce (1) a concise API for programmers to use, and (2) a pre-compiler which generates code for the introspectable types. This approach yields the clear benefit of dealing inherently with the legacy factor, and supporting the use of different compilers and runtime environments. We illustrate this approach through Erl-G [64], a pre-compiler generating introspection code for the Eiffel language [77]. Broadly speaking, Erl-G generates meta-classes akin to those found in Smalltalk’71 [87] for Eiffel classes. Erl-G makes it possible to introspect programs written in the Eiffel programming language, spanning many of the language constructs in a simple and uniform way, regardless of the compiler (ISE, SmartEiffel, etc.) or runtime environment (standard, .NET, etc.) at hand.

We first introduce an abstract core language which we use to explain both proposed reflection generation methods. The first, heterogeneous approach does not require the existence of a universal root type (i.e., a type without super-type to which all types in the system conform), whereas the second, more efficient, homogeneous approach does rely on the presence of such a type. Introspection is accessible through a generic API, whereby programs become entirely detached from the introspection code itself, removing any explicit dependencies between
7.2. REFLECTION GENERATION

Reflection offers a program the ability to query and/or modify its state and execution, as well as change the semantics and implementation of the programming language that it is written in. The two fundamental, complementary, aspects of full reflection are apparent from this definition: structural reflection allows a program to inspect and modify its own state as it would do with any other data that it manipulates; behavioral reflection allows a program to modify its code, as well as the very semantics and execution model of its source programming language. Both these levels of reflection require a process by which execution state is represented and made available to the program as ordinary data; this process is called reification. While the latter type of reflection has been devoted much attention in the past (see Chapter 3), we focus in this section on the former type of reflection, more widely encountered in some form or another, yet still lacking support in many languages.

7.3 Approach overview

As mentioned, many languages have been initially devised without support for reflection. In this section we deal with the more compelling case where a considered language is to be added reflective abilities, but should not change in any way.
way. As a consequence, the approach we propose is based on the generation of a set of meta-classes which represent the structure and state of the program to be reflected upon. These classes embody the necessary reification structure and make up a reflection library. Programs access these reflective features through the use of a regular library. Its API (described in detail in Section 7.7.4) offers the most common facilities needed for structural reflection: retrieving a class by its name, creating new objects, calling a routine, getting the value of an attribute, etc. Note that this API so far is not biased by any feature characteristic to any one particular programming language; rather, it allows for a set of operations that are required for structural reflection in a large family of object-oriented programming languages. The implementation of this API is then achieved through different means depending on the possibilities of the target language, and the API’s precise semantics is obviously guided by the semantics of the target programming language.

Built-in reflection facilities are often implemented using the internal run time structures of a program (see Figure 7.1(a)). We propose a preprocessor-based approach (see Figure 7.1(b)): before each compilation, a reflection library generator generates the implementation for the reflection API, which is specific to the code that should be reflective. This code, which implements a generic reflection API is added to the actual system and then both are compiled together. When the actual system makes a call to the reflection API, it executes code from the generated implementation.

7.4 A simple language

Figure 7.2 shows the syntax of a very simple abstract language on which the transformations are defined. The supported features are inspired by the Eiffel language, which is targeted by our main implementation. In particular, the proposed core
language includes multiple subtyping through multiple inheritance and generics, a feature present already early in Eiffel, and which nowadays enjoys various levels of support in many mainstream languages.

The notation $⟨V, W, ...⟩$ represents a declaration which involves different literals $V$, $W$, etc. The notation $⌈X_1, ..., X_n⌉$ denotes a list made of literals $X_1, ..., X_n$. The $⊕$ operator denotes list concatenation. The notation $Y^*$ denotes zero or more literals $Y$. The notation $Y^+$ denotes one or more literals $Y$. For example, a class is defined by a name and optional sets of

1. generic arguments,
2. super types,
3. constructors,
4. fields, and
5. routines respectively.

A type is defined by a class (its so-called base class, see Section 7.7.2), and actual types for all generic arguments. In other terms, a type is defined by a class, in which all generics have (recursively) been linked to types. A class with “open” generic arguments gives rise to a family of types but does not define a type by itself.

Traditionally names are considered to belong to different subsets of the names set depending on the way they are used. This differentiation is however not relevant for the scope of this section and subsequently ignored. Note however that names can be handled as expressions, by applying the $id$ operator. The result can be viewed as a reification of the name of some routine or field, rather than its evaluation, and reflects the first-class character strings found in many object-oriented programming languages.

The actual type system is not considered in detail at this point. A class can have several super-classes, and a formal generic argument can optionally involve a bound. Any given type system adds restrictions to the generation of meta-classes outlined in the following paragraphs. The only distinction we make subsequently is between (1) a heterogeneous and (2) a homogeneous approach. In the former case, we assume that there is no global super-type to which everything conforms (akin to Ada’95, C++, Eiffel prior to becoming an ECMA standard), and thus generate a meta-class $for each type$. In the presence of a universal root type (e.g., Java modulo primitive types, Oberon, now also ECMA Eiffel), we assume the same meta-class can be used to introspect a family of types by creating a meta-class $per class to reflect$ only. Ramifications of these approaches are discussed in Section 7.8.
### 7.5 Heterogeneous approach

In the first case mentioned above a class is generated for each type that should be made introspectable. Figure 7.3 shows, based on the abstract language introduced, how a meta class \( \mathcal{M}T[J] \cdot K \) is created for a given type. The operator \( \mathcal{M}T[J] \cdot K \) has a single argument, which is a type. The operator generates a class containing three routines, namely (1) \( \text{attr\_val} \), (2) \( \text{inv\_rout} \), and (3) \( \text{cre\_obj} \), which respectively fetch the values of a given attribute, invoke a given routine, and instantiate the class through a given constructor.

Figure 7.4 shows how to extend the application and enable reflection for a given set of types. The principle consists in generating a new class for each base class that needs reflection and to use the relevant types in the input.

Note that in this approach a meta-class is generated for each type to be made introspectable. In the absence of classes with generic arguments, there is a one-to-one correspondence between classes and types. In the presence of generic classes, each class allows for the construction of (potentially infinitely — see Section 7.9) many types.

### 7.6 Homogeneous approach

In the homogeneous approach, a meta-class is generated for each class that should be made introspectable. Figure 7.5 depicts how a meta class \( \mathcal{M}C[J] \cdot K \) is gener-
7.6. HOMOGENEOUS APPROACH

\[ MT[T] : T \rightarrow CL \]
\[ MT[T_0] = \langle mc_{T_0}, \varnothing, \varnothing, c_0, \varnothing, [M_{attr,val}, M_{inv,rout}, M_{cre, obj}] \rangle \]

where:
\[ CL_0 = \langle cl_0, \langle G_1...G_n \rangle, \langle T_1...T_m \rangle, \langle C_1...C_p \rangle, \langle F_1...F_q \rangle, \langle M_1...M_r \rangle \rangle \]
\[ T_0 = \langle CL_0, \ominus \rangle; c_0 = \langle \text{make}, \varnothing, \varnothing \rangle \]
\[ C_i = \langle c_i, \ldots \rangle; F_{i=1..q} = \langle \langle v_i, \ldots \rangle, \ldots \rangle; M_{i=1..r} = \langle m_i, \ldots \rangle \]

\[ M_{attr,val} = \langle \langle \text{attr}_val, \langle \langle \text{attr}_name, T_v \rangle, \langle \text{targ}, T_{\text{ANY}} \rangle \rangle, T_{\text{VAR}}, E_{\text{attr}} \rangle \]
\[ E_{\text{attr}} = \]
\[ \text{if } \text{id} v_1 = = \text{attr}_name \text{ then } \text{targ}.v_1 \text{ else} \]
\[ \ldots \]
\[ \text{if } \text{id} v_n = = \text{attr}_name \text{ then } \text{targ}.v_n \text{ else Error} \]

\[ M_{inv,rout} = \langle \langle \text{inv}_rout, \langle \langle \text{rout}_name, T_{\text{NAME}} \rangle, \langle \text{targ}, T_{\text{ANY}}, \langle \text{args}, T_{\text{LIST}[\text{ANY}]} \rangle \rangle, T_{\text{VAR}}, E_{\text{proc}} \rangle \rangle \]
\[ E_{\text{proc}} = \]
\[ \text{if } \text{id} m_1 = = \text{rout}_name \text{ then } \text{targ}.m_1(\text{args}_1..n) \text{ else} \]
\[ \ldots \]
\[ \text{if } \text{id} m_m = = \text{rout}_name \text{ then } \text{targ}.m_m(\text{args}_1..n) \text{ else Error} \]

\[ M_{cre, obj} = \langle \langle \text{cre}_obj, \langle \langle \text{cons}_name, T_c \rangle, \langle \text{args}, T_{\text{LIST}[\text{ANY}]} \rangle \rangle, T_{\text{ANY}}, E_{\text{cons}} \rangle \]
\[ E_{\text{cons}} = \]
\[ \text{if } \text{id} c_1 = = \text{cons}_name \text{ then create } T_0.c_1(\text{args}_1..n) \text{ else} \]
\[ \ldots \]
\[ \text{if } \text{id} c_k = = \text{cons}_name \text{ then create } T_0.c_k(\text{args}_1..n) \text{ else Error} \]

Figure 7.3: Generation per type (heterogeneous approach)
\[ \mathcal{M}_T_{\text{app}}[\cdot, \cdot] : A \times T_{\text{LIST}[T]} \to A \]
\[ \mathcal{M}_T_{\text{app}}[A_0, [T_1, \ldots, T_m]] = \langle [CL_1, \ldots, CL_n, \mathcal{M}_T[T_1], \ldots, \mathcal{M}_T[T_m]], \text{create } T_1.c(\text{Void}) \rangle \]

where:
- \( T_{\text{SET}} \) is the set of sets of types
- \( A_0 = \langle [CL_1, \ldots, CL_n], \text{create } T_1.c(\text{Void}) \rangle \)

Figure 7.4: Per-type reflection library generation

ated for a given class. As opposed to the operator \( \mathcal{M}_T[\cdot] \) in the heterogeneous approach, \( \mathcal{M}_C[\cdot, \cdot] \) takes two arguments: a (base) class and a set of derived types for that class (a type is derived from a class iff the class is the base class of the given type). This set contains the types for which instances should be creatable via reflection.

The operator \( \mathcal{M}_C[\cdot, \cdot] \) generates a class containing three routines, namely (1) \( \text{attr_val} \), (2) \( \text{inv_rout} \), and (3) \( \text{cre_obj} \). Just like in the case of \( \mathcal{M}_T[\cdot] \), these fetch the values of attributes, invoke routines and create instances, respectively, as shown in Figure 7.5. Similarly to the heterogeneous approach, Figure 7.6 shows how to extend an application and enable reflection.

7.7 The case of Eiffel

To illustrate the general solution described in the previous section, we present here its implementation in the case of the Eiffel language [75]. In a nutshell, Eiffel is purely object-oriented, strongly typed, and supports dynamic binding, (dynamic) single dispatch, and multiple inheritance. Eiffel makes for a very illustrative target language, since it innately only provides minimal support for reflection. We first overview the Eiffel language along with its type system, depicting how it impacts the implementation of the previously outlined introspection mechanisms, and then look at the API of the generated Eiffel reflection library.

7.7.1 The Eiffel language

Figure 7.7 overviews the core of the Eiffel language syntax, as a “mapping” from the abstract syntax presented in Figure 7.2. In the case of optional declarations (enclosed in \([\ldots]\) in Figure 7.2), leading keywords (e.g., \( \text{inherit} \)) are omitted in case of absence of corresponding declarations. The same goes for separating symbols (e.g. “,”) in case of multiple declarations. Note that the \textit{list} from Figure 7.2...
7.7. THE CASE OF EIFFEL

\[ \mathcal{FL}[\cdot] : T \rightarrow T_{LIST[T_NAME]} \]

\[ \mathcal{FL}[T_0] = [CL_0] \oplus \mathcal{FL}[T_1] \oplus ... \oplus \mathcal{FL}[T_m] \]

where:

\[ T_0 = \langle (\langle \cdot, \cdot \rangle, \{T_1, ..., T_m\}) \rangle \]

\[ \mathcal{MC}[\cdot, \cdot] : CL \times T_{LIST[T]} \rightarrow CL \]

\[ \mathcal{MC}[CL_0, \{T_{CR(1)}, ..., T_{CR(n)}\}] = \]

\[ \langle m_{CL_0}, \emptyset, \emptyset, \emptyset, \emptyset, [M_{\text{attr-val}}, M_{\text{inv-rout}}, M_{\text{cre-obj}}] \rangle \]

where:

\[ CL_0 = \]

\[ \langle \langle \cdot, \cdot \rangle, \{G_1, ..., G_n\}, \{T_1, ..., T_m\}, \{C_1, ..., C_p\}, \{F_1, ..., F_q\}, \{M_1, ..., M_r\} \rangle \]

\[ c = \langle \text{make}, \emptyset, \emptyset \rangle; T_{CR(i=1, t)} = \langle CL_0, \ldots \rangle \]

\[ C_{i=1,p} = \langle c_1, \ldots \rangle; F_{i=1,q} = \langle \langle v_1, \ldots \rangle, \ldots \rangle; M_{i=1,r} = \langle m_1, \ldots \rangle \]

\[ M_{\text{attr-val}} = \]

\[ \langle \text{attr-val}, \{\langle \text{attr-name}, v_1\rangle, \langle \text{args}, T_{\text{ANY}}\rangle\}, T_{\text{ANY}}, E_{\text{attr}} \rangle \]

\[ E_{\text{attr}} = \]

\[ \text{if } v_1 == \text{attr-name then target.v1 else} \]

\[ \text{... if } v_n == \text{attr-name then target.vn else Error} \]

\[ M_{\text{inv-rout}} = \]

\[ \langle \text{inv-rout}, \{\langle \text{rout-name}, T_{\text{NAME}}\rangle, \langle \text{args}, T_{\text{LIST[ANY]}\rangle}, T_{\text{ANY}}, E_{\text{proc}} \rangle \]

\[ E_{\text{proc}} = \]

\[ \text{if } m_1 == \text{rout-name then target.m1(args1, ...)} \text{ else} \]

\[ \text{if } m_m == \text{rout-name then target.mm(args1, ...)} \text{ else Error} \]

\[ M_{\text{cre-obj}} = \]

\[ \langle \text{cre-obj}, \{\langle \text{cons-name}, T_c\rangle, \langle \text{args}, T_{\text{LIST[ANY]}\rangle}, \langle \text{flat-type}, T_{\text{LIST[T_NAME]}\rangle}, T_{\text{ANY}}, E_{\text{cons}} \rangle \]

\[ E_{\text{cons}} = \]

\[ \text{if flat-type == FL[T_{CR(1)}] then} \]

\[ \text{if } c_1 == \text{cons-name then create } T_{CR(1)}, c_1(args1, ...)} \text{ else} \]

\[ \text{... if } c_k == \text{cons-name then create } T_{CR(1)}, c_k(args1, ...)} \text{ else Error} \]

\[ \text{... else if flat-type == FL[T_{CR(n)}] then} \]

\[ \text{if } c_1 == \text{cons-name then create } T_{CR(n)}, c_1(args1, ...)} \text{ else} \]

\[ \text{... if } c_k == \text{cons-name then create } T_{CR(n)}, c_k(args1, ...)} \text{ else Error} \]

\[ \text{else Error} \]

Figure 7.5: Generation per class (homogeneous approach)
 MC app : A × T LIST[T] → A

\[ MC app[A_0, [T_1, ..., T_m]] = \langle [CL_1, ..., CL_n], MC[CL_1, [T_1, ..., T_m]/cl_1], ..., MC[CL_n, [T_1, ..., T_m]/cl_n], create T_1.c(Void) \rangle \]

where:

\[ A_0 = \langle [CL_1, ..., CL_n], create T_1.c(Void) \rangle \]

\[ CL_{i=1..m} = \langle cl_1, ... \rangle \]

\[ [T_1, ..., T_m]/cl_0 = [T_{j=1..m}|T_j = \langle cl_0, ..., \rangle] \]

Figure 7.6: Per-class reflection library generation

does not appear here, as it is replaced by the generic LIST type.

7.7.2 Types

Informally, types in Eiffel are divided into three categories:

**References:** If an entity \( x \) is declared of type \( T \) and \( T \) is a reference type, then the value of \( x \) will be a reference to an object or Void.

**Expanded:** If an entity \( x \) is declared of type \( T \) and \( T \) is an expanded type (also called value type), then the value of \( x \) will be an object. The predefined expanded types (such as INTEGER, REAL, DOUBLE, BOOLEAN, and CHARACTER) are so-called basic types, but other expanded types can also be created by using the expanded keyword. The basic types are implemented by library classes, but, for obvious reasons of efficiency, compilers will implement operations on them directly through machine operations, not through routine calls.

**Formal generics:** In the case of generic classes (also called parameterized types), a formal generic represents a type parameter to be provided in actual uses of the class. The Eiffel genericity mechanism allows for both unconstrained and constrained (bounded) generic parameters; constraints in the latter case are expressed by the \( \rightarrow \) operator. An example would be the second parameter of class HASH_TABLE \( [G, H \rightarrow HASHABLE] \). Any type that is a valid generic derivation of this class must have, as the second actual generic parameter, a type that conforms to HASHABLE.

Every type is based on a class, which is called the type’s “base class”. The difference between types and classes appears only in the case of generic classes:
\[ \langle cl, [G^*], [T^*], C^*, [F^*], [M^*] \rangle \triangleq \text{class } cl[G_1, \ldots, G_m] \]

\[ \text{inherit } T_1, \ldots, T_n \]

\[ \text{feature} \]

\[ C_1, \ldots, C_p, F_1, \ldots, F_q, M_1, \ldots, M_r \]

\[ \text{end} \]

\[ \langle g, T \rangle \triangleq g \rightarrow T \]

\[ \langle g, \perp \rangle \triangleq g \]

\[ \langle v, T \rangle \triangleq v : T \]

\[ \langle v, g \rangle \triangleq v : g \]

\[ \langle V, E \rangle \triangleq V \text{ is } E \]

\[ \langle cl, [T^*] \rangle \triangleq cl[T_1, \ldots, T_n] \]

\[ \langle c, [V^*], E \rangle \triangleq \text{create } c(V_1, \ldots, V_n) \text{ is} \]

\[ \text{do } E \text{ end} \]

\[ \langle m, [V^*], T, E \rangle \triangleq m(V_1, \ldots, V_n) : \text{Tis} \]

\[ \text{do } E \text{ end} \]

\[ \langle m, [V^*], g, E \rangle \triangleq m(V_1, \ldots, V_n) : g \text{is} \]

\[ \text{do } E \text{ end} \]

\[ \langle m, [V^*], \perp, E \rangle \triangleq m(V_1, \ldots, V_n) \text{ is} \]

\[ \text{do } E \text{ end} \]

Figure 7.7: Simplified Eiffel language syntax
class ERL_CLASS
feature

attribute_value (name: STRING; target: ANY): ANY
  -- Value of attribute named 'name' of object 'target'

invoke_routine (name: STRING; target: ANY; args: ARRAY [ANY]): ANY
  -- Invoke routine 'name' of object 'target' using the
  -- items contained in 'args' as routine arguments.

create_object (name: STRING; type: STRING; args: ARRAY [ANY]): ANY
  -- Newly created object of type 'type' using creation
  procedure
  -- named 'name' and arguments from 'args'.
end

class ERL_TYPE
feature

attribute_value (name: STRING; target: VARIANT): VARIANT
  -- Value of attribute named 'name' of object 'target'

invoke_routine (name: STRING; target: VARIANT; args: ARRAY [VARIANT]): VARIANT
  -- Invoke routine 'name' of object 'target' using the
  -- items contained in 'args' as routine arguments.

create_object (name: STRING; args: ARRAY [VARIANT]): VARIANT
  -- Newly created object using creation procedure
  -- named 'name' and arguments from 'args'.
end

Listing 7.1: Interface of ERL_CLASS and ERL_TYPE

such classes do not describe a type, but a family of types. To obtain a type from a
generic class, one must provide actual generic parameters. For instance, the class
declared as LIST [G] only yields a type when a type existing in the system, such
as INTEGER, is substituted for G; hence, LIST [INTEGER] is a type.
7.7.3 Subtyping

Subtyping in Eiffel is defined as follows. A type $v$ is a subtype of a type $\tau$ if and only if both:

1. The base class of $v$ is a descendant of the base class of $\tau$.

2. If $v$ is generically derived, its actual generic parameters must subtype those of $\tau$'s corresponding ones.

Our first implementation of the meta-class generator for Eiffel was aimed at the pre-ECMA version of the Eiffel language, in which expanded types do not conform to $\text{ANY}$. As a consequence a meta-class had to be generated for every type, mandating the heterogeneous code generation. The introduction of a universal root type (in the ECMA standard) significantly improved the performance of the meta-class generator, since it meant that the generation of a meta-class for every type was no longer necessary: the generation of a meta-class for every class was sufficient. In Section 7.8.5 we assess the two approaches in more detail in terms of performance and limitations.

7.7.4 The Erl-G library API

As stated in Section 7.2, the introspection-enabling code is generated as a library of meta-classes. One such meta-class (inheritng from $\text{ERL\_CLASS}$) is generated for each class that requires introspection abilities.

The meta-model of the reflection API is represented in Figure 7.8. The main components of this simple meta-model are the classes $\text{ERL\_CLASS}$ and $\text{ERL\_UNIVERSE}$. Class $\text{ERL\_SHARED\_UNIVERSE}$ implements the singleton design pattern, by ensuring that only one instance of $\text{ERL\_UNIVERSE}$ exists and providing the access point to it. $\text{ERL\_CLASS}$ is the abstract ancestor of all generated meta-classes for the classes that must be introspectable. It provides the routines for creating a new instance, retrieving the value of an attribute, and calling a routine. For brevity, the arguments to these routines are omitted in Figure 7.8; they are however visible in Listing 7.1. The implementation of each routine can be thought of as a big switch on the name of the constructor, attribute, or routine, respectively. For languages supporting overloading the switch must be extended to also cover the types of the routine arguments. Note that, since Eiffel does not support overloading, we did not need to consider this issue in our implementation.

Class $\text{ERL\_UNIVERSE}$ provides routines for retrieving a meta-class through the name of its corresponding class and through an object that is an instance of this class.

It should also be noted that a very limited set of features related to reflection was already present in the Eiffel language, through class $\text{INTERNAL}$. This class can
check whether an object is an instance of a given type, can obtain the class name
and type name for a given object, can verify type conformance, and can query and
set the values of fields. The most notable capabilities missing are those of calling
routines and constructors and querying the arguments of a routine. However, none
of the features provided by \textit{INTERNAL} are necessary for our implementation, as
explained in Section 7.9.

![Meta-model for the homogeneous generation method](image)

**Figure 7.8: Meta-model for the homogeneous generation method**

### 7.8 Implementation

This section describes how the reflection library API for Eiffel presented in the
previous section can be implemented using a code generator. The general idea
is to apply a pre-compiler on the code that should be made introspectable and to
generate a custom-fit implementation for the generic reflection API.

#### 7.8.1 Erl-G

Our generators presented above have been implemented in the Erl-G tool. More
precisely, this tool implements both the homogeneous and the heterogeneous metaclass
generation techniques. Erl-G is published as open source and can be downloaded as binary and source [64]. Erl-G relies on the front-end of the Gobo Eiffel
7.8. IMPLEMENTATION

In the following we look at the high-level architecture of the tool, examine the details of the heterogeneous and homogeneous code generation methods, and provide an example to illustrate the approach.

7.8.2 General architecture

Both methods of generating code share a common architecture. The reflection library is generated via a pre-compiler before compilation. The pre-compiler parses the system (much like the first phases of the actual compiler) and then generates the corresponding meta-classes. The generated meta-code implements a predefined and static reflection API (described in section 7.7.4). The compiler then compiles a program consisting of the actual system, the reflection API, and the generated reflection implementation. Note that as soon as some parts of an interface within the actual system change, the pre-compilation step needs to be repeated.

For the pre-compiler to parse the initial system it is necessary that this system can be type-checked even before the pre-compiler has generated its meta-code. In order to solve this problem, a “dummy” reflection implementation is used. Thus, the system can be type-checked also before the real reflection implementation is available. As an added benefit, the dummy implementation allows the initial program to compile and be self-contained even if no meta-code has been generated yet.

7.8.3 Introspectable types in the heterogeneous and homogeneous approaches

Section 7.2 presented the two possibilities for generating meta-classes. Both instrumentations require a list of types (to be made introspectable) to generate the meta-classes. In practice this list is mostly generated automatically. In the heterogeneous approach, Erl-G generates a meta-class per type while the homogeneous approach has only one meta-class per class but needs the list of types instantiatable via reflection.

For the heterogeneous approach, the default behavior is to derive: (1) one type per non-generic class, (2) the most abstract derivation (i.e. the one where each formal generic parameter is replaced by its constraining type) for each generic class, and (3) each type that could be instantiated by the program at runtime. This last part is computed using the dynamic type set algorithm [76], which produces an over-approximation of the set of types that can be instantiated in the system.

For the homogeneous approach, the default behavior is to derive: (1) one type per non-generic class, (2) the most abstract derivation for each generic class.
All classes are then introspectable. Non-generic classes can be instantiated and generic classes’ most abstract derivations can be instantiated.

In practice, rather than redefining the default list, users can simply extend it.

### 7.8.4 Illustrations

To illustrate how Erl-G works, we use the example of a simple banking application. The main class of this application is outlined in Listing 7.2. Implementation details and contracts which are not germane to the subsequent presentation have been omitted for brevity. We use the homogeneous approach for this example.

```plaintext
class BANK_ACCOUNT
    feature
        make (p: PERSON; init_bal: INTEGER)
            -- Create new account owned by ‘p’ as owner
            -- having an initial balance of ‘init_bal’.
        make_with_default_balance (p: PERSON)
            -- Create new account owned by ‘p’.
        owner: PERSON
            -- Person who owns the account
        balance: INTEGER
            -- Balance of account
        withdraw (sum: INTEGER)
            -- Withdraw ‘sum’ from this account.
        set_owner (p: PERSON)
            -- Transfer ownership of this account to ‘p’.
        set_balance (b: INTEGER)
            -- Set balance of account to ‘b’.

end
```

Listing 7.2: Classes in the example

The homogeneous generation method creates a meta-class for each existing class and then one class that serves as a lookup-point for all meta-classes. Due to size restrictions we cannot show the generated code.

Listing 7.3 illustrates how the reflection library generated by Erl-G can be used. Note that this code only uses the general interface described in section 7.7.4.
class BANK_REFLECTION

inherit ERL_SHARED_UNIVERSE

feature

execute is

local

  c: ERL_CLASS
  p: ANY
  ba: ANY
  tmp: ANY

  do
    c := universe.class_by_name ("PERSON")
    p := c.create_object ("make", "", <<35, "John Doe">>)
    c := universe.class_by_name ("BANK_ACCOUNT")
    ba := c.create_object ("make", "", <<p, 300>>)
    tmp := c.invoke_routine ("withdraw", ba, <<20>>)
  end

end

Listing 7.3: Use of reflection API

and does not depend in any manner on the way in which the actual meta-code is generated.

7.8.5 Evaluation

This section evaluates the Erl-G tool and compares the two generation methods: the heterogeneous one (which generates a meta-class for every type) and the homogeneous one (which generates a meta-class for every class). First we show the results of some tests we performed using Erl-G and then discuss the applicability of the two methods.

Efficiency

We have used both the heterogeneous and the homogeneous generation methods implemented by Erl-G to make a set of libraries introspectable. As entry point for each library (in order to be able to generate an executable) we used a class implementing an interpreter. This interpreter reads instructions from the standard input and then uses the reflection API to execute the actual code (routine calls,
creation of objects, etc.).

Table 7.1 provides some figures illustrating the efficiency of both generation methods. Table 7.2 shows the factor gained by using the homogeneous method over the heterogeneous one. The tables show statistics of the application of Erl-G on three libraries: Base, the Eiffel standard I/O and data-structure library for the ISE Eiffel compiler; Gobo, a portable standard library and tool collection; Vision2, a multi-platform GUI library.

The tests were performed on a custom-built desktop PC equipped with an AMD Athlon 64 dual core and 2 GB RAM using a 64 bits version of Ubuntu Linux.

The top part of Table 7.1 shows the results for the heterogeneous generation and the bottom part shows the results for the homogeneous generation. In each part the columns represent:

1. Number of classes to be made introspectable,
2. Compilation time with dummy reflection implementation,
3. Pre-compilation time,
4. Number of meta-classes generated,
5. Compilation time with generated reflection implementation.

All times are given in the format hh:mm:ss.

Note, that the compilation time with the dummy reflection generation is almost constant. This is because the global analysis of the compiler determines that the executable to produce is only dependent on a fixed number of classes: the interpreter and the dummy implementation.

For the heterogeneous generation both the pre-compilation time and the full compilation time increase with the number of types in the system (which is greater than the number of classes). In the case of Vision2, the heterogeneous implementation takes over 4 hours to generate the meta-classes and the compilation of the full program crashes after 12 hours. Note that the number of classes is slightly different in both cases for each library as the new version of the compiler was not backward compatible and thus it was needed to process older versions for the heterogeneous approach. In each case, the most recent version of the libraries that would still compile with the heterogeneous compiler was used.

These results show that, due to efficiency reasons, the homogeneous method should always be preferred over the heterogeneous one for languages with universally-rooted type systems. For the other languages, the homogeneous method cannot be used.
7.9. APPLICABILITY

As one of the primary goals of our work was to develop an approach that is applicable as widely as possible, our reflection generation method has a minimal set of requirements: the target language must provide a means of obtaining the type of an object (or, in its absence, a safe casting mechanism) and the source code for the introspectable classes must be available. A higher resolution of the type information provided by the system makes it easier to provide accurate information upon introspection. Consider the case of Java; the absence of runtime support makes it hard to provide information on actual generic arguments.

As explained in Section 7.8.3, any system contains at least as many types as classes. Hence, the heterogeneous method generates more meta-code and causes longer generation and compilation time, and a bigger memory foot-print. To compensate, a user can choose to lower the number of types that should be introspectable and Erl-G provides built-in facilities to easily customize the generation process.

In some circumstances it is not even possible to generate meta-code for all classes.
<table>
<thead>
<tr>
<th></th>
<th>#classes</th>
<th>compil. time</th>
<th>gener. time</th>
<th>#meta-classes</th>
<th>compil. time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>without</td>
<td>with meta-classes</td>
<td>(hh:mm:ss)</td>
<td>with meta-classes</td>
<td>(hh:mm:ss)</td>
</tr>
<tr>
<td>Base</td>
<td>1.08</td>
<td>0.75</td>
<td>38</td>
<td>6.65</td>
<td>36.16</td>
</tr>
<tr>
<td>Gobo</td>
<td>1.01</td>
<td>0.69</td>
<td>103.66</td>
<td>5.35</td>
<td>47.41</td>
</tr>
<tr>
<td>Vision2</td>
<td>1.02</td>
<td>0.56</td>
<td>2401.14</td>
<td>9.46</td>
<td>∞</td>
</tr>
</tbody>
</table>

Table 7.2: Speedup factor of homogeneous approach over heterogeneous approach (time required with homogeneous approach divided by time required with heterogeneous approach)

alive types statically, since there are potentially infinitely many. Such a situation occurs for example for a generic class $\text{SET} \,[G]$, that includes a routine that returns its powerset. The return type of such a routine is $\text{SET} \,[\text{SET} \,[G]]$. This routine may be used in a loop, resulting in an unknown number of types.

The heterogeneous method is limited to a fixed subset of the potentially infinite number of types. The homogeneous method does not suffer from this problem since all classes can be made introspectable, but some types will not be instantiatable through the reflection library.

The pre-compiler approach makes it possible to add reflection a posteriori to a language without the need to modify the language, the core libraries, or the execution environment. However, in the presence of dynamic class loading, pre-compilation is not sufficient. When a new class is loaded, a new meta-class needs to be generated, loaded into the program, and registered with the universe object. We did not implement such a facility in Eiffel due to its lack of support for dynamic class loading.

### 7.10 Reflection generation in AutoTest

AutoTest as described earlier is a fully automatic testing tool for contract-equipped classes. It covers test case creation, execution, and evaluation. The user invokes AutoTest by specifying a set of classes to be tested. AutoTest then tests these classes and produces statistical results about the testing process and bug-reproducing witnesses (fragments of code which, when run, will cause a failure, thus showing the presence of the bugs).

For every class under test, AutoTest synthesizes a set of object creations and routine invocations to execute. During the execution of these instructions, the con-
tracts (in the form of preconditions, postconditions and invariants) are monitored for violations. Depending on the type of violation, AutoTest can decide whether a class was used improperly or a bug has been found.

To increase robustness, AutoTest separates test case creation and execution into two separate processes. The actual execution is performed by a special purpose interpreter. This interpreter can receive (via standard input) and execute simple instructions, such as calling a routine on an object, creating a new object, assigning values to variables, etc. The instructions are written in a subset of the Eiffel language.

Since this interpreter needs the ability to execute a routine given by its name and the Eiffel compiler used lacks the possibility to reflect routines, Erl-G is used to implement the interpreter. Every instruction read from standard input is parsed, then the required meta-class is looked up and the reflection interface is used to invoke the requested routine.

7.11 Summary

Reflection has become a fundamental feature of programming languages, not only object-oriented ones, going as far as impacting the very design and underlying computation model of recent languages, and even advocating for novel software development paradigms [61].

Many programming languages however have been initially introduced without reflection capabilities, or have been extended after their inception with features not covered by the initial reflection mechanisms. In particular languages compiled to native code, devoid of a standardized compilation and runtime infrastructure, are very prone to such extensions remaining beyond the reach of reflection. Even Java, in spite of its “compile once, run everywhere” credo, is a popular example of such a shortage: it has been augmented with generics lately but lacks support for introspection of corresponding generic parameters.

This section proposes a way out of such bottlenecks, by describing an approach for the generation of introspection libraries leveraging static program analysis. We have described our approach starting from a general context – a simple abstract language including generics. We have presented two variants of our generation function, depending on whether the type system of the target language has a universal root type (e.g., Java, Oberon) or not (e.g., C++, Ada’95). We discussed consequences of these options in qualitative as well as quantitative terms – showing an order of magnitude gained in both time (duration of program compilation) and space (number of meta-classes generated) when a universal root type exists. We presented workarounds for limitations such as infinite recursion, and discussed minimal support required from a target programming language.
The Eiffel language has been used in this section as an illustration for all of our contributions, as it has yielded the initial motivation for this work: only a very recent overhaul of the language has provided its type system with a universal root type, and several independent, highly evolved, compilers exist for it. We have illustrated the benefits of our approach in the Eiffel language for instance through AutoTest, an automatic program testing tool. We are currently in the process of validating our approach with other languages, and extending Erl-G with behavioral reflection.
CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Future work

Our current testing techniques do not cope with concurrent applications. The SCOOP [9] mechanism extends the semantics of contracts to the concurrent case. A contract equipped language supporting SCOOP could allow us to extend our techniques to handle concurrent applications.

The recently introduced selective capture and replay mechanism [83, 82, 88, 43] promises much increased performance of both the capture and replay phase. We are currently working on a selective capture and replay implementation that works together with test extraction.

Pre-state extraction. To correctly extract a test case from a failure, the state (i.e., the relevant objects on the heap) has to be captured right before the recipient of the failure is called. By default, we capture the state at the time of the failure. Only for repeated failures we capture the true pre-state.

Non-determinism. While most programming languages do not provide a source for non-determinism directly, programmers can typically use the foreign function interface to acquire external inputs (user input, network, database, etc.), and this can be considered a cause of non-determinism within the program. With selective capture and replay it is possible to put all sources of non-determinism into the external part, which makes replays completely deterministic. Even if a failure-producing trace was dependent on certain GUI inputs, network connections, or database state, the replay is completely freed from these dependencies.

External state The foreign function interface also introduces references to external state (e.g. window and file handles). Due to a lack of type information
for it, this state it cannot be serialized automatically. By choosing the capturing border so that all external state lies in the unobserved part, external state can be mocked out completely.

Our test synthesis and minimization techniques are based on techniques like random test generation, static program analysis, and experimental assessment. We believe that the future of program validation lies in such pragmatic combinations, and this is also what our future work will focus upon:

**Improved slicing.** As stated in Section 6.7, our static slicing algorithm is very fast, but unsound. We want to explore which other analyses (such as interprocedural, alias or points-to analyses) can be added to improve effectiveness and efficiency. *Dynamic slicing* is a particularly attractive option, as we need the slice only for the specific run at hand.

**Optimization techniques.** Reconsidering the minimized test case in Listing 6.9, Page 106. An advanced analysis could use *constant folding* together with data flow analysis to eliminate spurious assignments and reduce the test case to

```
create (RANDOM) v_135.set_seed (2)

v_136 := v_135.real_item
```

We are currently exploring which other optimizations from compiler construction could help in producing simpler test cases.

**Duplicate elimination.** As random testing becomes more effective, one can run a larger number of random tests in less time. We expect that several of these random tests will hint at the same fault, but end up with different failures. Identifying failures that are due to the same fault is interesting for two reasons: due to the grouping, the results are easier to browse for the developer and he may choose the test case suited best for further debugging from each group.

Much research has recently gone into making the output of model checkers easier to understand [92, 11, 53]. The counter examples produced by the model checker are similar in concept to failing test cases and it would be interesting to investigate if some of the findings from this community can be transferred.

**Automatic fixes.** The combination of generation, minimization, and automatic tests allows for identifying *missing routine calls* that, when added, cause the test to pass. In our example above, for instance, we could systematically generate additional routine calls on the objects involved and thus identify a missing call:
8.2. CONCLUSIONS

Create (RANDOM) v_135.set_seed (2)

v_135.start

v_136 := v_135.real_item

Inserting Line 148a, as above, makes the test pass – and indeed, the omission of this call is exactly what makes the defect in the set_seed constructor. If there were a way to synthesize candidate fixes, a large regression test suite could help to identify those ones that are likely to fix the failure, but do not introduce other faults.

8.2 Conclusions

Testing is an important instrument for increasing the quality of software. Many modern development methods put testing at their center. This shift places new challenges on testing techniques and tools. Since it is no longer considered an afterthought, testing also becomes a primary concern for the developer. More tests have to be created, they have to be run more often, and they have to give fast feedback. This thesis presents testing techniques, supported by the benefits provided by Design By Contract, that bring testing closer to the developer.

We present a test case extractor that observes the developer as he manually tests the software he develops. In the event of a failure our approach automatically generates a failure-reproducing test case. This turns a test case that requires manual execution and is easily forgotten into an automatically executing test case that aids debugging and remains in the regression test suite forever. Our extraction technique is based on program state. The state of the program is serialized at the time of the failure. Upon execution of the test case, this state is recreated and the routines that were on the stack at the time of the failure are invoked again. In contrast to other approaches our extraction process comes at no additional runtime-overhead and integrates directly into the main tool of developers, the integrated development environment. Our approach trades efficiency for effectiveness: in order to reliably reproduce a failure, we need the state of the program before the failure, instead we capture the state at the time of the failure (because it does not incur any execution overhead). Surprisingly, our evaluation shows that despite this weakness our approach is effective. A user study showed that our mechanism extracts reproducing test cases for almost all failures.

In order to make testing more responsive, this thesis presents a fast prioritization algorithm. Given the latest change a developer made to a program, the algorithm identifies those test cases that need to be rerun. Our approach is based on lexical analysis only, requiring no test execution. It is fast enough to be run before each testing session.
Contracts enable fully automated testing. Depending on the synthesis strategy used, the resulting test cases can grow large and become a burden further debugging and maintenance. This thesis presents a fully automated tester (AutoTest) and shows how the resulting test cases can be minimized. Minimized test cases provide two benefits: they give faster feedback (because they execute faster) and they are easier to maintain (because they consist of less code). We propose two minimization techniques: one based only on static slicing and one based on a combination of static slicing and delta-debugging. Our proposed minimization approach based on static slicing is highly efficient, practical, and easy to implement. In our evaluation, running static minimization came at virtually no cost, and the speed gain of the resulting minimized test case pays off with the first execution. To gain another 1% in minimization, one can combine the approach with an additional delta debugging step. This combination yields the same results as delta debugging (the state of the art) alone, but is far more efficient.

In summary, developers need better techniques and tools in order to cope with the increased importance of testing. This thesis tries to bring such techniques and tools to them.


CURRICULUM VITAE

General information

Name: Andreas Leitner
Date of birth: January 16, 1977
Citizen of: Austria
E-mail address: aleitner@raboof.at

Diplomas

• Diplom-Ingenieur (Dipl.-Ing.), TU Graz, Austria (2005), with distinction
• Bakkalaureus der technischen Wissenschaften (Bakk.techn.), TU Graz, Austria (2004)

Selected published papers


• Leitner, A., Eugster, P., Oriol, M., Ciupa, I., "Reflecting on an Existing Programming Language", Proceedings of TOOLS EUROPE 2007 - Objects, Models, Components, Patterns, (Zurich, Switzerland), June 2007