

# Automatic Generation Of Object-Oriented Unit Tests Using Genetic Programming

Stefan Wappler, M.Sc.

Von der Fakultät IV – Elektrotechnik und Informatik  
der Technischen Universität Berlin  
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften  
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:	Prof. Dr. rer. nat. Peter Pepper (Vorsitzender)
	Prof. Dr.-Ing. Ina Schieferdecker (Berichter)
	Prof. Dr.-Ing. Stefan Jähnichen (Berichter)

Tag der wissenschaftlichen Aussprache:	19. Dezember 2007
---	-------------------

Berlin 2008  
D 83



## Acknowledgements

I would like to express my sincere gratitude to my supervisors, Ina Schieferdecker and Joachim Wegener, for their professional guidance, inspiring discussions, and encouragement during the period of this research. My thanks are also due to Stefan Jähnichen for his broad support and guidance. Furthermore, I would like to thank all my colleagues from both Daimler AG and the Technical University of Berlin. In particular, I thank all my DCAITI team members for the very good cooperation and time we had together: Andreas Windisch, Fadi Chabarek, Linda Schmuhl, Abel Marrero-Perez, Kerstin Buhr, Steffen Kühn, Andrea Tüger, Oliver Heerde. I also thank in particular Harmen Sthamer for his review of this thesis. I gratefully acknowledge the encouraging meetings and discussions with Mark Harman from King's College, London, Phil McMinn from Sheffield University, Leonardo Bottaci from Hull University, and all other participants of the EvoTest project. A special thank is due to Andrea Tüger, Oliver Heerde, and Richard Norridge from London for their quick and straightforward language-oriented review of this thesis. Thanks also go to my friends, my parents, my family, my in-laws for all support and encouragement. Finally my greatest thanks to Lord Jesus Christ, who enabled me to perform this research and who accomplishes everything according to His glorious mind.



## Abstract

Automating the generation of object-oriented unit tests for structural testing techniques has been challenging many researchers due to the benefits it promises in terms of cost saving and test quality improvement. It requires test sequences to be generated, each of which models a particular scenario in which the class under test is examined. The generation process aims at obtaining a preferably compact set of test sequences which attains a high degree of structural coverage. The degree of achieved structural coverage indicates the adequacy of the tests and hence the test quality in general.

Existing approaches to automatic test generation for object-oriented software mainly rely either on symbolic execution and constraint solving, or on a particular search technique. However, these approaches suffer from various limitations which negatively affect both their applicability in terms of classes for which they are feasible, and their effectiveness in terms of achievable structural coverage. The approaches based on symbolic execution and constraint solving inherit the limitations of these techniques, which are, for instance, issues with scalability and problems with loops, arrays, and complex predicates. The search-based approaches encounter problems in the presence of complex predicates and complex method call dependences. In addition, existing work addresses neither testing non-public methods without breaking data encapsulation, nor the occurrence of runtime exceptions during test generation. Yet, data encapsulation, non-public methods, and exception handling are fundamental concepts of object-oriented software and require also particular consideration for testing.

This thesis proposes a new approach to automating the generation of object-oriented unit tests. It employs genetic programming, a recent meta-heuristic optimization technique, which allows formulating the task of test sequence generation as a search problem more suitably than the search techniques applied by the existing approaches. The approach enables testing non-public methods and accounts for runtime exceptions by appropriately designing the objective functions that are used to guide the genetic programming search.

The value of the approach is shown by a case study with real-world classes that involve non-public methods and runtime exceptions. The structural coverage achieved by the approach is contrasted with that achieved by a random approach and two commercial test sequence generators. In most of the cases, the approach of this thesis outperformed the other methods.



## Zusammenfassung

Die Automatisierung der Testfallermittlung für den struktur-orientierten Unit-Test objektorientierter Software verspricht enorme Kostenreduktion und Qualitätssteigerung für ein Softwareentwicklungsprojekt. Die Herausforderung besteht darin, automatisch Testsequenzen zu generieren, die eine hohe Überdeckung des Quellcodes der zu testenden Klasse erreichen. Diese Testsequenzen modellieren bestimmte Szenarien, in denen die zu testende Klasse geprüft wird. Der Grad an erzielter Code-Überdeckung ist ein Maß für die Testabdeckung und damit der Testqualität generell.

Die existierenden Automatisierungsansätze beruhen hauptsächlich auf entweder symbolischer Ausführung und Constraint-Lösung oder auf einem Suchverfahren. Sie haben jedoch verschiedene Begrenzungen, die sowohl ihre Anwendbarkeit für unterschiedliche zu testende Klassen als auch ihre Effektivität im Hinblick auf die erreichbare Code-Überdeckung einschränken.

Die Ansätze basierend auf symbolischer Ausführung und Constraint-Lösung weisen die Beschränkungen dieser Techniken auf. Dies sind beispielsweise Einschränkungen hinsichtlich der Skalierbarkeit und bei der Verwendung bestimmter Programmierkonstrukte wie Schleifen, Felder und komplexer Prädikate. Die suchbasierten Ansätze haben Schwierigkeiten bei komplexen Prädikaten und komplexen Methodenaufrufabhängigkeiten. Die Ansätze adressieren weder den Test nicht-öffentlicher Methoden, ohne die Objektkapselung zu verletzen, noch die Behandlung von Laufzeitausnahmen während der Testgenerierung. Objektkapselung, nicht-öffentliche Methoden und Laufzeitausnahmen sind jedoch grundlegende Konzepte objektorientierter Software, die besonderes Augenmerk während des Tests erfordern.

Die vorliegende Dissertation schlägt einen neuen Ansatz zur automatischen Generierung objektorientierter Unit-Tests vor. Dieser Ansatz verwendet Genetische Programmierung, ein neuartiges meta-heuristisches Optimierungsverfahren. Dadurch kann die Testsequenz-Generierung geeigneter als Suchproblem formuliert werden als es die existierenden Ansätze gestatten. Effektivere Suchen nach Testsequenzen zur Erreichung von hoher Code-Überdeckung werden so ermöglicht. Der Ansatz umfasst außerdem den Test nicht-öffentlicher Methoden ohne Kapselungsbruch und berücksichtigt Laufzeitausnahmen, indem er die für die Suche verwendeten Zielfunktionen adequat definiert.

Eine umfangreiche Fallstudie demonstriert die Effektivität des Ansatzes. Die dabei verwendeten Klassen besitzen nicht-öffentliche Methoden und führen in zahlreichen Fällen zu Laufzeitausnahmen während der Testgenerierung. Die erreichten Code-Überdeckungen werden den Ergebnissen eines Zufallsgenerators sowie zweier kommerzieller Testsequenz-Generatoren gegenübergestellt. In der Mehrheit der Fälle übertraf der hier vorgeschlagene Ansatz die alternativen Generatoren.





## Declaration

The work presented in this thesis is original work undertaken between October 2004 and September 2007 at the DaimlerChrysler AG, Research and Technology, Software Technology Lab, and the Technical University of Berlin (DaimlerChrysler Automotive IT Institute). Portions of this work have been published elsewhere:

- S. Wappler, F. Lammermann, Using Evolutionary Algorithms for the Unit Testing of Object-Oriented Software, In *GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, pages 1053-1060, Washington, D.C., USA, ACM Press, 2005
- S. Wappler, J. Wegener, Evolutionary Unit Testing of Object-Oriented Software Using Strongly-Typed Genetic Programming, In *GECCO '06: Proceedings of the 2006 Conference on Genetic and Evolutionary Computation*, pages 1925-1932, Seattle, WA, USA, ACM Press, 2006
- S. Wappler, J. Wegener, Evolutionary Unit Testing of Object-Oriented Software Using a Hybrid Evolutionary Algorithm, In *Proceedings of the IEEE World Congress on Computational Intelligence (WCCI-2006)*, pages 3227-3233, Vancouver, BC, Canada, IEEE Press, 2006
- S. Wappler, A. Baresel, J. Wegener, Improving Evolutionary Testing in the Presence of Function-Assigned Flags, In *Proceedings of Testing: Academic and Industrial Conference (TAIC PART)*, to appear, 2007
- S. Wappler, I. Schieferdecker, Improving Evolutionary Class Testing in the Presence of Non-Public Methods, In *Proceedings of the 2007 Conference on Automated Software Engineering (ASE)*, to appear, 2007



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Aims and Objectives . . . . .	3
1.2	Contributions . . . . .	4
1.3	Structure . . . . .	5
<b>2</b>	<b>Background and Related Work</b>	<b>7</b>
2.1	Structure-Oriented Class Testing . . . . .	7
2.1.1	Principles of Object-Oriented Software . . . . .	7
2.1.2	Software Testing in General . . . . .	9
2.1.3	Class Testing . . . . .	10
2.1.4	Structure-Oriented Testing Techniques . . . . .	11
2.2	Automatic Test Generation . . . . .	14
2.2.1	Static Test Generation . . . . .	15
2.2.2	Dynamic Test Generation . . . . .	20
2.2.3	Commercial Test Generators . . . . .	29
2.2.4	Limitations of the Existing Approaches . . . . .	31
2.3	Evolutionary Algorithms . . . . .	35
2.3.1	Evolutionary Algorithm Principles . . . . .	36
2.3.2	Genetic Algorithms . . . . .	44
2.3.3	Genetic Programming . . . . .	46
2.4	Summary . . . . .	51
<b>3</b>	<b>Evolutionary Class Testing</b>	<b>53</b>
3.1	Overview . . . . .	53
3.2	A Formal Consideration of Test Sequences . . . . .	54
3.3	Representation by Method Call Trees and Number Sequences . . . . .	56
3.3.1	The Method Call Dependence Graph . . . . .	57
3.3.2	Method Call Trees . . . . .	62
3.3.3	Primitive Arguments and Parameter Object Selectors . . . . .	66
3.3.4	Test-Sequence-Generating Algorithm TCGen1 . . . . .	69
3.4	Representation by Extended Method Call Trees . . . . .	72
3.4.1	Incorporating Parameter Space into Sequence Space . . . . .	72
3.4.2	Test-Sequence-Generating Algorithm TCGen2 . . . . .	74
3.5	Objective Function Construction . . . . .	76
3.5.1	Classification of Execution Flows . . . . .	77
3.5.2	Dynamic Test Sequence Infeasibility . . . . .	78

3.5.3	Endless Loops . . . . .	78
3.5.4	Unfavorably Evaluated Conditions . . . . .	79
3.5.5	Runtime Exceptions . . . . .	80
3.5.6	Non-Public Methods . . . . .	83
3.5.7	Putting it all Together . . . . .	84
3.6	Test Cluster Definition . . . . .	85
3.6.1	Mock Classes . . . . .	86
3.6.2	Interface Implementers and Abstract Class Implementers . . . . .	87
3.6.3	Array Generators . . . . .	88
3.7	Function-Assigned Flags . . . . .	88
3.7.1	Existing Approaches to Flag Removal . . . . .	90
3.7.2	Method Substitution . . . . .	92
3.7.3	Boolean Variable Substitution . . . . .	93
3.8	Summary . . . . .	98
<b>4</b>	<b>Experiments</b>	<b>103</b>
4.1	Implementation of EvoUnit . . . . .	103
4.2	General Effectiveness Case Study . . . . .	106
4.2.1	Test Objects . . . . .	106
4.2.2	Setup and Realization . . . . .	111
4.2.3	Results . . . . .	116
4.3	Non-Public Method Coverage Case Study . . . . .	125
4.3.1	Test Objects . . . . .	126
4.3.2	Setup and Realization . . . . .	126
4.3.3	Results . . . . .	126
4.4	Function-Assigned Flag Case Study . . . . .	127
4.4.1	Test Object . . . . .	127
4.4.2	Setup and Realization . . . . .	129
4.4.3	Results . . . . .	129
4.5	Summary . . . . .	130
<b>5</b>	<b>Conclusion and Future Work</b>	<b>133</b>
5.1	Summary of Achievements . . . . .	133
5.2	Restrictions and Limitations . . . . .	134
5.3	Summary of Future Work . . . . .	136
5.3.1	Addressing the Limitations . . . . .	137
5.3.2	Other directions . . . . .	138
	<b>Bibliography</b>	<b>141</b>
<b>A</b>	<b>Source Codes and Algorithms</b>	<b>147</b>
A.1	Source Listings . . . . .	147
A.2	Algorithms . . . . .	152
A.2.1	TCGen2 . . . . .	153

# List of Figures

2.1	Example control flow graph . . . . .	12
2.2	Example symbolic execution tree . . . . .	17
2.3	Execution flows of a simple function . . . . .	23
2.4	Example application of Tonella's crossover operator . . . . .	27
2.5	Decomposition of a test sequence . . . . .	28
2.6	Classification of evolutionary algorithms . . . . .	37
2.7	Evolutionary algorithm context . . . . .	38
2.8	Principle procedure of an evolutionary algorithm . . . . .	39
2.9	Stochastic universal sampling . . . . .	42
2.10	Simple program tree . . . . .	47
2.11	Subtree crossover . . . . .	49
2.12	ERC mutation . . . . .	50
2.13	Demotion mutation. . . . .	50
2.14	Promotion mutation. . . . .	51
3.1	Basic concept of evolutionary class testing . . . . .	53
3.2	Method call dependence graph . . . . .	60
3.3	Method call tree . . . . .	62
3.4	Method call dependence graph with additional call-contributing edges .	64
3.5	Method call tree containing state-changing methods; with annotated instances and their roles . . . . .	65
3.6	Method call tree, generated by loosened tree creation algorithm . . . . .	67
3.7	Method call dependence graph, augmented by primitive types . . . . .	73
3.8	Method call tree including parameter information . . . . .	75
3.9	Classification of test sequence executions . . . . .	77
3.10	Control flow graph including exceptional branches . . . . .	80
3.11	Objective functions for the different situations . . . . .	84
4.1	EvoUnit System Architecture . . . . .	104
4.2	Experimental ECJ pipeline . . . . .	112
4.3	Results for parameter number of individuals . . . . .	113
4.4	Results for parameter tournament size . . . . .	114
4.5	Coverage achieved by EvoUnit and random generator; J2SDK test objects	120
4.6	Coverage achieved by EvoUnit and random generator; Quilt test objects	120
4.7	Coverage achieved by EvoUnit and random generator; JFreeChart test objects . . . . .	121

4.8	Coverage achieved by EvoUnit and random generator; both Colt and Math test objects . . . . .	121
4.9	Coverage achieved by all generators; J2SDK test objects . . . . .	125
4.10	Coverage achieved by all generators; Quilt test objects . . . . .	126
4.11	Coverage achieved by all generators; JFreeChart test objects . . . . .	127
4.12	Coverage achieved by all generators; both Colt and Math test objects . .	128
4.13	Coverage of non-public methods . . . . .	129
4.14	Objective value development for transformed Stack . . . . .	130
5.1	Example class diagram . . . . .	137

# List of Tables

2.1	Distance functions . . . . .	24
2.2	Related approaches . . . . .	31
2.3	Limitations of the approaches . . . . .	36
2.4	Typed function set . . . . .	47
3.1	Example type set . . . . .	70
3.2	Example function set . . . . .	70
3.3	Extended type set . . . . .	75
3.4	Tactic 1 . . . . .	94
3.5	Tactic 2 . . . . .	95
3.6	Tactic 3 . . . . .	96
3.7	Arguments for <i>func1</i> and the resulting flag values . . . . .	97
3.8	Properties of evolutionary class testing with respect to the limitations . . . . .	100
4.1	Test objects; general complexity metrics . . . . .	108
4.2	Test objects; properties related to limitations . . . . .	109
4.3	Test objects; properties related to the evolutionary search . . . . .	110
4.4	Settings of the genetic programming system ECJ . . . . .	112
4.5	ERC value ranges . . . . .	114
4.6	Results from EvoUnit (optimizing mode) . . . . .	117
4.7	Results from EvoUnit (random mode) . . . . .	118
4.8	Clover results for EvoUnit and CodePro . . . . .	123
4.9	Clover results for EvoUnit and Jtest . . . . .	124





# Listings

2.1	Test sequence examining method equals of class IntegerRange . . . . .	11
2.2	Simple function sorting two integers . . . . .	16
2.3	Simple C function . . . . .	21
3.1	Linearized method call tree . . . . .	66
3.2	Test sequence, augmented by framework methods . . . . .	68
3.3	Linearized method call tree . . . . .	76
3.4	Statically feasible but dynamically infeasible test sequence . . . . .	78
3.5	Exceptional test sequence . . . . .	82
3.6	DatabaseAdapter class to be replaced . . . . .	86
3.7	DatabaseAdapter mock class . . . . .	86
3.8	Array generator for class Integer . . . . .	88
3.9	Example of function-assigned flag . . . . .	89
3.10	Problematic flag transformation . . . . .	90
3.11	Polymorphic stack types . . . . .	91
3.12	Original predicate . . . . .	92
3.13	Modified predicate . . . . .	93
5.1	Example pointer-comparing predicate . . . . .	138
A.1	Class Integer . . . . .	147
A.2	Class IntegerRange . . . . .	147
A.3	Class State1 . . . . .	150
A.4	Class Stack . . . . .	150
A.5	Class StackT . . . . .	151
A.6	Test-sequence-generating algorithm TCGen1 . . . . .	152
A.7	Test-sequence-generating algorithm TCGen2 . . . . .	153



# 1 Introduction

Creating relevant test cases is the most critical activity during software testing. The set of test cases with which the software under test will be examined must not only possess a good ability to reveal faults, but also be a representative and maintainable subset of all possible input situations. Both quality and significance of the overall test are directly affected by the set of test cases used during testing.

With object-orientation, testing on the unit level – the most elementary level – focuses on the examination of a single class. Classes are the atoms which, assembled together, constitute an object-oriented application. A test case for unit testing a class includes the information as to how to create an instance of the class under test, how to create other instances that are needed during the test (for instance to serve as arguments for operations), and which object states and results are expected when particular operations of the class under test are executed. This information is represented by a *test sequence* – a sequence of method calls which involve creating objects, putting the objects into proper states, and invoking the operations to be examined – and a *test evaluation*, consisting of one or several checks of the final state and the outputs.

Various techniques to derive relevant tests from different types of development artifacts have been proposed. One important category of testing techniques is *structure-oriented testing*. A structure-oriented testing technique utilizes the implementation (the source code) of the software under test to identify relevant tests. This type of testing technique is often applied to complement a *function-oriented testing* technique, which focuses on the coverage of the requirements. Since both types of testing techniques have different failure models in mind, their combination increases the quality of the overall test.

A structure-oriented testing technique employs a *code coverage criterion* to guide the identification of relevant tests. For instance, statement testing utilizes the criterion *statement coverage* and focuses on the statements of the software under test: tests are to be generated that lead to the execution of all (or a high number of) statements of the software under test. Faults related to the statements of the unit under test are expected to be exhibited by the tests generated this way. Other important criteria are *branch coverage* and *condition coverage*. Industrial quality standards demand that the tests applied to software of a particular application domain exceed a predefined code coverage rate. For instance, the avionics standard RTCA DO-178B (RTCA Inc., 1992) requires that for airborne software belonging to a high risk level the corresponding test cases satisfy decision coverage. Another example is the automotive standard ISO/WD 26262 (ISO, 2005). Depending on the risk level *ASIL* (automotive safety integrity level) to be attained, statement coverage, decision coverage, path coverage, condition coverage, or modified condition decision coverage must be maximized. The standard demands for full coverage, meaning that 100% code coverage must be achieved. However, it allows

deviating from full coverage in justified situations.

Although initially developed for testing procedural software, such as C or Ada modules, structure-oriented testing techniques are also effectively applied to testing object-oriented software. Recent investigations have shown that they are well-suited to create relevant tests for object-oriented class testing, and are advised to be applied in conjunction with other object-oriented testing techniques (Kim, Clark and McDermid, 1999; Kim, Clark and McDermid, 2000).

Software testing consumes up to half of the budget of a software development project (Beizer, 1990). A survey carried out by DaimlerChrysler confirms the findings of other companies: while 50% of the costs for a development project are spent for implementation activities, the remaining 50% are spent for testing purposes (Grochtmann, 2000). Unit testing and integration testing need 30% of the total budget. The process of creating relevant tests consumes significant resources in terms of time, human capacity, and thus costs. When done manually, it is also tedious and error-prone.

Several approaches exist that automate the creation of test sequences for object-oriented unit testing in order to benefit from reductions in time, labor, and budget. The structure-oriented approaches, which will be considered in this work, rely on either symbolic execution and constraint solving (King, 1976; Tsang, 1993), or on concrete execution and a search strategy. The former will be referred to as *static approaches*, while the latter will be referred to as *dynamic approaches*. More recent approaches combine aspects of the two categories. The common idea is to divide the source code to be covered by tests into individual components, referred to as *test goals* in the following. For instance, in the case of branch testing, each branch of the control flow graphs of the methods of the class under test is considered a test goal. An attempt is made to create a test sequence for each test goal. The static approaches apply symbolic execution, which emulates the actual execution of the software under test using symbolic inputs instead of concrete ones. Path conditions are thereby collected which formulate the requirements to be satisfied by the participating objects in order for the execution to cover the targeted test goal. Constraint solving then tries to compute a concrete accumulation of object states from the path conditions. In contrast, the dynamic approaches execute the software under test using concrete objects and inputs. A search strategy is employed to search the space of all possible test sequences for a covering one.

However, the existing approaches possess several limitations which diminish their value: symbolic execution suffers from the problem of state space explosion if the software under test is complex. When a huge set of symbolic states results from the structure of the software under test, memory and computation power may not suffice to maintain and examine these states with a practical performance. For instance, loops in the source code will result in an infinite set of symbolic states, if not appropriately bounded. Symbolic execution is also limited in the presence of polymorphism due to its static nature. Constraint solving suffers from the problem of non-linear and sometimes too complex constraints: today's constraint solvers are not able to compute a solution for any given collection of path constraints, in particular if the constraints contain severe non-linearities. Furthermore, most static approaches do not create the desired test

sequences, but rather in-memory representations of the objects participating in the tests. Such a representation must be transformed to a proper test sequence in order to be maintainable and insusceptible to class refactorings. However, the respective works do not propose an algorithm realizes such a transformation. Additionally, some static approaches are only applicable to classes whose methods have exclusively primitive argument types.

The dynamic approaches have deficiencies concerning both the effectiveness and efficiency of the search: (1) the incorporated search strategy may fail to find a test sequence which covers a test goal that is dependent on a complex condition, (2) the search is inefficient since it allows the generation of inexecutable test sequences, (3) the search requires detailed additional problem-specific, user-provided information to be effective. Furthermore, the dynamic approaches are limited in the presence of runtime exceptions: due to the random nature of the search of these approaches, implicit method preconditions might be violated, causing a runtime exception to be raised during a search. In this case, the search just terminates and does not deliver a result.

Many approaches also break the encapsulation of the classes under test. The generated tests are formulated so that the encapsulated data is accessed during test execution in order to put the objects into proper states. Doing so is critical since object states can be achieved which violate class invariants and hence contradict the specification of the classes. Using test sequences obtained by breaking the encapsulation questions the expressiveness of the overall test. No existing approach addresses directly testing non-public methods without breaking encapsulation.

## 1.1 Aims and Objectives

This thesis suggests a new approach to automatic test sequence generation for object-oriented class testing. Its main objective is to tackle the following limitations of the existing automation techniques in order to allow for broader applicability and improved effectiveness:

1. limitations of symbolic execution and constraint solving in general
2. limited applicability due to limited support for class type arguments
3. limited maintainability and usability of the generated results
4. inefficiency due to inexecutable test sequences
5. weaknesses in the presence of complex predicates
6. insufficient treatment of runtime exceptions
7. insufficient support of testing non-public methods

These limitations will be addressed by developing a new search-based automation approach which follows the ideas of *evolutionary structural testing*. Evolutionary structural

testing is a dynamic test generation technique that has been developed for testing procedural software. It employs evolutionary algorithms for the search for test data that maximize the code coverage of a procedure. Applying evolutionary algorithms eliminates the need to perform symbolic execution and constraint solving and hence overcomes the limitations inherent to both techniques (limitation 1).

An objective of this thesis is to enable the generation of test sequences that can create arbitrary objects that serve as arguments for succeeding method calls. This further allows the application of automatic test generation to classes that do not only possess method with primitive argument types (limitation 2).

An evolutionary algorithm requires both a suitable representation of candidate solutions (points in the search space) and an objective function that guides the search to be defined. An objective of this thesis is to develop a representation of test sequences that (a) relies on the public class interfaces only, and (b) defines a search space that contains preferably executable test sequences only in order to cope with both limitations 3 and 4.

Another objective is to design the objective functions used for the search so that they provide sufficient guidance also (a) in the presence of complex predicates controlling the test goal to be attained, (b) in the presence of undesired runtime exceptions which prematurely terminate the evaluation of a test sequence, and (c) in the case of a test goal that belongs to a non-public method. The strategy for objective function construction aims at treating limitations 5, 6, and 7.

The thesis exemplifies the automation of a particular type of decision testing. However, the approach is supposed to be also applicable to other structure-oriented techniques without great modification. The object-oriented concepts of the Java programming language (Gosling, Joy and Steele, 2005) are considered; the examples discussed in this thesis are classes and methods written in Java. Yet, the ideas of this thesis are expected to be applicable to testing software written in other object-oriented programming languages, albeit additional adaptation might be required.

## 1.2 Contributions

The contributions of this work are the following:

1. The investigation in the peculiarities of class testing, with particular regard to automatic test sequence generation;
2. The analysis of the state of the art of automatic test generation for class testing, along with the identification of deficiencies of current approaches;
3. The proposal of an approach to automatic test generation for class testing based on genetic programming, which consists of
  - the proposal of a representation of test sequences based on method call trees, which enables the use of an off-the-shelf genetic programming system for test sequence generation, and

- the proposal of a strategy for objective function design for decision testing which copes with complex predicates, runtime exceptions, and non-public methods;
4. The demonstration of the effectiveness of the approach in terms of achieved code coverage;
  5. The proposal of two strategies to improve the guidance to the evolutionary search in the presence of Boolean predicates;
  6. The demonstration of the effectiveness of these two strategies.

## 1.3 Structure

This thesis is organized as follows:

**Chapter 2 – Background and Related Work** lays the foundation of this work. It starts with an introduction to object-oriented class testing, including a short summary of the principles of object-orientation and the description of structure-oriented testing techniques. Afterwards, automatic test generation for class testing is discussed. Finally, evolutionary algorithms are detailed. Particular emphasis is given to genetic programming which is the key ingredient of the new approach.

**Chapter 3 – Evolutionary Class Testing** describes the new approach to automatic test generation for class testing in detail. First, it discusses the structure of test sequences in general. Then, two different representations of test sequences are suggested. The second representation is an extension of the first and simplifies the applied search algorithm significantly. Following this, the strategy for designing a suitable objective function for a given test goal is detailed. This includes a discussion of how to cope with runtime exceptions and non-public methods. An approach to handling non-instantiable classes is explained. Finally, the chapter discusses two strategies for improving the landscape of the objective functions in the presence of function-assigned flags, a frequently used code construct which sometimes hinders the evolutionary search.

**Chapter 4 – Experiments** reports on the results of three case studies which were performed to empirically assess the effectiveness of the approach. The first case study aims at demonstrating the effectiveness of the approach in terms of achieved code coverage in general. The coverage results obtained by the evolutionary class testing approach are contrasted with the results achieved by a random test sequence generator and two commercial generators. The second case study investigates the value of the objective functions for test goals belonging to non-public methods. It contrasts the results obtained by using the extended objective functions with the results obtained without using the extensions. The third case study evaluates one of the two strategies for objective landscape improvement.

**Chapter 5 – Conclusion and Future Work** summarizes the achievements of the thesis, points out the restrictions and limitations of the new approach, and gives directions for future research.



## 2 Background and Related Work

This chapter introduces structure-oriented unit testing of object-oriented software in Section 2.1 and reviews work in the field of automatic test generation in Section 2.2 on page 14. Evolutionary algorithms, the search technique on which this thesis builds, are presented in Section 2.3 on page 35. The basic concepts discussed here are key to the remainder of this thesis.

### 2.1 Structure-Oriented Class Testing

This section introduces structure-oriented unit testing of object-oriented software. It describes the technical scope of this thesis. First, Section 2.1.1 highlights the concepts of object-orientation. Next, Section 2.1.2 on page 9 gives an introduction to software testing in general, while Section 2.1.3 on page 10 elaborates on testing object-oriented software on the unit level in particular. Finally, Section 2.1.4 on page 11 discusses structure-oriented testing techniques in depth.

#### 2.1.1 Principles of Object-Oriented Software

According to Stroustrup (1988), a programming language is object-oriented if it provides full support for data abstraction, encapsulation, inheritance, polymorphism, and self-recursion. For example, C++ (Stroustrup, 2000) and Java (Gosling et al., 2005) are object-oriented programming languages. In contrast, the language C (ISO/IEC 9899, 1990), whose primary abstraction is a module control flow, is a procedural programming language (Binder, 1999). In the following, the mentioned object-oriented concepts will be explained in more detail, along with the description of the basic terminology.

#### Data Abstraction (Classes, Objects, and Interfaces)

An object-oriented application is a composition of interacting objects that communicate with each other by issuing function calls. An *object* is an instance of a class. At runtime of an application, more than one instance of the same class can exist. A *class* is an abstract data type. It assembles *attributes* and *methods*. Both attributes and methods are called class *members*. The attributes are variables that represent the state of an object. An attribute may be of primitive type, such as `integer` or `float`, or of a class or interface type. The methods are procedures that typically operate on the attributes. An *interface* is an abstract data type that consists of method declarations only; no implementations are assigned to the method declarations. A class can implement an interface by providing a method implementation for each method declaration of the

respective interface. An *abstract class* is a class, some or all of whose methods are not implemented, or that is explicitly declared as being abstract, respectively. An abstract class cannot be instantiated.

## Encapsulation

The attributes and methods of a class can be marked to be *visible* in certain contexts only. Visible means that, in case of an attribute, the value of the attribute can be read and written, and in case of a method that it can be invoked. Typically, an object-oriented programming language offers the visibility modifiers *public*, *protected*, and *private* (Java also offers the modifier *package*). A class member marked public is visible to all objects of the application, regardless which class declares it. A class member marked protected is only visible to objects of the class that declares it and objects of all subclasses of the declaring class (see Section 2.1.1 for subclassing; in some programming languages, for instance in Java, protected members are also visible to classes belonging to the same package). A class member marked private is only visible to objects of the class that declares it. A class member marked package is visible to the objects of all classes that belong to the same package. A package is a particular collection of classes.

Visibility is enforced by the compiler for the programming language. A programmer cannot write and compile code that accesses a private member from outside the class that declares that private member – the compiler denies compiling. However, some programming languages, such as Java, allow one to circumvent the visibility control mechanism, and thus to break encapsulation by providing an additional programming interface. Via this interface (in case of Java it is the Reflection API) non-public members can be accessed freely. Section 2.2.4 on page 33 discusses the implications of breaking encapsulation for software testing in more detail.

## Inheritance

Subclasses can be derived from a given class. A subclass possesses all members of the super class (the class from which it is derived) without the need to define these members itself. This mechanism is called inheritance. Usually, inheritance is used to realize some *specialization* of a class. A subclass may specify additional members and also may *override* inherited methods, if they are accessible. Overriding means to redefine the implementation of the method, hence possibly changing the behavior of that method.

## Polymorphism

Different kinds of polymorphism are integrated in a programming language. Polymorphism of object identifiers (variables) is the most significant kind for an object-oriented programming language. This polymorphism is the concept that allows a variable, which is declared to be of a particular class type, to refer to an object of a subclass of that class type. Whenever a member is accessed via the variable, the access is made on the actual class, which is not necessarily the declared class. The actual method to invoke

is identified at runtime. The mechanism of detecting the actual method to call during runtime is called *dynamic binding*. Polymorphism is usually restricted by inheritance, meaning that it applies to classes that belong to the same inheritance hierarchy. Other kinds of polymorphism are, for instance, the template concept in C++ or the overloading of operators.

### Self-Recursion

Self-Recursion is the ability of an object to refer to its own identity. This means, for instance, that a method of an object can call another method on the same object.

### 2.1.2 Software Testing in General

Testing is an important analytical quality assurance means in the area of software development. It is an integral part of the established process models for software development, such as the spiral model (Boehm, 1988). Its systematic application is required by industrial standards, e.g. ISO WD 26262. The primary intention of testing is to find faults in the software under test and to gain confidence in the correct implementation of the functionality if no faults are found. Testing is an execution-based technique meaning that the software under test will be executed. Thereby, the behavior of the system under test will be observed and evaluated.

A comprehensive and complete test requires the tested software to run in each possible scenario with any possible input. Since this is practically impossible (due to the combinatorial explosion caused by the typically huge input value ranges), testing also includes a sampling activity that selects *relevant* test inputs with which the test will be performed. This sampling activity, called *test case generation* or simply *test generation*, is crucial to software testing since it directly affects the quality of the overall test. Either the selection of the sample tests is poor, possibly involving redundancy or leaving gaps, in which case the overall test quality is also poor. Or the selection of tests covers a wide range of possible behaviors of the system under test, in which case the significance of the overall test as well as the fault-revealing potential is high.

Various approaches exist to guide the process of test generation. In general, one distinguishes between approaches based on the specification of the system under test (function-oriented testing, also called specification-based testing, or black-box testing), and approaches based on the implementation of the system under test (structure-oriented testing, also called implementation-based testing, or white-box testing). While function-oriented approaches guide the process of test generation by the semantics (the software specification), structure-oriented approaches guide it by the syntax (structural aspects of the implementation). As described in Chapter 1 on page 1, function-oriented and structure-oriented techniques complement one another since they are based on different fault models.

Testing takes place at different aggregation levels of the software. *Unit testing* is considered to be the most elementary level of testing. It addresses the examination of the “atoms” of the software under test. With regard to the paradigm of object-orientation,

these atoms are the classes, the instances of which the overall application is composed. Therefore, unit testing of object-oriented software is also referred to as *class-level testing*, or simply *class testing*. *Integration testing* applies to the level of compositions of atoms. Different combinations of these compositions are examined on this level. The focus is on the interaction of the elements of a composition. For the integration test of object-oriented systems, the single classes are integrated step by step in order to finally realize the intended application. At the system level, *system testing* examines the behavior of the overall application in conjunction with all peripheral components.

### 2.1.3 Class Testing

Class testing focuses on the examination of a single class. Due to the data dependencies among the methods (several methods access the same attributes) and data encapsulation, often a single method cannot be tested in isolation, rather the interplay of several methods is examined. For instance, a class test is intended to examine the correctness of method `equals` of class *C*; however, at the same time the constructor of class *C*, which is involved in the test since it creates an object for which to invoke method `equals`, is also tested. The method on which a class test focuses will be referred to as *method under test*.

Testing a particular class often involves other classes. For instance, the constructor of class *C* might require an instance of class *D* to be passed as an argument. Other methods might require instances of other classes as arguments. The entirety of classes needed to test a particular class *C* will be referred to as *test cluster* for *C*. The test cluster of *C* includes *C*. Attempts are made to minimize the “negative impacts” of the additional classes on the tests by using surrogate classes, for instance *mock classes* (Beck, 2003). A surrogate class is a replacement for a genuine class; while it possesses the same public interface, it might have completely different implementations of the methods. For instance, a complex class which requires particular resources to be available (such as database content or network resources) is often replaced by a mock class which mimics the behavior of the surrogated class but does not require its resources. Instead of delivering real database content, the methods of the mock class may return fixed, user-adjustable values. Another reason for using mock classes is to avoid a failure caused by an object of an associated class propagating to a failure of the primarily tested instance, thus making the localization of the fault difficult. In general, it is not reasonable to replace each class of the test cluster by a mock class. Therefore, unit testing is sometimes already integration testing.

An object-oriented unit test consists of a sequence of method calls that model a particular test scenario, and a sequence of assertions that checks whether or not the test is passed. The sequence of method calls will be referred to as *test sequence*, the sequence of assertion statements will be referred to as *test evaluation*.

A test sequence normally does not involve branching statements, such as `if` statements or `switch` statements, because a test sequence considers one particular scenario and does not allow alternatives: either the scenario is run through as expected, then the test passes, or the scenario is not run through as expected, then the test fails. This

thesis also assumes that a test sequence does not involve loop statements, such as `while`. However, a test sequence can formulate a loop as the repetition of a subsequence (that is, as an unrolled loop).

The test sequence shown in Listing 2.1 focuses on testing method `equals` of class `IntegerRange` (its source code is shown in Listing A.2). However, it indirectly tests the constructor and method `clone`, too. Statements 1 to 4 create the instances needed, whereas statement 5 calls the method under test.

Listing 2.1: Test sequence examining method `equals` of class `IntegerRange`

---

```

1 // test sequence
2 Integer i1 = new Integer( 0 );
3 Integer i2 = new Integer( 100000 );
4 IntegerRange ir1 = new IntegerRange( i1 , i2 );
5 IntegerRange ir2 = ir1.clone();
6 boolean result = ir1.equals( ir2 );
7
8 // test evaluation
9 assert( result == true );

```

---

Basically, a test sequence creates the objects necessary to execute the method under test by calling object-creating methods and puts the created objects into particular states by calling instance methods on them. The test sequence in Listing 2.1 does not include state-changing methods; the initial states of the objects already accommodate the objective of the test. In the example, at first two instances of class `Integer` are created. These instances are then passed on to the constructor of the class under test `IntegerRange`. Afterwards, method `clone` is called to create a copy of the `IntegerRange` instance. Finally, the equality of the genuine and the copy is checked. According to the test evaluation, the test only passes if the check delivers the `true` result.

#### 2.1.4 Structure-Oriented Testing Techniques

Structure-oriented testing techniques derive relevant tests from the implementation, that is the source code, of the unit under test. Various categories of structure-oriented testing techniques exist, such as control-flow-oriented techniques or data-flow-oriented techniques. This work focuses on control-flow-oriented techniques. Their characteristic is that they derive relevant tests from the *control flow graph* (Hecht, 1977) of the unit under test. The control flow graph is a graphical representation of all control flows that can occur in a function (procedural programming) or method (object-oriented programming). To simplify matters, both functions and methods will be referred to as functions in the following.

**Definition 2.1.1.** The *control flow graph*  $G$  of function  $f$  is a directed graph, defined by the tuple  $(N, E, s, x)$  where  $N$  is the set of nodes, each of which represents a basic block of function  $f$ ,  $E \subseteq (N \times N)$  is the set of edges (branches), each of which represents a possible transfer of control between two basic blocks,  $s \in N$  is the starting node, and  $x \in N$

is the exit node. Additionally, the following two restrictions hold:  $\forall n \in N : (n, s) \notin E$ , and  $\forall n \in N : (x, n) \notin E$ .

Figure 2.1 shows the control flow graph of function `func` from Listing 2.3. The

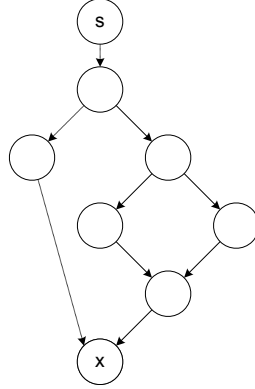


Figure 2.1: Example control flow graph

start node is labeled “s”, while the exit node is labeled “x”. A branching node (a node from which two branches originate) represents a conditional statement, while a normal node represents a basic block, that is, a series of sequentially executed statements. A conditional statement refers to a *predicate* which can be composed of several atomic *conditions*. Each conditional statement represents a *decision*.

The control flow graph of a function is the basis for various testing techniques. For instance, *branch testing* drives the generation of tests by the question which branches of the control flow graph are traversed during the execution of the tests. The technique generates tests with the intention of maximizing the number of traversed branches. *Branch coverage*, the ratio between the number of branches already covered by tests and the total number of branches, is an indicator for the adequacy and completeness of a given set of tests. Beizer (1990) discusses the various testing techniques in greater detail.

The following list gives a selection of common structure-oriented testing techniques along with both the underlying fault model and the related coverage criteria:

- *Statement testing* assumes that each statement of the unit under test may contain a fault. When executing each statement during testing the occurring failures reveal the faults related to the statements of the code (presuming that a fault actually propagates to an observable failure). Therefore, statement testing aims at maximizing the number of statements executed during testing. Statement coverage (also referred to as  $C_0$  coverage) indicates test adequacy and completeness for statement testing. It is defined as the ratio between the number of all statements executed during the execution of all tests and the number of all statements of the software under test.
- *Branch testing* assumes that each branch of the control flow graph of the unit under test may contain a fault. When traversing each branch during testing the

occurring failures reveal the faults related to the transfer of control of the code (presuming that a fault actually propagates to an observable failure). Therefore, branch testing aims at maximizing the number of branches traversed during testing. Branch coverage (also referred to as  $C_1$  coverage) indicates test adequacy and completeness for branch testing. It is defined as the ratio between the number of branches traversed during the execution of all tests and the total number of branches of the respective control flow graph.

- *Decision testing* is very similar to branch testing. The only difference is that decision testing takes only those branches of the control flow graph into account that start at branching nodes. Other branches, such as those connecting the start node with the first basic block node, are not considered.
- *Path testing* assumes that each path through the control flow graph of the unit under test may contain a fault. When traversing each path during testing the occurring failures reveal the faults related to the control flow paths (presuming that a fault actually propagates to an observable failure). Therefore, path testing aims at maximizing the number of program paths traversed during testing. Path coverage indicates test adequacy and completeness for path testing. It is defined as the ratio between the number of paths traversed during the execution of all tests and the total number of paths of the respective control flow graph.
- *Condition testing* assumes that each predicate of the unit under test may contain faults. When evaluating various combinations of the atomic conditions of a predicate during testing, the occurring failures reveal the faults related to the predicates in the code under test. Several versions of condition testing exist, each of which focuses on different combinations of the atomic conditions of a predicate. An important version is modified condition/decision testing.

Although the code-coverage-based testing techniques were originally designed for testing procedural software, their applicability to testing object-oriented software is widely accepted. Thorough investigations into the suitability of these techniques to object-oriented testing, such as Kim, Clark and McDermid (2001) or Kim et al. (2000), suggest their effectiveness and advise their use in combination with other, specifically object-oriented, techniques.

The code coverage criteria listed above apply to a single function and not to a whole class. In order to allow one to make statements concerning code coverage on the class level, this thesis suggests the application of the metric *method/decision coverage*, which has been developed during the research of this thesis. It combines the techniques of decision testing and *method testing*. Method testing assumes that each method of the class under test may contain a fault. When executing each method during testing, the occurring failures reveal the faults in the methods. Therefore, method testing aims at maximizing the number of methods called during testing. Method coverage indicates test adequacy and completeness for method testing. It is defined as the ratio between

the number of methods executed during the tests and the total number of methods (both public and non-public).

Method/decision coverage is defined as follows:

**Definition 2.1.2.** Let  $d_c$  be the number of decisions that occur in the source code of class  $c$ . Additionally, let  $s_c$  be the number of methods of  $c$  whose implementation is free of decisions, meaning that it consists of a sequence of statements only. Furthermore, let  $S$  be the set of test cases that are executed during testing. Let  $d_{c,S}^{true}$  be the number of decisions evaluated to **true** during test case execution at least once, and  $d_{c,S}^{false}$  be the number of decisions evaluated to **false** during test case execution at least once. Finally, let  $s_{c,S}$  be the number of decision-free methods entered during the execution of the test cases in  $S$ . Then, **method/decision coverage**  $D^+(c, S)$  for class  $c$  achieved by test suite  $S$  is defined as follows:

$$D^+(c, S) = \frac{d_{c,S}^{true} + d_{c,S}^{false} + s_{c,S}}{2d_c + s_c} \quad (2.1)$$

Method/decision coverage accumulates the decision coverage of the single methods of a class. However, in addition it also accounts for methods that do not possess any predicates. It combines the fault models behind both decision coverage and method coverage.

## 2.2 Automatic Test Generation

When accomplished manually, the process of test generation is tedious, error-prone and costly. The literature states that between 30% and 70% of a software project's budget is spent on testing (for instance, Beizer (1990) reports that 50% of the costs are typically spent for testing). Furthermore, extensive testing can only be accomplished by effective test automation (Staknis, 1990). The benefits of test automation are reductions in time, manual labor, and cost.

Various approaches to automatic test sequence generation for structure-oriented class testing have been proposed. They aim at generating a set of test sequences that achieve high structural coverage of the source code of the class under test. They usually build on the traditional test automation techniques for procedural software and extend them to the field of object-oriented software. The approaches are either *static* or *dynamic*. Static approaches do not execute the unit under test for test generation; rather, they compute suitable tests from the program logic using symbolic execution and constraint solving. Section 2.2.1 on the next page describes the static approaches to automatic test generation for class testing, including a short explanation of symbolic execution and constraint solving. Dynamic approaches execute the unit under test for test generation. They transform the task of test generation to a set of search problems where the search space is the set of all possible tests. A search strategy is then applied to find covering tests. The unit under test is executed with a usually large set of tests before a covering test will be encountered. Section 2.2.2 on page 20 describes the dynamic approaches in



more detail. Section 2.2.3 on page 29 presents three commercial test generators. Due to the lack of information as to which technology they rely on, a categorization according to static or dynamic appeared not to be definitively justifiable. Therefore, this extra section is introduced. Section 2.2.4 on page 31 generalizes the limitations of the approaches and gives a summary.

### 2.2.1 Static Test Generation

The static approaches do not execute any test sequence for obtaining a covering one; rather, they try to compute it. In order to do so, symbolic execution – a form of abstract interpretation – together with constraint solving is applied. Since all static approaches rely on symbolic execution and constraint solving, these techniques will be described first, followed by the description of the individual static approaches.

#### Symbolic Execution and Constraint Solving

Symbolic execution is a static analysis technique. Its application to software testing was pioneered by King (1976). The main idea of symbolic execution of a given program is to exercise the program with abstract (symbolic) inputs rather than concrete ones. All computations of the program affecting the inputs are not resolved to concrete results, but are rather kept on an abstract level by using symbolic expressions. This implies the program under consideration is not actually executed, its execution is rather “simulated” step by step. After each step, the program is in a new *symbolic state*. If a branching statement is encountered, each of the possible branches is visited according to the chosen strategy (depth-first, breadth-first, or others). Typically two new symbolic states result from a branching statement. A symbolic state represents a concrete statement along with a concrete path to that statement. For each symbolic state, symbolic execution delivers a set of constraints (referred to as the *constraint system*) which a concrete input must satisfy in order for the path to the statement, represented by the symbolic state, to be traversed.

The symbolic execution of a program can be visualized using a *symbolic execution tree*. The nodes of the tree represent symbolic states while the links between the nodes represent possible transitions. A symbolic state consists of the relevant symbolic inputs, a *path condition* (PC), and a program counter, respectively. The path condition is a Boolean expression applicable to the relevant symbolic inputs. It accumulates the constraints that must be satisfied in order for the symbolic state to be reached. The program counter is the reference to the statement to be executed next. The following example shall clarify the working of symbolic execution. It is taken from Khurshid, Păsăreanu and Visser (2003). Listing 2.2 shows the source code of a function that sorts the inputs  $x$  and  $y$ ; it ensures that, after the execution of it,  $x$  is smaller than  $y$  (overflows should be neglected). Figure 2.2 on page 17 shows the corresponding symbolic execution tree. The root node of the tree is the initial symbolic state (denoted state 1). It shows that  $x$  and  $y$  are assigned the symbolic values  $X$  and  $Y$ , respectively. The path condition is initially true meaning that this state is reachable without any constraints. Since the

Listing 2.2: Simple function sorting two integers

---

```

void sort(int x, int y)
{
    if ( x > y )
    {
        x = x + y;
        y = x - y;
        x = x - y;
        if ( x - y > 0 )
            assert( false );
    }
}

```

---

first statement of function `sort` is a decision, two distinct subsequent symbolic states are achievable (states 2 and 3). Either, the true branch of the decision is followed (state 2); then the predicate of the condition is incorporated into the path condition as shown in the left child of the root node. Or, the execution follows the false branch (state 3); then, the inversion of the predicate is added to the path condition as shown in the right child of the root node. In the former case, symbolic execution considers the subsequent assignment statements (states 4 to 6). While the path conditions remain unchanged during the assignments, the symbolic values for `x` and `y` are adapted accordingly. The final decision leads to a branch in the symbolic execution tree and the corresponding new symbolic states (states 7 and 8) with the accumulated path conditions. Note that during constraint solving, which might occur simultaneously or after symbolic execution, it would turn out that the symbolic state 7 is infeasible due to the contradictory path condition that evaluates to false.

Once the constraint systems are acquired for each relevant program element to cover, a constraint solver tries to obtain the concrete inputs for each of the paths in order to generate a test set with high code coverage.

### Automated Testing of Classes

Buy, Orso and Pezze (2000) suggest an approach to generating test sequences based on symbolic execution and automated deduction. Their work is concerned with the data-flow-oriented coverage criterion *all def-use pairs*. This criterion demands that the test sequences involve the assignment of each program variable (the *def*), followed by a reference to the respective variable (the *use*) without an intermediate reassignment. The approach consists of 3 steps:

*Step 1: Data flow analysis.* This analysis aims at collecting all def-use pairs present in the code of the class under test. A def-use pair is a pair of statements that relate to each other in that the one statement defines a particular variable (writes the value of it), while the other uses the same variable (reads the value of it); no redefinition of the variable is allowed to occur between the considered definition and the use. Since the

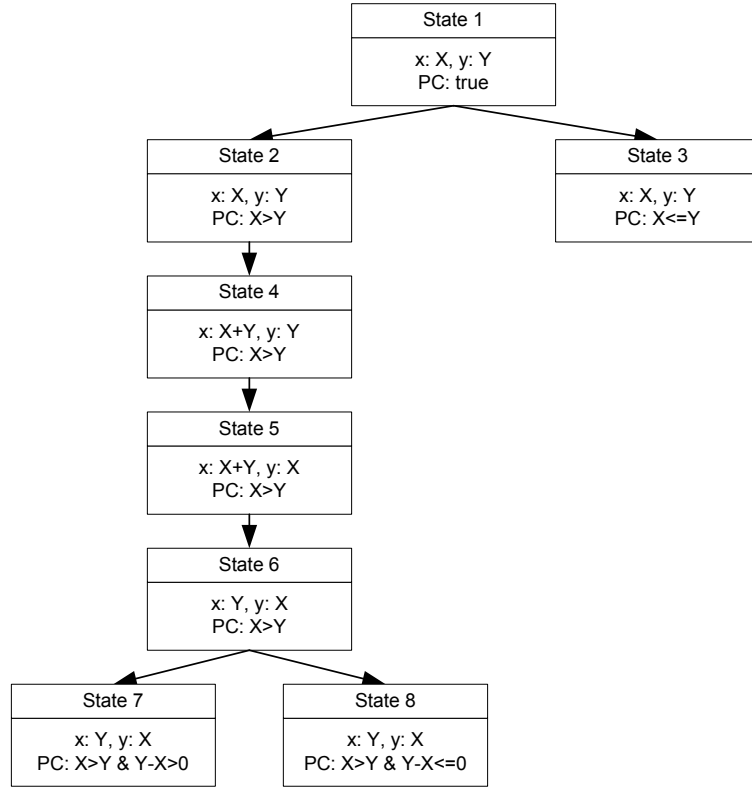


Figure 2.2: Example symbolic execution tree

analysis is applied to the whole class, a def-use pair can relate to statements that belong to different methods.

*Step 2: Symbolic execution.* This step obtains the possible paths through the methods of the class under test, including the predicates to be satisfied for a particular path to be taken during execution. For each path, symbolic execution analyzes the relations between the inputs and the outputs in an abstract (symbolic) fashion. These relations are interpreted as method preconditions and postconditions.

*Step 3: Automated deduction.* During this step, test sequences are incrementally built in order to execute the methods of the class under test so that a particular def-use pair is covered without violating the requirement that a definition-clear path is taken between the two code points. Automated deduction starts with method  $m_u$  that contains the statement involving the use of a particular variable and puts this method as initial element into the test sequence to be built (resulting in  $\langle m_u \rangle$ ). Then, all methods satisfying the preconditions of  $m_u$  are considered. If there are none, the def-use pair is deemed to be infeasible. If there are multiple candidate methods, the approach starts building a tree of method sequences. Tree building finishes once a constructor is inserted or a predefined size limit is reached. In the first case, a feasible covering test sequence has been found.

The authors consider primitive instance variables only; they do not define what a definition and a use of a class-type variable is. The example provided in their paper involves methods with empty formal parameter lists only. Furthermore, the approach addresses public methods only. The authors state that both symbolic execution and automated deduction involve complex computation, making the approach expensive and not scale well.

### Concolic Testing

Sen, Marinov and Agha (2005) propose a test generation technique that combines symbolic execution with concrete execution. They call this strategy *concolic testing* (concolic = *concrete* + *symbolic*). Their early works are on concolic testing of procedural software, while the later works also deal with object-oriented programs, in particular with Java classes (Sen and Agha, 2006). It is classified as a static approach in this thesis, because it primarily involves symbolic execution and constraint solving. However, it also incorporates aspects of dynamic test generation.

The motivation behind concolic testing is that in practice, the path conditions of the symbolic states can grow very complex, hence being not solvable by contemporary constraint solvers. Therefore, the method under test is primarily executed using concrete input values. These inputs are generated randomly or are provided by the user. During concrete execution, the symbolic path conditions are collected for the traversed path. Then, by systematically modifying the (symbolic) path conditions (e.g. by negating part of the conjuncts) and solving the resulting constraints, new concrete input values are obtained. These new inputs are likely to take an alternative path through the program. By doing so repeatedly, eventually a high number of possible paths might be detected for which the corresponding concrete input values are identified simultaneously. Also, if the symbolic path constraints become too complex during concrete execution, parts of it are replaced by the current concrete values.

For pointer variables, *memory graphs* are used that represent dynamic data structures (such as objects) including their associations. Path constraints referring to pointer variables are maintained separately from those referring to primitive variables. *Logical input maps* are used to keep memory addresses abstract (logical) and to allow for symbolic execution of pointer accesses.

A limitation of the concolic testing approach is that the constraint solver might still be not powerful enough (Sen and Agha, 2006). Therefore, a requirement for the class to be tested is that the number and lengths of the paths through a method is finite (which practically means that neither loops nor recursion may be involved). The description of the approach lacks an algorithm that transforms an obtained memory graph satisfying an obtained constraint system to a method call sequence. This means the publications do not describe how to construct the concrete objects that satisfy the symbolic constraints via the public interfaces of the involved classes. Rather, the approach seems to presume that all object attributes can be freely accessed, hence neglecting data encapsulation. The work does not discuss how legality of the instances is ensured. It does not address testing non-public methods either.

### Java PathFinder

Visser, Păsăreanu and Khurshid (2004) present a testing framework based on a Java model checker called Java PathFinder. They transform the task of creating a test that leads to the coverage of a particular code element to a model checking task. Model checking in this context is essentially equivalent to symbolic execution and constraint solving. Thereby, it is formulated as a model property that the code element in question is not reachable. Then the model checker tries to provide a counterexample by trying to reach the symbolic state representing the code element to cover. If the symbolic state is reached, the corresponding constraint system defines an adequate, covering test as an object graph (not as a method call sequence). The authors do not discuss how to obtain a method call sequence; rather, they consider single methods which they model check.

The authors introduce the notion of *lazy initialization* which means that the constraint system does not necessarily refer to a complete instance of a class: constraints do not necessarily exist for all object attributes. Later in the process, new constraints may refer to unreferenced attributes, making the consideration of these attributes necessary, especially when the attribute at hand has a class type. For the symbolic initialization of newly accessed class-type attributes, the authors suggest a heuristic based on random choice: either, the attribute is initialized to *null*, or it is initialized to a new instance of the class with uninitialized attributes, or a reference to an already created object is reused. This heuristic is intended to systematically tread pointer aliasing.

Additionally, the work deals with a facility for symbolically executing method preconditions in order to restrict object instantiations to legal ones. When solving the constraint system for a particular path, optionally provided method preconditions are executed symbolically in order to initialize the instances with reasonable attribute values.

The work does not include an algorithm to translate the obtained object graph to a test sequence which creates the required instances satisfying all the constraints (Xie, Marinov, Schulte and Notkin, 2005) of the associated constraint system. Data encapsulation is broken since all attributes are written and read freely, regardless of whether or not they are public. However, this is not critical presuming that formal class invariants are also provided by the user. In experiments, the authors found that the approach does not scale well and is not good in achieving high structural coverage.

### Symstra

Xie et al. (2005) propose a testing framework called Symstra. It is based on exhaustive method sequence exploration and symbolic execution. All conceivable method sequences derived from the class under test are explored up to a predefined length. In order to acquire concrete primitive arguments for the methods of a sequence that covers a particular code element, symbolic execution of that method sequence is carried out. Once a path to the symbolic state representing the code element in question is detected, a constraint solver is employed to find suitable concrete primitive argument values.

The approach can handle public methods that take primitive arguments. As the authors state, the approach cannot directly transform non-primitive arguments into

symbolic variables of primitive type. The legality of the considered method call sequences is ensured using additionally provided formal specifications (method preconditions and postconditions). However, the exhaustive exploration of the space of all method sequences is an expensive process.

### 2.2.2 Dynamic Test Generation

In contrast to the static approaches, where the methods of the class under test are not actually executed but only symbolically, the dynamic approaches involve concrete execution of candidate test sequences in order to obtain a covering one. A solution is not systematically constructed, but sought using a search technique.

The idea of dynamic test input generation dates back to the work of Miller and Spooner (1976) which deals with the dynamic generation of floating point test data. Later, Korel (1990) used the alternating variable search technique to obtain test data for structure-oriented testing of procedural software in general, not only for floating point inputs. The main motivation for dynamic test generation is to overcome the limitations of symbolic execution and constraint solving Korel (1990).

The next section recapitulates the history of dynamic test generation. The development of dynamic test generation techniques culminates in evolutionary structural testing, a highly developed approach to dynamic test generation that applies evolutionary algorithms as a search technique (cf. Section 2.3 on page 35). The section presents state of the art evolutionary structural testing of procedural software, before the next sections describe the dynamic approaches to automatic test generation for class testing.

### Evolutionary Structural Testing

In 1990, Korel (1990) suggested the *dynamic approach* to automatic software test data generation in order to cope with the limitations of the existing static approaches based on symbolic execution and constraint solving. The main idea of Korel's approach is to transform the task of creating a set of test inputs which achieve high path coverage to a set of search problems. For each path to be covered, a concrete test input is searched: the input space of the function under test, defined by the data type ranges of its arguments and possible other inputs, is heuristically explored by a trial-and-error strategy. Korel starts with a randomly created input. The function under test is executed with the input and the execution is monitored. For monitoring, the tested function is instrumented, meaning that additional trace statements are inserted which allow the comprehension of the details of the execution. A *cost function* (that is, an objective function) expresses to what extent the execution path taken by the input deviates from the targeted program path. Then, a new – and hopefully more suitable – input is created via the *alternating variable method*. By iteratively applying this method, finally a covering test input is supposed to be found.

Other researcher adopted Korel's approach to address other structure-oriented testing techniques, such as branch testing. Furthermore, other search strategies, such as genetic algorithms, were applied instead of the alternating variable method. For instance,

Sthamer (1996) and Jones, Sthamer and Eyres (1996) apply genetic algorithms to find test inputs that cover a given program path. A genetic algorithm is a meta-heuristic optimization technique; it is described in detail in Section 2.3 on page 35. It performs parallel searches that are guided by an *objective function*. This function assigns a quantitative rating to each candidate solution which expresses the *fitness* of the solution. Tracey, Clark, Mander and McDermid (1998b) modify the approach of Jones et al. (1996) by introducing additional distance functions for conditions which involve logical operators, such as *AND*, *OR*, and *NOT* in order to yield better objective functions; furthermore, they apply simulated annealing (Kirkpatrick, Gellat and Vecchi, 1983), another heuristic search technique. Wegener, Baresel and Sthamer (2001) extend the dynamic approach further in order to attack the limitation of the previous approaches that a particular program path must be selected which leads to the code element to cover. They suggest an objective function which is composed of two distance metrics. This objective function guides the search for covering test inputs irrespective of the path to be taken to the targeted code element. Genetic algorithms are used to carry out the searches. Their approach, which can be considered as the state of the art of evolutionary structural testing for procedural software, will be described in more detail in the following. Worthy of mention as other pioneering works in the area of evolutionary structural testing are those of Xanthakis, Skourlas and LeGall (1992), Pargas, Harrold and Peck (1999), and Michael, McGraw and Schatz (2001).

The application of an evolutionary algorithm as a search technique requires the definition of the search space and what a point in this search space is. With evolutionary structural testing, such a point is a test input used to execute the function under test. The *representation* defines how a concrete test input is encoded by a data structure that an evolutionary algorithm is able to operate on. In addition, an objective function is required to apply an evolutionary algorithm. This function assesses a generated candidate solution according to its ability to cover a given code element. Section 2.3 on page 35 provides details on the terminology and concepts of representations and objective functions.

The phenotype search space  $\Phi$  is the space of all value vectors that comply with the interface of the function under test.

Listing 2.3: Simple C function

---

```

1 int func(int a, int b, double c)
2 {
3     int local;
4     if( a == 0 )
5     {
6         local = read_integer();
7         if( local == b )
8             return round(c);
9         else
10            return -round(c);
11    }
```

```

12     else
13         return round(a*c);
14 }

```

For instance, for the function shown in Listing 2.3,  $\Phi = D_{int} \times D_{int} \times D_{double} \times D_{int}$  where  $D_{int}$  is the value range of type `integer` and  $D_{double}$  is the value range of type `double`. These value ranges correspond to the four input variables of the function (note the input variable in line 6). In order to limit the search to semantically reasonable inputs only, the user can provide more restrictive value ranges. With evolutionary structural testing, phenotype search space and genotype search space are conceptually identical. This is possible since suitable variation operators exist for each primitive data type of a procedural programming language such as C. Structured data, for example `structs` and `unions`, are decomposed into their building blocks. The task of the decoder (cf. Section 2.3.1 on page 36) is then to construct the respective data structures from a sequence of primitive values.

The overall task to obtain a set of test data which maximizes the given coverage criterion is divided into subtasks. For instance, with branch coverage, each branch becomes a test goal for which an individual evolutionary search is carried out. Hence, each test goal requires an individual objective function to be defined that is particularly tailored to the test goal. However, the construction of the objective functions for the test goals of the function under test can be automated. Different types of code coverage criteria require different strategies when considering how to construct a suitable objective function. Baresel, Sthamer and Schmidt (2002) describes the strategies for various control-flow-oriented criteria, such as branch coverage, and data-flow-oriented criteria. In the following, the strategy for branch coverage is described since it is similar to the criterion method/decision coverage for which this work will later exemplify evolutionary class testing in Chapter 3 on page 53.

An objective function, as suggested by Wegener et al. (2001), consists of two distance metrics which express how close the execution of the function under test with a concrete input is to reaching the targeted test goal. These two distance metrics are *approximation level* and *branch distance*. The former will be referred to as *control dependence distance* in this work for reasons of consistency (the “approximation” is in fact a distance). Before defining these two metrics, the concepts of *critical branches* and *critical nodes* must be introduced.

**Definition 2.2.1.** *A branch  $c$  of the control flow graph of the function under test is called **critical branch** with respect to a particular branch  $t$  if no path exists between  $c$  and  $t$ . Node  $p(c)$  is called a **critical node**, where function  $p$  assigns each branch its source node (from which the branch starts).*

In other words, this means that, once a critical branch is taken during execution, it is not possible to reach the target branch any more.

**Definition 2.2.2.** *Let  $t$  be the targeted branch and  $c$  the first critical branch that execution diverged down. Then  $n_p = p(c)$  is called **problem node**. Let  $P_{n_p, t}$  be the*



set of all paths from problem node  $n_p$  to target branch  $t$ . Additionally, let  $\chi(\pi)$  be the number of critical nodes of path  $\pi$ . Then, the **control dependence distance**  $d_C$  is the minimum number of critical nodes that lay on a path between the problem node and the target:

$$d_C = \min(\{\chi(\pi) | \pi \in P_{n_p, t}\}) \quad (2.2)$$

Figure 2.3 shows on the left a control flow graph of the function from Listing 2.3, including the path provoked by an example input (depicted by the nodes in gray and the dotted branches). Whereas on the right, the control flow graph of the same function but with a different path, provoked by another input, is shown. Neither of the inputs

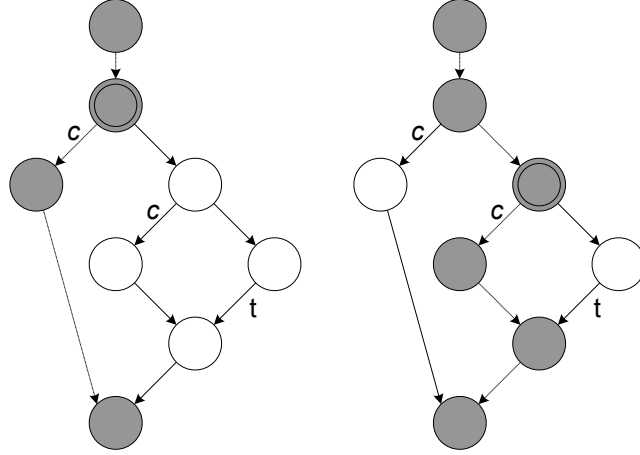


Figure 2.3: Two execution flows of the function from Listing 2.3

leads to the coverage of target branch  $t$ . The value of  $d_C$  for the left execution flow is 1, which is the minimum number of critical nodes of all paths from the problem node (the double-line node) to branch  $t$ . In the case of the right execution flow,  $d_C = 0$  as there is no critical node on the way from the problem node to the target.

The other metric, branch distance, is relevant if two different inputs yield the same execution path. In this case, the values of  $d_C$  are the same. However, one of the inputs might be closer to reaching the target in terms of the predicate assigned to the problem node. For instance, assume that two test inputs, input A and input B, lead to the execution path shown on the left in Figure 2.3. Additionally, assume input A leads to the concrete predicate  $(1 == 0)$  at the problem node, and input B leads to the concrete predicate  $(100 == 0)$ . Intuitively, input A is “closer” to evaluating the first condition so that the **true** branch will be traversed, which is favorable when targeting branch  $t$ . The metric branch distance formalizes the distance of the execution in terms of the predicate assigned to the problem node.

**Definition 2.2.3.** Let  $\mathbb{P}$  be the set of all predicates and  $\mathbb{B} = \{true, false\}$ . **Branch distance**  $d_B(p, b)$ , where  $p \in \mathbb{P}$  is the predicate in question and  $b \in \mathbb{B}$  is the desired

outcome (desired with respect to the target), is defined as follows:

$$d_B(p, b) = \begin{cases} 0 & \text{if } E(d) = b \\ d_p & \text{otherwise} \end{cases} \quad (2.3)$$

where  $E(p)$  with  $E : \mathbb{P} \rightarrow \mathbb{B}$  is the evaluated outcome of decision  $p$ , and  $d_p$  is the relation-specific distance function for  $p$ .

For each relational operator, such as  $<$ ,  $>$ , a specific distance function  $d_p$  is defined which expresses how distant the evaluation of the predicate  $p$  was to being evaluated to the alternative outcome. For instance, in the case of the predicate (  $a == 0$  ), the distance function is  $d_{a==0} = |a - 0|$ , mapped into the interval  $[0, 1)$ . Hence, the distance for an input which leads to a small value of  $a$  (and is thus closer to satisfying the condition than a large value of  $a$ ), is also small. The mapping into the interval  $[0, 1)$  ensures that the greatest possible distance is smaller than the smallest possible value of the control dependence distance.

Table 2.1 shows the generic distance functions that are typically applied. The value range of all distance functions is  $[0, 1)$ . The table shows in the first column the names of

	<b>true</b> desired	<b>false</b> desired
$d_{x==y}$	$1 - (1 + \varepsilon)^{- x-y }$	$d_{x \neq y}$
$d_{x < y}$	$1 - (1 + \varepsilon)^{y-x}(1 - \kappa)$	$d_{x \geq y}$
$d_{x \leq y}$	$1 - (1 + \varepsilon)^{y-x}$	$d_{x > y}$
$d_{x > y}$	$d_{y < x}$	$d_{x \leq y}$
$d_{x \geq y}$	$d_{y \leq x}$	$d_{x < y}$
$d_{x \neq y}$	1	$d_{x == y}$
$d_{e_1 \wedge e_2}$	$\max(d_{e_1}, d_{e_2})$	$\frac{d_{e_1} + d_{e_2}}{2}$
$d_{e_1 \vee e_2}$	$\frac{d_{e_1} + d_{e_2}}{2}$	$\max(d_{e_1}, d_{e_2})$
$d_{\neg e}$	$(d_e, \mathbf{false})$	$(d_e, \mathbf{true})$

Table 2.1: Distance functions

the distance functions for the relational and logical operators. In the second column, it shows the definition of the respective distance function, if the desired outcome of the predicate is **true**. Analogously, the third column shows the definition of the respective distance function, if the desired outcome of the predicate is **false**. Which outcome is desired depends on the location of the target branch.  $\varepsilon \in (0, 1)$  is a configurable parameter, and  $\kappa$  refers to the smallest possible value of the operands' data types. The definitions of the last row mean that the distance function for the opposite outcome is to be applied.

In conclusion, the objective function  $\omega_t(i)$  for a test goal  $t$  and the individual (=input)  $i$  is defined as follows:

$$\omega_t(i) = d_C + d_B \quad (2.4)$$

where  $d_C$  and  $d_B$  are the metrics control dependence distance and branch distance with respect to the problem node caused by the execution of input  $i$ . In the case of the example function above, the metric values for test input A, leading to the concrete predicate (  $1 == 0$  ) and consequently to a miss of the target branch  $t$ , are  $d_C = 1$  and  $d_B \approx 0.005$  (with  $\varepsilon = 0.005$ ). Then, the objective value  $\omega_t(A) = 1.005$ .

The following sections describe the search-based approaches in the field of automatic test generation for class testing. While the first approach relies on a binary search strategy, the latter two apply genetic algorithms.

## BINTEST

Beydeda and Gruhn (2003) propose a test generation approach based on a binary search strategy which they call *BINTEST*. The authors modified the test data generation approach of Korel (1990) by replacing the alternating variable search with a binary search. They consider the attributes of the class under test to be additional inputs to the method under test besides its regular arguments. Hence, they do not generate test sequences, but an input which includes the arguments for the method under test along with the attribute values of the instance under test.

Following the strategy of Korel, BINTEST tries to iteratively satisfy the predicates that occur along a particular path in the control flow graph of the method under test by incrementally modifying a concrete user-provided candidate input. In addition to the concrete input, BINTEST makes use of user-provided domain intervals which are iteratively bisected on a per-variable basis if the input does not satisfy some path predicate. The middle element of one of the bisected intervals becomes the variable value at the considered position of the input. The assumed monotony of the expressions of the condition to be evaluated favorably is exploited to select the interval to continue with after bisecting.

For class-type arguments, the state of the input objects is modified using a particular `midValue` method that each participating class must implement. This method creates an object that is the middle element of a given interval.

BINTEST requires that a total ordering exists for the domain of each input variable. Additionally, the path predicates must exhibit monotone behavior in order for the search to be effective and efficient. However, especially for objects, usually no (intuitive) total ordering exists. For instance, when thinking of a class `Person` that models the properties of a human, specifying an adequate ordering relation is hard or even impossible. Consequently, it is hard or impossible to implement the `midValue` method for such a class. Another consequence of this is that the monotony of the predicates cannot be exploited and hence no direction is provided to the binary search. Even if a total ordering can be specified for a particular class, the `midValue` method artificially introduces data dependence among the attributes of this class, possibly preventing relevant object states from being explored during the binary search.

Since Beydeda and Gruhn consider the attributes of an object as additional inputs, they implicitly break the encapsulation of the object. The legality of a test input must

be ensured by the user who is responsible for providing correct input domain intervals and proper implementations of the `midValue` methods for all relevant classes.

As the authors state, BINTEST suffers from the problem of combinatorial explosion when the input domains have to be divided into multiple intervals. Since each combination of intervals will be considered, a large number of binary searches are carried out in the worst case, thus making the approach inefficient. Furthermore, complex statements of the method under test must be decomposed into atomic ones in order for the framework to work. The authors argue that this decomposition could be automated. However, they do not offer a specific technique for doing so. Their work does not address non-public methods.

### eToC

Tonella (2004) uses a genetic algorithm for the generation of test sequences. He proposes a representation of test sequences using a source-code-like structure. Effectively, Tonella makes no distinction between phenotype search space and genotype search space. Tonella's GA applies four special mutation operators and one crossover operator for the incremental evolution of the candidate solutions: mutation of a primitive input value, constructor change (substitution of a constructor by an alternative constructor of the same class), insertion of a method invocation, removal of a method invocation, and one-point crossover (cutting two sequences at randomly defined parts and reassembling the obtained fragments). Tonella's objective functions follow the distance-oriented approach, meaning that they incorporate a distance metric which expresses how close the execution of a test sequence is to covering the desired code element. As the distance metric, he uses the number of control dependences covered during test sequence execution, as suggested by Pargas et al. (1999). In initial experiments, Tonella's approach was able to automatically create tests for six classes from the Java standard class library that achieved a relatively high percentage of branch coverage.

McMinn (2004b) argues that using solely the number of covered control dependences as objective value creates a search space containing plateaus which degenerate the evolutionary search to a random search if the algorithm reaches such a plateau. Hence, the search may be unsuccessful in the presence of non-trivial predicates. Tonella's approach does not address the coverage of non-public methods. Furthermore, the mutation operators *insertion of a method invocation*, *constructor change*, and the crossover operator *one-point crossover* do not inherently preserve the feasibility of the test sequences they operate on. Consider the example application of the crossover operator as shown in Figure 2.4 on the facing page. At the top, the figure shows the two parent method sequences to be crossed over (using Tonella's notation: constructor calls are abbreviated, and primitive values are separated from the calls and located at the end of the sequence after the '@' sign, the assignment with the `null` reference includes the actual type). The crossover operator (cut-points indicated by the lines in the parent individuals) produces two offspring individuals that are shown in the middle of Figure 2.4 on the next page. The right offspring individual represents an infeasible method sequence since there is no parameter object available for the call

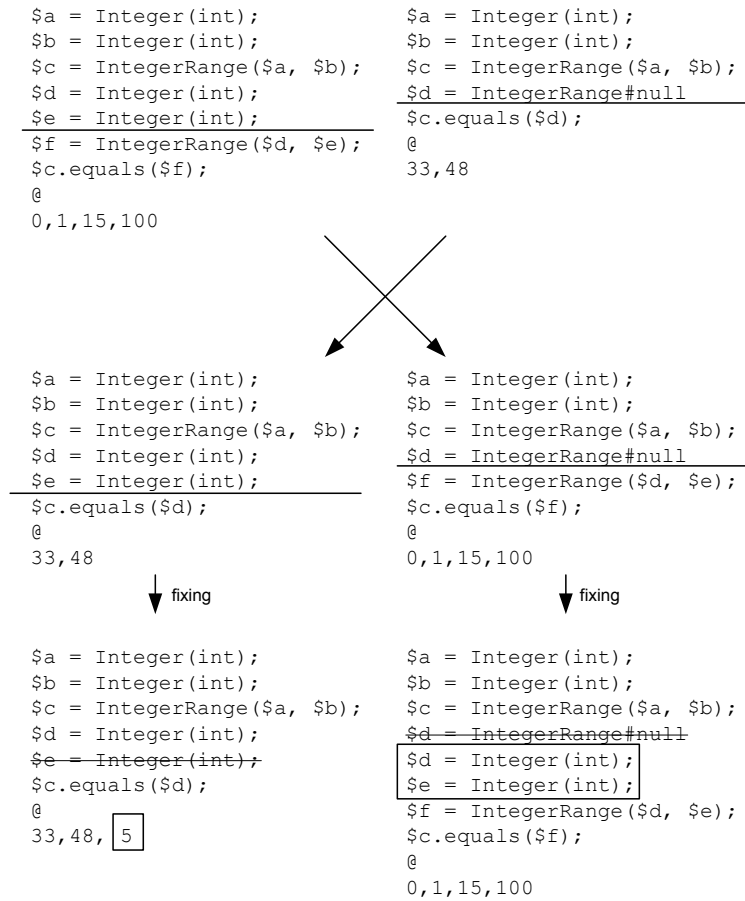


Figure 2.4: Example application of Tonella's crossover operator

of constructor `IntegerRange`. Additionally, the offspring individuals contain useless method calls. To deal with these anomalies, the crossover operator also includes a fixing phase in which constructors for the missing objects are randomly inserted before the method that requires them (in the figure, the boxed statements are the inserted ones). This can be a recursive procedure if some classes have no parameter-less constructor. Furthermore, unnecessary method calls are removed and the primitive arguments are adapted (the boxed value has been inserted). The resulting offspring individuals (shown at the bottom of Figure 2.4 on the preceding page) represent feasible method sequences. The mutation operators *insertion of method invocation* and *constructor change* also exhibit the issue of missing and unnecessary method calls and therefore include a fixing phase, too. Applying fixing to the variegated individuals is not problematic; however, it reduces the performance of the search. The approach has also problems if a candidate test sequence causes an uncaught runtime exception (which, for instance, occurs if a randomly generated parameter violates a method precondition).

## ETOOS

Wappler (2004) elaborates on an extension of the sequence testing approach of Baresel, Pohlheim and Sadeghipour (2003) in order to apply it to object-orientated software (ETOOS = evolutionary testing of object-oriented software). It considers a method sequence as being composed of the three elements *methods*, their respective *object assignments* and *primitive parameter values*. For instance, the method sequence shown in Listing 2.1 can be considered as the composition of the elements shown in Figure 2.5. The *methods* identify which methods are to be called, the *object assignments* specify

Integer(int)	-	0
Integer(int)	-	100000
IntegerRange(Integer,Integer)	1@Integer, 2@Integer	-
IntegerRange.clone()	1@IntegerRange	-
IntegerRange.equals(Object)	1@IntegerRange, 2@IntegerRange	-
<i>methods</i>	<i>object assignments</i>	<i>basic-type params</i>

Figure 2.5: Decomposition of a test sequence

which instances to be used for the individual method calls, and the *basic type parameters* specify the primitive argument values for the method calls. Composing a test sequence in this way allows for a representation based on integers and reals. In turn, this allows using off-the-shelf genetic algorithms for test sequence search. The methods are represented using a vector of integers. For evaluation, the integer values of a genotype individual are mapped to the actual methods using a method lookup table. Thereby, an integer serves as a key for the table entries. The object assignments are also represented by a vector of integers. The candidate instances which are potentially available for the particular method calls are serially numbered. As shown in Figure 2.5, for each class, a pool of available instances is maintained. The genotype integers are then used to identify the instances to be used from the appropriate instance pool for the respective method

calls. Wappler applies a two-level evolutionary algorithm: on the first level, method integer sequences are optimized, whereas on the second level, the respective object assignments and basic type parameters derived from a concrete method integer sequence are optimized. Performing a two-level optimization allows the genotype specification for the second level to be exactly tailored to the candidate method sequence from the first level.

However, when optimizing the method integer vectors, infeasible individuals can also be generated. For instance, the first integer might be mapped to a call of a non-static method. This method cannot be executed since, initially no target objects are available. Therefore, Wappler uses different penalty functions in order to guide the evolutionary search to regions in the search space that contain feasible solutions. Nevertheless, the generation of infeasible individuals makes the approach inefficient since, in some cases, many generations are necessary to encounter a feasible solution. Furthermore, the search may suffer from a loss of diversity if the search space contains only relatively few feasible individuals and the regions containing them are not closely related to each other. As the search then concentrates on one of those regions, the approach may fail if the ideal solution can only be found in another region. The approach of Wappler does not address non-public methods. Similar to the approach of Tonella, the approach of Wappler has problems if a candidate test sequence causes an uncaught runtime exception. Furthermore, the user must specify the maximum length of the test sequences.

### Other Dynamic Approaches

Liu, Wang and Liu (2005b) combine ant colony optimization (ACO) with a multi-agent genetic algorithm (MAGA). While the former is applied to construct method call sequences, the latter is employed to optimize the corresponding primitive arguments. The description of the approach and its validation are very preliminary.

Oster (2007) applies multi-objective search strategies to create data-flow-oriented test suites. The specialty of this approach is that a candidate solution is a complete set of tests rather than a single test. One criterion to optimize is the number of tests in the set of tests, another criterion is the number of covered code elements. The multi-objective search then attempts to find a set of tests with a trade-off between the number of individual tests and achieved code coverage. By doing so, however, the achievable coverage might be not maximal.

### 2.2.3 Commercial Test Generators

Today's market for software testing tools offers a great variety of different products to support the testing process. Among these, three test sequence generators are found which are capable of automatically creating structure-oriented class tests. Since they are commercial, the vendors do not reveal much of their internal functioning. Therefore, the following descriptions assemble information taken from technical whitepapers and published investigations which attempted to disclose their underlying principles.

### **Agitator**

The testing tool Agitator is provided by Agitar Software, Inc. Besides other features that support the test process, it includes a facility to generate test sequences for a given class automatically. Agitator accomplishes a static analysis which Boshernitsan, Doong and Savoia (2006), the vendors of the tool, characterize as a heuristics-driven path analysis intended to identify input value constraints. Various constraint solvers are involved in order to find suitable input values. In order to cope with complexity and performance issues, Agitator uses multiple shortcuts and approximations. For instance, constraint solving occurs only for part of the desired execution path. Also, specialized and generic constraint solvers are applied. For instance, for string objects, a special string constraint solver is used. The static analysis is combined with a set of heuristics. For instance, for an integer value occurring as an input value, the values  $-1$ ,  $0$ , and  $1$  are tried in addition to each constant found in the source code. Each of the encountered constants is additionally tried with an increment and decrement of  $1$ , respectively. For string values, random text and arbitrary combinations of alpha, numeric, and alphanumeric elements are tried. String constants found in the source code under test are also considered.

It appears that Agitator performs this analysis for the method of interest only. The creation of the required instances is accomplished by using either an arbitrary available constructor, or a user-provided instance factory. In order to provoke state transitions of the involved objects, the instance methods are called with auto-generated arguments. Agitator breaks the encapsulation of the tested objects by directly accessing the class internals. The tool comprises an additional reflection framework. Methods referring to this framework are part of the generated test sequences.

### **CodePro**

CodePro is a testing tool provided by Instantiations, Inc. Like the other commercial tools, beside a lot of other features it offers a mechanism to automatically generate unit test sequences. CodePro attempts to maximize line coverage, which is very similar to statement coverage. The generation algorithm is based on a set of heuristics for primitive input generation and object instantiation: at first, it performs some kind of static analysis in order to identify relevant values for primitive and string parameters. For instance, in the case of a switch statement, each value appearing in a non-empty case label is considered interesting. In the case of a class-type parameter, it tries to instantiate the class using zero-argument static methods. If this fails, it tries to create an instance using constructors that require arguments, by recursively creating the necessary parameter values. Once interesting parameters are obtained, CodePro examines various combinations of them. Since the number of all possible combinations of parameter values for a given method can be impractically huge, additional heuristics are applied in order to focus on promising combinations only.

CodePro's test-case-generating algorithm is deterministic, meaning that the application of the heuristics occurs in a predefined manner and without any randomness. Thus, it always generates the same test sequences for a given test object.



CodePro takes public class members into consideration only while non-public class elements are ignored. However, in contrast to many other related approaches, the test sequences that CodePro generates exclusively refer to the public interface of the classes without breaking object encapsulation.

### Jtest

Jtest is a testing tool provided by Parasoft, Inc. The main purpose of Jtest is to generate test sequences at random which provoke uncaught runtime exceptions. At the same time, it tries to maximize statement coverage or branch coverage, depending on the configuration by the user. The vendors state that some kind of random generation takes place which, however, involves static analysis information. Csallner and Smaragdakis (2004) make the observation that Jtest uses chains of method sequences where constructor calls are recursively chained up to a chain length of 3. The authors also assume that Jtest heuristically applies values that are known to usually cause problems (such as `null` references).

Xie et al. (2005), whose approach is built on top of Jtest, state that Jtest uses *smart random generation*. These authors suggest – with respect to Jtest – that random tools often generate the same test sequences and obtain no test sequences that achieve full coverage. They also observe that Jtest creates many redundant test sequences. Jtest also attacks non-public methods. However, it breaks encapsulation meaning that the resulting test sequences involve calls to the Jtest framework in order to intrusively create objects in needed states and to directly call non-public methods.

#### 2.2.4 Limitations of the Existing Approaches

Table 2.2 recapitulates the existing approaches. Column *originator* names the originators

originator	name	type	criterion
Buy et al. (2000)	<i>ATOC</i>	academic	def-use pairs
Xie et al. (2005)	Symstra	academic	branch
Visser et al. (2004)	PathFinder	academic	branch
Tonella (2004)	eToC	academic	decision
Wappler (2004)	<i>ETOOS</i>	academic	decision
Beydeda and Gruhn (2003)	BinTest	academic	path
Sen et al. (2005)	jCUTE	academic	path
Boshernitsan et al. (2006)	Agitator	commercial	decision
Parasoft, Inc. (n.d.)	Jtest	commercial	decision
Instantiations, Inc. (2007)	CodePro	commercial	line

Table 2.2: Related approaches

of the approach. Column *name* gives the name of the approach assigned by its originator.

A name in *italic* indicates that it was not assigned by the originator but is only used in the context of this thesis for the purpose of better reference. Column *type* indicates whether the approach is mainly academic or commercial. In the former case, publications including a technical description of the underlying algorithms is available, as opposed to the latter case. Column *criterion* gives the most advanced coverage criterion that the respective approach supports (some approaches support multiple criteria, such as Jtest which allows choosing between statement coverage and branch coverage). Most of the approaches are academic. Nearly all approaches support control-flow-oriented coverage criteria; the only approach supporting a data-flow-oriented coverage criterion is that of Buy et al. (2000).

In the following, the limitations that have already been pointed out in the individual descriptions of the approaches are systematically elaborated, discussed in more detail and finally summarized.

### Limitations of Symbolic Execution and Constraint Solving

Mainly the usage of loops, arrays, and dynamic data structures hinders the application of symbolic execution. When loops, or recursive function calls, appear in the code, these constructs must be expanded in order to make a comprehensive exploration of the state space possible. In the case of unbounded loops, where the loop counter is not bounded by a constant, as well as in the case of recursion, the expansion might result in a very high or even infinite number of symbolic states (Tillmann and Schulte, 2006; Michael et al., 2001; Tracey, Clark and Mander, 1998a). Also, in the case of a large number of branching statements, the number of symbolic states may grow impractically huge. These issues are known as *state space explosion*. Problems are also encountered when arrays are accessed using non-constant variables as indices (Michael et al., 2001). Then, array element determination might be infeasible (Korel, 1990). Also, the presence of pointers causes symbolic execution to run into difficulties (Michael et al., 2001). It may also struggle if the involved statements of the program are not purely mathematical. Constraint solving, which is applied to simplify the path conditions and to obtain concrete input values, can be difficult if the path conditions are complex; then, the constraint systems might not be solvable. In particular, solving constraints including pointers is an undecidable problem (Sen et al., 2005).

An approach to automatic test generation that applies symbolic execution and constraint solving automatically inherits the limitations immanent to these techniques. However, some of the related approaches use approximations and heuristics to address these issues.

### Limited Support of Class Type Arguments

Object-oriented real-world applications make use of classes whose methods possess not only primitive argument types, such as `integer` or `char`, but also classes, interfaces, and arrays. Therefore, testing a class by examining its methods involves the creation of parameter objects. For the coverage of some code elements during testing, the parameter

objects must be in particular states. Consequently, a class test involves the creation of an instance of the class under test along with the creation of needed parameter objects. Additionally, the test calls methods on the instances in order to provoke state changes.

Demanding that the class under test possesses methods with primitive arguments only does not require test sequences that create parameter objects. Assuming that an instance of the class under test can easily be created, this requirement even enables the application of conventional test generation techniques for procedural software. An object-oriented test generation approach that demands that solely primitive arguments are required is applicable to a very limited range of classes only.

### **Limited Maintainability and Usability of the Results**

Some researchers suggest violating the principle of encapsulation in order to facilitate the setup of a test and to increase observability. For instance, some suggest accessing the non-public attributes of a class via additional methods that are added to the class during testing. Then, the states of the objects participating in the test can be set easily. Additionally, the states can easily be inspected, enabling the formulation of more detailed assertions. An alternative to inserting additional methods is using the reflection mechanisms provided by the runtime system, if such is available.

However, breaking encapsulation is considered bad practice by other researchers (Binder, 1999) for several reasons. One reason is that inserting special testing methods potentially introduces faults, hence making the localization of the causes of erroneous behavior during test execution more difficult. Another reason, possibly the most important one, is that by directly setting the state of an object, states may be achieved that contradict the specification, meaning that they do not comply with the class invariant. For instance, imagine a public setter method that sets the value of a private attribute. Assume that it checks the value passed to it: if it satisfies a particular condition, the attribute will actually be set to that value, otherwise the value will be rejected. When directly accessing that attribute via reflection or an artificially inserted setter method, it is possible to set the attribute value to any arbitrary value, regardless of whether or not it would have passed the check by the original setter. As a result, the tests may examine behaviors of the tested class that cannot occur in reality when the class is integrated into an application. Furthermore, tests that directly refer to the non-public members of the class under test are easily broken by refactorings. For instance, if a private method is renamed, the tests that refer to that method must be reworked.

### **Search Spaces with Inexecutable Tests**

The nature of the dynamic approaches is that they search the space of all possible test sequences to find a sequence that attains the targeted test goal. It depends on the definition of the search space as to whether or not it contains only executable test sequences. For instance, the representation used by Wappler (2004) defines the search space in such a manner that inexecutable test sequences can occur during the search. Although finally an executable test sequence will be delivered as the result of a search

(presuming it is successful), the search must also deal with inexecutable sequences that intermediately occur. However, searching through regions of the search space that contain inexecutable test sequences incurs extra time and cost as a result of the high number of objective function evaluations. This tends to lower the efficiency of the search in general. In the presence of complex method call dependences, for instance if the test cluster classes do not possess default constructors that do not need any argument to be called, the regions with inexecutable sequences are relatively large. As a result, the evolutionary search must first discover regions with executable test sequences before exploring them in depth to find a covering one.

### **Limited Effectiveness for Complex Predicates**

A predicate is complex if it is composed of several conditions whose satisfaction is not trivial. A condition is trivial if the majority of input situations (parameters of method calls and object states) satisfies it. Only a test that meets all the prerequisites to satisfy the conditions by detecting appropriate method arguments and setting up the object states suitably is able to cover a test goal depending on a complex predicate. However, complex predicates are a challenge to both static and dynamic test generation approaches. With the static approaches, a complex predicate becomes part of the path constraint when symbolically executing the class under test with the intention to encounter a path to the test goal at hand. The constraint solver must calculate a solution to the constraint system which includes the complex predicate. This might require great computational power or is even not possible with today's constraint solvers (Sen and Agha, 2006). With the dynamic approaches, the search is not guided sufficiently if the objective function used does not account for the single conditions of which the predicate is composed. In this case, the objective function landscape contains plateaus which do not provide great help to the search, since they do not indicate any direction in which better solutions are expected to be found. For instance, the objective functions that Tonella (2004) uses do not account for the single conditions and hence degenerate the evolutionary search to a random search in the presence of complex predicates.

### **Limited Treatment of Runtime Exceptions**

Runtime exceptions pose a problem especially to the dynamic test generation approaches. Section 3.5.5 on page 80 elaborates on the concept of runtime exceptions in more detail. They occur for instance, if a method is called with unsuitable arguments, meaning that the arguments violate implicit method preconditions. In an optimal case, the search space that a dynamic approach explores does not contain test sequences that contain method calls with improper arguments. However, since the source code usually does not indicate which arguments are valid and which are not via static analysis without formally specified preconditions, the search space is typically defined so that also test sequences violating method preconditions can be encountered. If this happens and such a test sequence is evaluated, very often a runtime exception occurs that terminates the execution of the test sequence. The existing dynamic approaches have problems in this

case; they simply abort the current search and proceed with the next test goal to do, if they do not break down completely.

### Limited Support for Non-Public Methods

Achieving high code coverage of the class under test demands that non-public methods are also addressed during test generation. Thoroughly testing a non-public method requires that a public method that calls the non-public method is invoked with appropriate arguments and the instance under test is in a proper state. However, the existing approaches do not address the generation of tests for non-public methods. As a result, the achievable code coverage is suboptimal. Although an extension of symbolic execution to non-public methods as well as the distance calculation with respect to test goals belonging to non-public methods might be feasible, no existing work is able to handle the problems of recursion and polymorphic method calls in this context.

### Need of Additional User Input

Some approaches require the user to provide additional input to the test generator apart from the source code of the class under test. For instance, BINTEST (Beydeda and Gruhn, 2003) depends on a user-provided test to start with and user-provided domain intervals. Furthermore, the classes involved in the test must implement a particular framework method.

Demanding additional user input, such as the implementation of framework methods or formally specified method preconditions, postconditions, and class invariants, is not considered a substantial limitation. It is therefore not listed in Section 1.1 on page 3. However, the usability of an approach is considered limited in the case if the provision of the respective input is not straightforward but requires intensive labor on part of the user.

### Summary

Table 2.3 on the following page gives an overview of the limitations of the existing approaches. An “x” in a cell indicates that the column property applies to the respective approach. A “\*” indicates that the approach is likely to possess the respective limitations. For instance, the approaches relying on constraint solving typically have problems with complex predicates. However, this depends on the type of constraint solver and whether constraint solving is combined with a heuristic, as Agitator does this for instance. A “?” indicates that it is not known whether or not the approach has the respective limitation due to the lack of description in the literature.

## 2.3 Evolutionary Algorithms

Search-based solutions to problems become particularly interesting when analytical techniques are either too expensive or not appropriate. For instance, NP-hard problems

Limitation	ATOC	Symstra	PathFinder	eToC	ETOOS	BINTEST	jCUTE	Agitator	CodePro	Jtest
SE+CS limitations	x	x	x	-	-	-	x	*	?	?
primitive arguments only	x	x	-	-	-	-	-	-	-	-
breaking of encapsulation	-	-	x	-	-	x	x	x	-	x
search space issues	-	-	-	x	x	-	-	-	-	-
complex predicate issues	*	*	*	x	-	*	*	?	?	?
runtime exception issues	-	-	-	x	x	x	?	-	-	-
public methods only	x	x	x	x	x	*	*	-	x	-
need of additional input	-	x	x	-	-	x	x	-	-	-

Table 2.3: Limitations of the approaches

(NP = Nondeterministic Polynomial-time) are such problems (Garey and Johnson, 1979), for the solution of which an analytical, deterministic algorithm may be known but might not be practicable due to its complexity. It might require a computer to calculate for thousands of years in order to provide a final solution.

Evolutionary algorithms represent an optimization technique that has been shown to be effective for attacking NP-hard problems, and, in general, for handling problems for which classical techniques are not feasible. They do not require the derivation of the function to be optimized nor gradient information, not even a mathematical model of the problem such as differential equations. They have turned out to be extremely powerful when being applied to problems that are related to non-linear, multi-modal, and discontinuous search spaces. Search-based software test generation involves problems of this type. Evolutionary algorithms suggest themselves as the optimization technique of choice for search-based testing since the respective search spaces typically exhibit characteristics with which traditional optimization techniques struggle.

This section describes the principles of evolutionary algorithms to enable a better understanding of the principles of evolutionary structural testing. Section 2.3.1 presents the foundational mechanisms of evolutionary algorithms in general. Genetic algorithms and genetic programming, both of which are particular types of evolutionary algorithms, are focused on in Section 2.3.2 on page 44 and Section 2.3.3 on page 46, respectively. Particular emphasis is given to these two types since they are important ingredients for the evolutionary class testing approach described in Chapter 3 on page 53.

### 2.3.1 Evolutionary Algorithm Principles

An evolutionary algorithm is a stochastic, metaheuristic optimization technique, or search strategy, respectively. It is based on the fundamental mechanisms of *selection* and *variation* in analogy with the Darwinian theory of biological evolution (Darwin,

1859). This theory postulates the *survival of the fittest* as one of its main principles.

Since research in the field of evolutionary algorithms has started in the 1940s (Eiben and Smith, 2003), many different types of evolutionary algorithms have been developed. While in the beginning various researchers worked independently, using their own terminology and mechanisms, today the field of evolutionary computation tries to unify the terminology and to abstract from particular implementations. As a consequence,

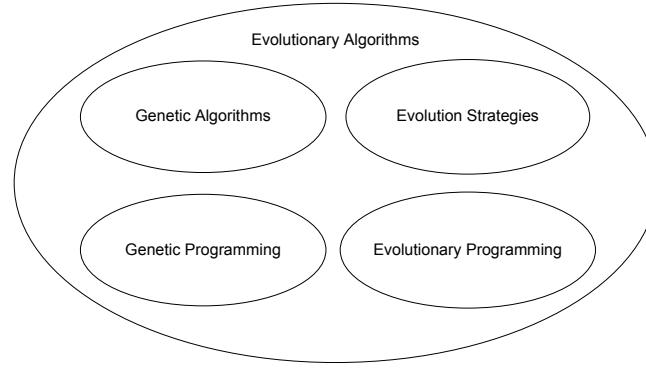


Figure 2.6: Classification of evolutionary algorithms according to Eiben and Smith (2003)

the term evolutionary algorithm is widely used to denote the abstract concept of a bio-inspired algorithm, hence constituting a particular class of optimization algorithms. Elements of this class are again classes of algorithms that have evolved independently. Figure 2.3.1 shows a common classification of evolutionary algorithms according to Eiben and Smith (2003). Pioneers of the four types of evolutionary algorithms are the following:

- genetic algorithms: Holland (1975)
- genetic programming: Koza (1992)
- evolution strategies: Rechenberg (1971)
- evolutionary programming: Fogel, Owens and Walsh (1965)

The most prominent distinguishing features of the different types are both the representation of a potential solution to the search problem under consideration and the specific variation and selection mechanisms which each type characteristically uses. The principles of evolutionary algorithms in general will be detailed in the following. The subsequent two sections are dedicated to genetic algorithms and genetic programming, respectively. Both evolution strategies and evolutionary programming are not addressed since they are not applied in this thesis.

An evolutionary algorithm works on a set of candidate solutions, the so-called *population* (denoted by  $P$  in the following). The candidate solutions are also called *individuals*. The individuals represent potential solutions to the optimization problem to be solved. An individual consists of one or more *genes*, each of which models an independent

aspect of the solution. The parallel nature of the search by working on a population of individuals is intended to avoid getting stuck in local optima as optimization techniques working on a single candidate solution only may, such as hill climbing (Russell and Norvig, 1995). An evolutionary algorithm requires the notion of *fitness* which expresses how well an individual is suited to solve the optimization problem at hand in relation to the other individuals. Based on the information about the fitness of the individuals, an evolutionary algorithm iteratively selects promising individuals and creates new individuals from them using crossover and mutation (which, together, constitute the variation mechanism). An iteration of selection and variation is called a *generation*. The algorithm terminates if a sufficiently good solution has been found or another termination criterion applies.

Figure 2.7 shows how an evolutionary algorithm is employed. On the left, the figure shows the evolutionary algorithm as an agent. In order to generate reasonable individuals,

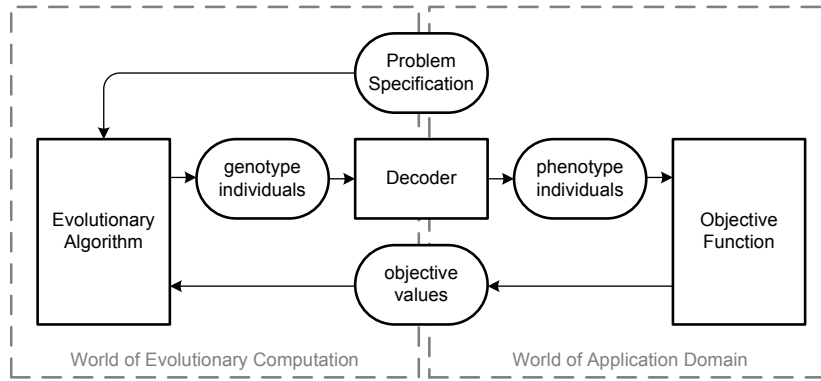


Figure 2.7: Evolutionary algorithm context

the algorithm needs a problem specification which describes how a candidate solution must look like and which constraints it must satisfy. This specification defines the *representation* of the candidate solutions, also called *encoding*. A representation defines how a candidate solution of the *phenotype search space*  $\Phi$  is encoded as a candidate solution in the *genotype search space*  $\Gamma$ . The phenotype search space is the set of all possible candidate solutions to the given optimization problem. The genotype search space is the set of all instances of the representation used. For instance, when optimizing the design of an airplane wing, all conceivable wing designs constitute the phenotype search space; a phenotype individual is one particular wing design. A particular wing design can uniquely be described by several design parameters. Thus, the genotype individual corresponding to a particular wing design is the set of the respective concrete design parameter values. The evolutionary algorithm does not work on wing designs, but rather on the parameters describing a wing design. The mapping of genotype individuals to phenotype individuals is performed by the decoder as shown in Figure 2.7. The task of the decoder is to *interpret* a genotype individual as a solution to the optimization



problem. The decoder can be modeled as a function  $\delta$  in the following way:

$$\delta : \Gamma \rightarrow \Phi \quad (2.5)$$

An evolutionary algorithm requires an *objective function*  $\omega$  to be provided. The fitness of an individual is based on the objective value that the objective function returns. In analogy to the Darwinian theory of evolution, the objective function defines the environment in which the individuals must prove their value. The objective value of an individual expresses how well this individual is suited to solve the given optimization problem in general. Poor individuals receive bad objective values while promising individuals receive good objective values. The objective function can be modeled as a mapping of the individuals to a particular rating value which is typically a real value. In the case of a multi-objective optimization, the objective value consists of multiple real values of which the evolutionary algorithm tries to obtain the best trade-off. In the case of single-objective optimization, as applied in this thesis, the objective function is defined as follows:

$$\omega : \Phi \rightarrow \mathbb{R} \quad (2.6)$$

where  $\mathbb{R}$  is the set of real numbers. An evolutionary algorithm aims at either minimizing or maximizing the objective value. For many optimization problems the ideal (theoretic) objective value is known; then, it is tried to approach this value as close as possible.

Figure 2.8 shows the major steps of an evolutionary algorithm including the evolutionary loop. These steps apply to the genotype individuals, not to the phenotype individuals. In the following, these steps are described in short. Afterwards, a more detailed description follows, including various common strategies for these steps.

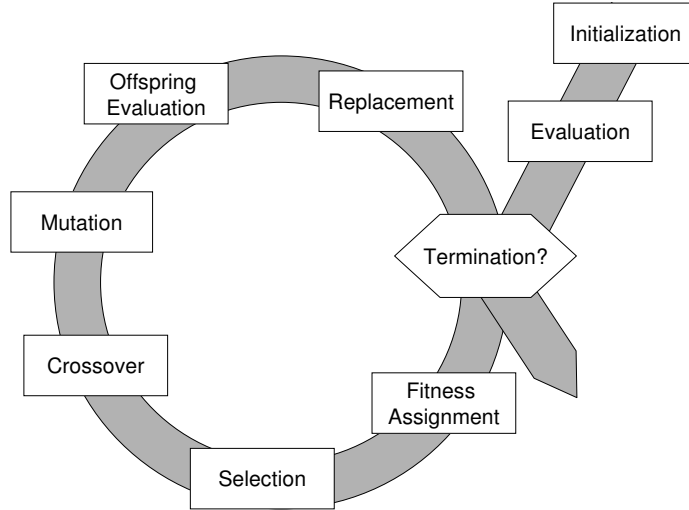


Figure 2.8: Principle procedure of an evolutionary algorithm

First, the algorithm creates the initial population (*Initialization*). Typically, the initial population consists of randomly generated individuals. Then, the algorithm evaluates

the individuals by calculating the objective value for each (*Evaluation*; thereby, the individuals are decoded to phenotype individuals to which the objective function is then applied). Thereby, it may turn out that a sufficiently good solution has already been found (*Termination*, where the user specified in advance what “sufficiently good” is). In this case, the algorithm terminates and delivers this solution. Otherwise, the algorithm enters the evolutionary loop. Before carrying out the selection of parent individuals that are to produce offspring (i.e. new candidate solutions), it assigns a fitness value to each individual (*Fitness Assignment*). The fitness values are based on the objective values of the individuals. Afterwards, the algorithm selects a subset of the current population based on the fitnesses of the individuals. This subset undergoes crossover where several individuals are recombined, thereby creating new individuals (*Crossover*). The new offspring individuals are then mutated meaning that some of their parts are slightly modified (*Mutation*). The mutated individuals are evaluated to obtain their objective values (*Offspring Evaluation*). The algorithm updates the current population according to the chosen update strategy (*Replacement*). Possibly, it discards existing individuals and inserts some of the offspring individuals. Finally, the algorithm checks whether any termination criterion applies in which case the algorithm exits the evolutionary loop and delivers the best individual found (*Termination*).

In the following, the steps are described in more depth. Various strategies to implement these steps are presented and discussed. The selection of the strategies to be described was driven by the question of whether they will be applied by the evolutionary class testing approach described in Chapter 3 on page 53. However, in order to provide a general overview of the field, additionally, popular strategies are pointed out for some of the steps.

## Initialization

During initialization, the evolutionary algorithm creates the initial population  $P_0$ . It creates random individuals that are equally distributed within the search space. However, if good solutions are already known – e.g. from a previous run of the optimization – then the algorithm can include these solutions. This strategy is called *seeding*. Also, problem-specific knowledge can be used to create initial individuals that represent promising candidate solutions, thus accelerating the evolutionary search.

## Evaluation

Evaluation means that the algorithm calculates the objective value of each individual. For doing so, it hands the set of relevant individuals over to the objective function that in turn delivers the objective values. The objective function is not necessarily a mathematical function. More generally, it can be imagined as a black box that accepts a set of candidate solutions and delivers the objective values of these solutions by performing any kind of computation.

### Fitness Assignment

The purpose of fitness assignment is to assign a fitness value to each individual in order to enable the directed selection of parent individuals. In general, the fitness of an individual directly depends on its objective value. Note that, while the objective value of an individual is an absolute value, the fitness of an individual is a value relative to all current other “competing” individuals. Two categories of fitness assignment strategies exist: ranking, and fitness-proportional assignment.

Ranking (Baker, 1985) exploits the ordering of the individuals defined by their objective values: it simply sorts the individuals according to their objective values. Thus, the amount of the objective value becomes essentially irrelevant; rather the position (rank) in the sorted list of individuals  $L = (i_1, i_2, \dots, i_n)$  will be decisive.

Linear ranking, as proposed by Baker (1985), defines a linear correlation between the fitness and the rank of an individual:

$$f(i) = 2 - p_{sel} + 2(p_{sel} - 1) \frac{p_L(i) - 1}{|P| - 1}, |P| > 1 \quad (2.7)$$

where  $p_L(i)$  is the index of the individual within the sorted list  $L$ ,  $p_{sel}$  is the selection pressure, and  $|P|$  is the size of the current population. Selection pressure is an adjustable parameter controlling the preference of individuals in terms of the objective value. The higher it is, the more individuals at good ranks will be favored.

Non-linear ranking (Pohlheim, 1995) is a modification of linear ranking intended to allow a higher selection pressure than linear ranking does. Non-linear ranking assigns the fitness according to the following formula:

$$f(i) = \frac{|P| \cdot X^{p_L(i)-1}}{\sum_{j=1}^{|P|} X^{j-1}} \quad (2.8)$$

where  $X$  is the solution to the polynomial  $0 = (p_{sel} - 1)X^{|P|-1} + p_{sel}X^{|P|-2} + \dots + p_{sel}X + p_{sel}$ .

Fitness-proportional assignment (Goldberg, 1989), an alternative category of fitness assignment strategies, establishes a direct correlation between the objective value and the fitness of an individual. Goldberg (1989) suggests the following scaling for fitness assignment:

$$linear : f(i) = a\omega(i) + b \quad (2.9)$$

$$linear - dynamic : f(i) = a\omega(i) + bg \quad (2.10)$$

$$logarithmic : f(i) = b - \log(\omega(i)) \quad (2.11)$$

$$exponential : f(i) = (a\omega(i) + b)^k \quad (2.12)$$

where  $a, b, k$  are problem-specific scaling parameters and  $g$  is the number of performed evolutionary cycles (generations).

However, since fitness-proportional assignment suffers from the issues of premature convergence (loss of diversity of the population, convergence to a local optimum) and

stagnation (extremely small differences of the fitness values that lead to an effectively random selection), ranking strategies have shown to be superior to fitness-proportional assignment (Pohlheim, 1999).

### Selection

Selection is conducted in order to choose the individuals that will produce offspring via crossover. A common requirement posed on a selection strategy is that it must not select the fittest individuals only – which decreases the ability of the algorithm to escape local optima – but also selects individuals with poor fitness with a particular (low) probability. In the following, the selection strategies stochastic universal sampling, tournament selection, and truncation selection will be explained.

Stochastic universal sampling (Baker, 1987) is a popular selection strategy that exhibits optimal characteristics concerning the resulting set of selected individuals (Pohlheim, 1999). Figure 2.9 illustrates how stochastic universal sampling works. Each individual

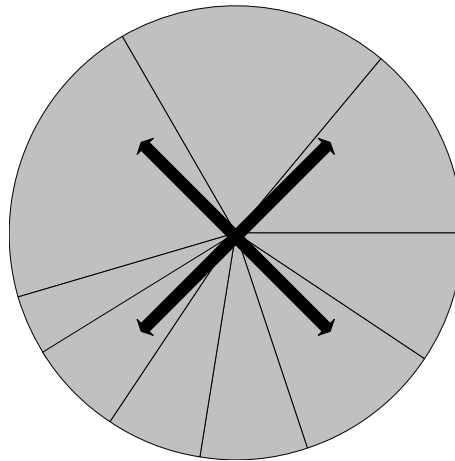


Figure 2.9: Stochastic universal sampling

of the current generation is assigned to a segment of a circle where the segment size correlates to the fitness of the individual: the higher the fitness is, the larger the segment will be. The segments are sorted according to their sizes. A pointer wheel is then placed at the circle where the angle between two pointers is equal for each two neighboring pointers. The number of pointers is equal to the number of individuals to be selected. When positioning the wheel, it is ensured that at least one pointer points at the individual with the best fitness (at the largest segment). Finally, all those individuals are selected at which at least one pointer points.

Tournament selection (Goldberg and Deb, 1991) performs individual competitions – so-called tournaments – among the individuals. For each individual to be selected, a single tournament will be carried out. A tournament consists of a random preselection of  $s_T$  individuals where  $s_T$  is the (user-defined) tournament size. The individual with the best fitness wins the tournament and will be added to the set of selected individuals.

In contrast to both stochastic universal sampling and tournament selection, which are rather natural strategies, truncation selection (Muehlenbein and Schliekamp-Voosen, 1993) is a rather artificial one. It simply selects the  $m$  best individuals of the population. The parameter *truncation threshold*  $T$  implies how many individuals will be selected:  $m = |P|T$ . Truncation selection defines the resulting set of selected individuals  $I_S$  as follows:

$$I_S = \{i \in P | p(i) \leq |P|T\}, 0 \leq T \leq 1 \quad (2.13)$$

### Crossover and Mutation

Crossover and mutation realize variation of the population meaning that they create new individuals. New individuals relate to unexplored points in the search space; hence, crossover and mutation are key steps of the evolutionary algorithm.

Crossover recombines multiple individuals (usually 2) that have been previously selected by a selection strategy. Typically, it generates as many offspring individuals as parent individuals are involved. The intention of crossover is to assemble those building blocks of the existing individuals that highly contribute to a good objective value to form new individuals that are even better suited to solve the search problem at hand. This idea is also known as the *building block hypothesis* (Goldberg, 1989).

While crossover applies to multiple individuals, mutation operates on exactly one individual. It slightly modifies an offspring individual by varying a gene of it with a relatively small probability. In case of an integer-valued gene, a mutation could be adding a small value  $\Delta v$  to the current value. Mutation is intended to explore the environment of a point in the search space where the size of the environment depends on the extend of the possible mutation step.

For both crossover and mutation, various strategies exist that depend on the underlying encoding. Both Section 2.3.2 on the next page and Section 2.3.3 on page 46 present crossover and mutation operators that are common in the field of genetic algorithms and genetic programming, respectively.

### Population Update (Replacement)

Population update aims at creating the next generation of individuals, thus establishing the new population. Population update is also called reinsertion or replacement, respectively. The following parameters define the behavior of the population update: the *reinsertion rate* specifies to which extend the individuals of the current population are to be discarded, while *generation gap* specifies the ratio of the number of individuals of the current population to the number of offspring individuals to be inserted into the new population. For example, a reinsertion rate of 1.0 indicates that the current population is to be completely replaced by the offspring individuals. Particular configurations of the two parameters have led to the definition of distinct population update strategies which have practically proven of value. Some of these will be explained in the following.

*Pure reinsertion* denotes the strategy where the number of offspring individuals is equal to the size of the current population; all individuals of the current population are

replaced by the offspring individuals.

*Uniform* and *elitist reinsertion* require the number of offspring individuals to be smaller than the size of the population. While uniform reinsertion replaces individuals of the current population uniformly randomly, elitist reinsertion replaces the unfittest individuals by offspring individuals.

*Reinsertion with offspring selection* does not insert the entirety of the offspring individuals, but selects among them based on their objective values. Typically, it applies truncation selection.

## Termination Criteria

An evolutionary algorithm terminates if a predefined termination criterion applies. The algorithm can be instructed to observe multiple criteria simultaneously; if at least one of them is satisfied, the evolutionary algorithm terminates. One distinguishes between direct termination criteria and indirect termination criteria. Direct termination criteria relate to a single property of the state of the evolutionary algorithm, whereas indirect termination criteria take other aspects, such as the history of the evolutionary loops, into account. Direct criteria are for instance: the predefined number of generations is reached, a sufficiently good solution has been found, or a timer has expired. Indirect criteria are for instance: the standard deviation of the objective values of the whole population declines beyond a predefined threshold, the difference of the average value of the best objective values of the last  $k$  generations and the current best objective value declines beyond a predefined threshold, or the difference of the best and worst objective value of the current generation declines beyond a predefined threshold.

Taken individually, some of the mentioned criteria do not guarantee that the evolutionary algorithm terminates at all. Hence, in practice a combination of multiple criteria is usual where at least one criterion guarantees that the algorithm terminates after a practically relevant amount of time. For instance, the criterion imposing a maximum number of generations gives this guarantee.

### 2.3.2 Genetic Algorithms

Genetic algorithms constitute a particular type of evolutionary algorithms. It was Holland (1975) who pioneered its research. Originally, genetic algorithms were designed to operate on bitstrings (meaning that the genotype individuals were concrete bitstrings), providing crossover and mutation operators that reassemble bitstring and flip bits randomly in order to create offspring individuals. In the context of this thesis, the term genetic algorithm shall denote any evolutionary algorithm which is defined on a representation that allows encoding data structures of *fixed size*, meaning that each individual has the same length (the same number of genes). Typically, the underlying representation of a genetic algorithm is a vector of real values which, for instance, model particular design parameters. This representation requires accurate, representation-specific variation operators for crossover and mutation, while the evolutionary operators

for fitness assignment, selection, population update, and termination, described in Section 2.3.1 on page 36 can be applied (since they are representation-independent).

In the following, the strategies for individual initialization, the crossover operators and mutation operators used by genetic algorithms are described. Since there is a great number of different strategies, only those are included that are relevant for this thesis.

### Individual Initialization

The initial population of a genetic algorithm is created randomly, unless existing good solutions are known. To this end, for each gene  $G_i$  of an individual, a value is selected from the (user-provided) value range  $D_i$  with equally distributed probability. If the number of seeded individuals (known solutions) is smaller than the desired population size, the remaining individuals are created randomly as described above. If the number of seeded individuals is greater than the size of the population, a bigger initial population is allowed whose size is then adjusted after the first evaluation.

### Crossover Strategies

*Discrete recombination* creates a new individual by randomly assembling the genes of the parent individuals. A random variable  $b_i$  decides from which parent the gene at position  $i$  is to be copied to the offspring individual. The following formula describes this procedure:

$$G_i^{Offspring} = \begin{cases} G_i^{ParentA} & \text{if } b_i = 1 \\ G_i^{ParentB} & \text{otherwise} \end{cases} \quad (2.14)$$

where  $b_i \in \{0, 1\}$  is a Boolean variable with an equally distributed random value.

### Mutation Strategies

*Mutation of real values and integer values* applies to real and integer values. The treatment of integer values is essentially the same as that of real values, except that the values are rounded after mutation and that the smallest mutation step is 1 (However, the actual mutation step may be a real value nevertheless.)

This mutation strategy incorporates multiple adjustable parameters. *Mutation rate* is the probability of occurrence of a mutation (i.e. if the mutation strategy is to be applied to a single gene). *Mutation step* defines the maximum possible extent of a mutation in terms of the “delta” of a gene’s value. Muehlenbein and Schliekamp-Voosen (1993) suggest the following relation between the genuine individual, being composed of the genes  $G_i^{Genuine}$ , and the mutated one, being composed of the genes  $G_i^{Mutant}$ :

$$G_i^{Mutant} = G_i^{Genuine} + s_i r_i a_i \quad (2.15)$$

where  $i$  denotes the randomly selected position of the gene to be mutated,  $s_i \in \{-1, +1\}$  is the sign of the mutation value,  $r_i = r D_i$  where  $r$  is the *mutation range* and  $D_i$  the value range of the gene  $i$ , and  $a_i = 2^{-u_i k}$  is the proportion of the maximum step where  $u_i \in [0, 1]$  and  $k$  is the *mutation precision*. The values of both  $s_i$  and  $u_i$  are chosen uniformly randomly.

### 2.3.3 Genetic Programming

Genetic programming is a particular type of an evolutionary algorithm intended to evolve computer programs that fulfill a given task. As opposed to genetic algorithms, where a candidate solution consists of a fixed-size data structure representing a number of parameters, genetic programming basically allows generating and evolving data structures of variable size that represent program statements or computational operators, respectively.

Though dating back to the early 1980s, genetic programming has been intensively researched during the 1990s. The theoretical foundations and description of the principles of genetic programming are mainly due to Koza (1992).

Historically, genetic programming emerged from a machine learning background: given a training set of inputs and expected outputs, a computer program was sought that maps the given inputs to the outputs correctly, hence enabling to obtain the probable, so far unknown, outputs from new inputs. Multiple candidate computer programs were considered and iteratively modified in order to reduce the difference of the expected outputs and the actual outputs for all of the training inputs. The objective function was defined as follows:

$$\omega(i) = \sum_{s \in S} (i(s) - t(s))^2 \quad (2.16)$$

where  $i \in \Phi$  is a computer program ( $\Phi$  is the space of all computer programs),  $S$  is the set of inputs (samples), and  $t(s)$  is the expected output for the input  $s$ . For variation, crossover and mutation of computer programs were applied.

The most common representation that genetic programming uses is the tree representation. Thereby, a candidate solution is a program tree assembling multiple functions together and defining the sequence of execution. Alternative representations, which are not considered further in this work, are linear structures and graph structures. A description of these representations is given by Banzhaf, Nordin, Keller and Francone (1998). In the following, the tree representation is described in more detail since it is an essential component of the approach described in this thesis.

A program tree is a directed graph that consists of nodes, each of which represents a particular function, a variable or a constant, respectively. The children of a node are considered to deliver the arguments for the function that the node represents. Figure 2.10 on the facing page shows a simple program tree representing the mathematical expression  $-7x + 3$ . This expression might be a candidate solution for a particular optimization problem. The tree was constructed by randomly selecting functions, variables, and constants from a user-defined repertoire, referred to as the *function set*. Functions in this repertoire with at least one argument are referred to as *non-terminals*, whereas variables, constants, and functions with no arguments, are referred to as *terminals*. A constant, such as  $-7$ , is referred to as *ephemeral random constant* (ERC).

Originally, the arguments of the functions of the function set were not typed. This implies that the return value of any function as well as the type of any variable and constant must be able to be passed as an argument at each argument position of any function. This restriction is known as the *closure property* of the function set. While this



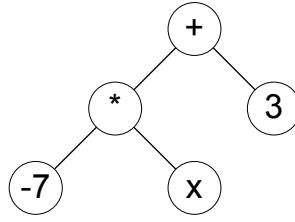


Figure 2.10: Simple program tree

restriction is easily fulfilled by function sets used in a purely mathematical context (e.g. when considering operations in real space  $\mathbb{R}$ ), it imposes, however, an impractical hurdle to the formulation of optimization problems of other application domains. Therefore, Montana (1995) extends genetic programming by introducing types and the notion of strong typing. Montana’s extension requires an additional, user-defined, *type set* to be specified. Both the return values and the parameter values of the functions of the function set are then typed, using a type from the type set. The evolutionary operators for initialization, crossover, and mutation care about the types and create individuals which satisfy all type constraints. Additionally, the tree itself is assigned a particular type  $\tau$ , allowing only functions returning values of that type  $\tau$  to appear as root node. Consider the function set shown in Table 2.4 Note that function “/” returns a real

function name	return type	argument types
+	$\mathbb{Z}$	$\mathbb{Z}, \mathbb{Z}$
-	$\mathbb{Z}$	$\mathbb{Z}, \mathbb{Z}$
*	$\mathbb{Z}$	$\mathbb{Z}, \mathbb{Z}$
/	$\mathbb{R}$	$\mathbb{Z}, \mathbb{Z}$

Table 2.4: Typed function set

value while all functions require integer values as arguments. Consequently, a genetic programming algorithm is not able to create a tree that contains the “/” function since it cannot appear as a child due to the type mismatch (presumed that the type of the tree is  $\mathbb{Z}$ ). In conclusion, strong typing means that the functions of the function set are formulated using types which are then respected during the evolutionary search.

Haynes, Schoenefeld and Wainwright (1996) extend the idea of strong typing by introducing type inheritance which enables the inclusion of polymorphic functions in the function set. A polymorphic function is a function that is declared to accept an argument of a particular type  $T$ , but is capable of being executed with arguments of any subtype of  $T$ .

In the following, the tree-based genetic programming operators for initialization of the initial population and for variation are described. Most of the strategies are applicable to both untyped genetic programming and to strongly-typed genetic programming. Where appropriate, the different implementations of the operators tailored to untyped and

strongly-typed genetic programming are pointed out.

### Individual Initialization

In order to set up the initial population of a genetic programming algorithm, the strategies *GROW* (Koza, 1992), *HALF* (Koza, 1992), and *Uniform* (Böhm and Geyer-Schulz, 1996) have proven of particular value. Uniform is a relevant strategy in the context of strong typing.

*GROW* (Koza, 1992) requires a maximum tree depth to be specified in advance. The depth of a tree is defined as the number of nodes that occur along the longest path from the root node to a terminal node. *GROW* starts by selecting a function from the function set randomly. This function becomes the root node. Then, *GROW* iteratively adds child nodes to the root by randomly selecting from both the non-terminals and the terminals. This procedure continues until all nodes possess their required children or the tree depth prior to the maximum tree depth is reached. Then, in order to satisfy the remaining unsatisfied child relations, *GROW* exclusively selects from the set of terminals. *GROW* does not necessarily build trees that have the specified maximum tree depth, since all child relations might be satisfied by terminals before reaching the tree depth. *GROW* requires that a terminal exists for each type.

The *FULL* strategy (Koza, 1992) is very similar to *GROW*. The only difference is that *FULL* does not select from both the non-terminals *and* the terminals, but only from the non-terminals unless the maximum tree depth is reached. Once, this depth is reached, it chooses nodes from the terminals only. Consequently, all branches of a tree created by *FULL* have the specified maximum depth. Like *GROW*, *FULL* requires a terminal to be present for each type.

*Uniform* (Böhm and Geyer-Schulz, 1996) is a tree building strategy intended to allow random tree creation with a uniform probability distribution of all program trees that are conceivable with respect to the given non-terminals and terminals. When using *GROW* or *FULL*, the probability distributions are not uniform since the probability of generating a tree with a particular depth decreases significantly as the desired tree depth increases (Böhm and Geyer-Schulz, 1996): for all conceivable trees with a valid depth (valid according to the desired maximum tree depth), the probability of being created during initialization is not equally distributed (in general, smaller trees have higher probabilities than bigger ones). Another restriction that *Uniform* deals with is that *GROW* and *FULL* require terminals to be present for each relevant type, since *GROW* and *FULL* must be able to add a terminal that returns a particular type when the maximum tree depth is being approached. This limits the applicability of *GROW* and *FULL* for some practical problems. *Uniform* precalculates the number of subtrees that are possible when considering the types that are used by the functions in the function set. For instance, for a type that is returned by exactly one function without any argument, exactly one subtree is possible (consisting of that function only). In the case that multiple functions return a value of the considered type, the number of subtrees is recursively accumulated over these functions. Once the subtree counts are calculated, *Uniform* generates trees taking the subtree counts into account. It also requires the

specification of a maximum tree depth. Details of the procedure can be found in Böhm and Geyer-Schulz (1996).

### Crossover Strategy

In general, one basic crossover strategy exists for tree-based genetic programming: subtree crossover. Thereby, randomly selected subtrees of the parent individuals are exchanged. Various ways have been suggested to select the subtrees. In the following, the initial, rather “naive” but very common version of subtree crossover, including its subtree selection strategy, is described.

Subtree crossover is applied to copies of the parent individuals. Figure 2.11 illustrates the procedure of subtree crossover. At first, it selects a subtree of each parent by random.

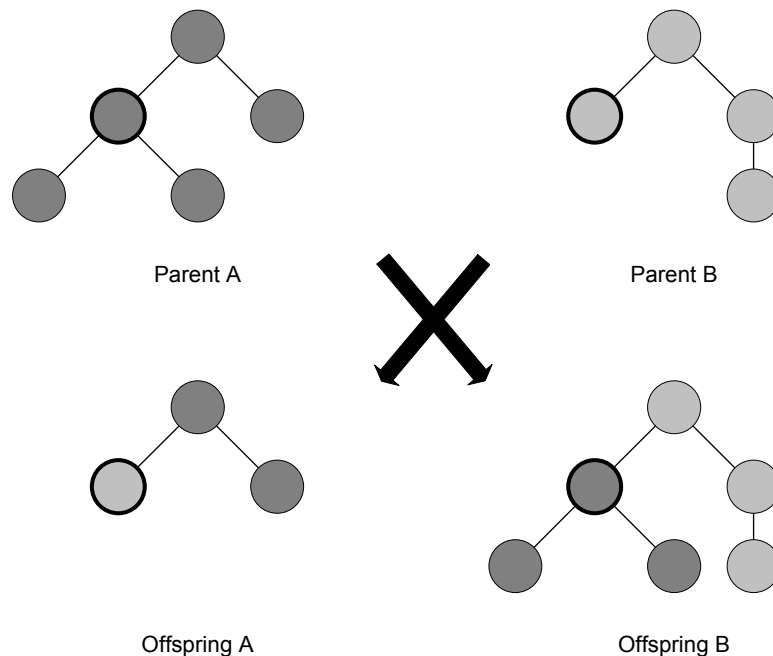


Figure 2.11: Subtree crossover

In the case of strongly-typed genetic programming, the selection of subtrees accounts for the types of the subtrees (i.e. the return types of the functions represented by the root nodes of the subtrees) and ensures that only subtrees with the same type are selected. Once two subtrees are selected, they are exchanged between the two individuals, resulting in two new program trees. If at least one of the new trees exceeds the predefined tree depths, the process will be repeated if appropriately configured. Subtree crossover returns the unchanged copies of the parents if it fails to select subtrees that would result in program trees of legal size.

### Mutation Strategies

*ERC mutation* applies to the ephemeral random constants. Figure 2.12 depicts how ERC mutation works. At first, it randomly selects a subtree of the individual to be

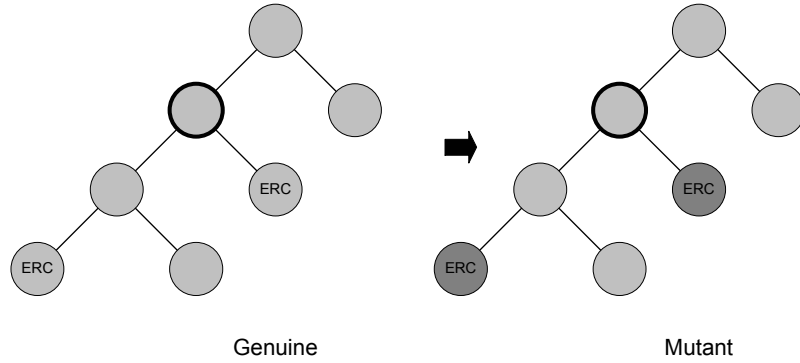


Figure 2.12: ERC mutation

mutated. Then, it identifies all ERC leaf nodes of the selected subtree and applies a type-specific mutation operator to each of the ERCs. For instance, in the case of an integer ERC, integer mutation as described in Section 2.3.2 on page 45 is applied. Note that an ERC itself represents a subtree and can potentially be selected. In contrast, as an extreme case, also the root node of the tree might be selected which results in the mutation of all ERCs of the individual.

*Demotion mutation* (Chellapilla, 1998) inserts a new function node into the individual to be mutated. Figure 2.13 illustrates the process. Initially, a subtree is selected randomly.

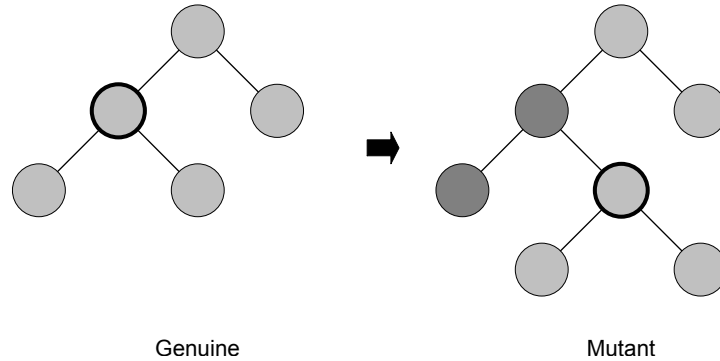


Figure 2.13: Demotion mutation.

Then, a new function randomly chosen from the function set is inserted between the parent of the selected subtree and the subtree itself. The new function becomes the new parent of the subtree, while the new function becomes a child of the former parent of the selected subtree. If the new function requires further children, randomly generated subtrees will be added to the new function. In the case of strongly-typed genetic

programming, the selection of subtrees is restricted to those exhibiting the following characteristics: at least one function exists in the function set which (1) has a return type that is compatible with the child type expected by the parent node of the subtree, and (2) expects at least one child of the type of the root node of the subtree. If the maximum tree depth is exceeded after the mutation, the offspring individual will be discarded and the procedure will be repeated until a tree with legal size is attained. If this fails after a predefined number of tries, the unchanged copy of the parent will be returned.

While demotion adds additional nodes to an existing tree, *promotion mutation* removes nodes. It can be seen as the inversion operation of demotion mutation. Figure 2.14 shows how promotion mutation is accomplished. At first, a subtree is selected randomly.

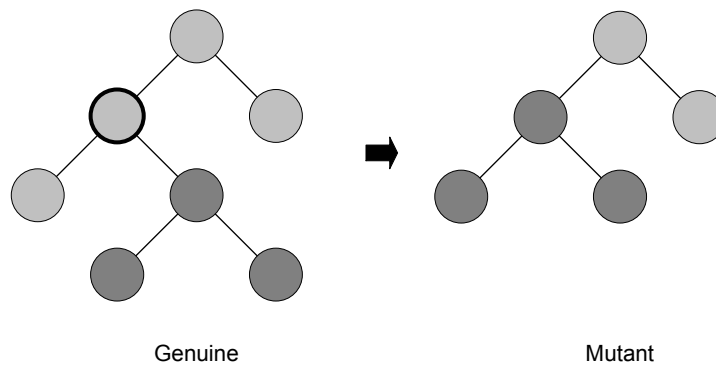


Figure 2.14: Promotion mutation.

Then, this subtree is replaced with a randomly selected child of the root node. In the case of strongly-typed genetic programming, the selection of subtrees and the selection of the child that becomes the new root of the subtree take the type constraints into account.

## 2.4 Summary

This section introduced the field of automatic test generation for object-oriented unit testing. At first, Section 2.1 on page 7 explained structure-oriented testing techniques. Structure-oriented testing techniques rely on fault models related to the source code of the unit under test. Various types of code elements, such as statements or branches of the respective control flow graph, are supposed to reveal faults when being executed or traversed, respectively. Each structure-oriented testing technique aims at generating a set of tests that maximize the number of covered code elements. Code coverage criteria, such as branch coverage, indicate the adequacy of a given set of tests and hence the quality of the overall test.

The existing approaches to automating various structure-oriented testing techniques for class testing were described in Section 2.2 on page 14. Two categories of approaches exist: static test generation and dynamic test generation. Static test generation relies

on symbolic execution and constraint solving in order to compute a test that covers a particular code element. Dynamic test generation interprets the task of generating a covering test for a given code element as a search problem. It applies a search strategy in order to obtain a covering test. The prominence of evolutionary structural testing as a dynamic test generation approach was also pointed out. It applies evolutionary algorithms, which are meta-heuristic optimization techniques, dealt with in detail in Section 2.3 on page 35. The existing approaches were investigated and their limitations were analyzed in Section 2.2.4 on page 31. The analysis identified a set of seven limitations that this thesis will address in the following chapter.

## 3 Evolutionary Class Testing

This chapter proposes evolutionary class testing, a new approach to the automatic generation of object-oriented unit tests. The approach has been developed in order to address the limitations studied in Section 2.2.4 on page 31. It is a dynamic test generation approach that uses genetic programming for the automatic generation of test sequences. The effectiveness of the approach will later be empirically assessed in Chapter 4 on page 103.

### 3.1 Overview

The major idea of evolutionary class testing is to transform the task of generating covering test sequences for the class under test to a set of optimization problems which a genetic programming algorithm then tries to solve. Figure 3.1 illustrates the basic concept of evolutionary class testing. All steps can be accomplished completely automatically without the need of human interaction. Initially, the set of test goals will be defined

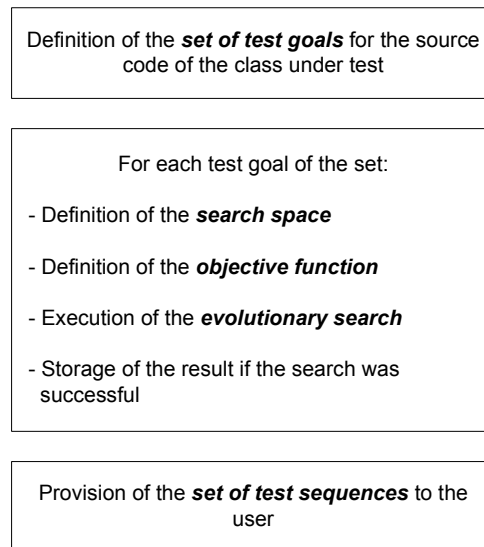


Figure 3.1: Basic concept of evolutionary class testing

based on the source code of the class under test. In the case of branch testing, each branch of the control flow graphs of the methods becomes a test goal. Other testing techniques lead to different definitions of test goals. Once the set of test goals is defined,

the following steps will be performed for each test goal: first, the search space will be defined, that is the space of all conceivable test sequences referring to the test goal. The definition of the search space is accomplished by specifying both the function set and the type set to be used by a genetic programming algorithm. Second, the objective function will be defined with respect to the test goal. Third, an evolutionary search will be carried out using a genetic programming algorithm that attempts to optimize the objective function. Finally, if the search was successful, the found test sequence will be stored. After all test goals have been processed in this way, the stored test sequences will be provided to the user.

The definition of the search space will be discussed in the following three sections (Section 3.2, Section 3.3 on page 56, and Section 3.4 on page 72). The first of these sections reconsiders test sequences more formally in order to better understand the phenotype search space  $\Phi$  and the ideas of the two representations discussed in the following sections. The second section describes the representation of test sequences based on method call trees and number sequences, while the third section extends this representation. These representations implicitly define the search space.

The definition of the objective function for a test goal will be discussed in Section 3.5 on page 76. The various situations that may occur when evaluating a candidate test sequence will be studied in detail. Several distance metrics and penalties used to define the objective function will be explained.

Afterwards, Section 3.6 on page 85 pays attention to the definition of the test cluster in the presence of uninstantiable types, such as interfaces or abstract classes. Arrays will also be treated in this section.

Before assessing the evolutionary class testing approach with respect to the attacked limitations in Section 3.8 on page 98, objective function definition will be revisited in Section 3.7 on page 88. The section suggests two strategies to improve the objective functions for test goals that depend on *function-assigned flags*.

## 3.2 A Formal Consideration of Test Sequences

In general, a test sequence is a sequence of method calls when assuming that no control structures, such as `if` statements or loops, are present (cf. Section 2.1.3 on page 10). Before defining test sequences, the definitions of both *test cluster* and *set of candidate methods* are necessary:

**Definition 3.2.1.** *Let  $\mathcal{C}$  be the set of all classes and  $\triangleright$  be the class association relation so that  $c_i \triangleright c_j$  with  $c_i, c_j \in \mathcal{C}$  means that class  $c_i$  is either a superclass of class  $c_j$ , or  $c_i$  is associated to  $c_j$  by a general association relation. Furthermore, let  $\triangleright^+$  be the transitive closure of the class association relation  $\triangleright$ . Then,*

$$C = \{c_i | c_i \triangleright^+ c_t\} \tag{3.1}$$

where  $c_i, c_t \in \mathcal{C}$  is the **test cluster**  $C$  of the class under test  $c_t$ .



**Definition 3.2.2.** Let  $M_c$  be the set of public methods of class  $c$ . The **set of candidate methods**  $M_C$  for test cluster  $C$  is the union set

$$M_C = \bigcup_{c \in C} M_c \quad (3.2)$$

Now a test sequence can be defined as follows:

**Definition 3.2.3.** A **test sequence**  $T = (m_1, m_2, m_3, \dots, m_n)$  is an ordered set of methods where  $m_i \in M_C$ ,  $i = 1, 2, \dots, n$ , and  $M_C$  be the set of candidate methods of the considered test cluster  $C$ . The **test sequence space**  $\Theta_C$  is the space of all test sequences from test cluster  $C$ .

Since a test sequence finally focuses on the examination of a particular method, at least one method call  $m_t \in T$  must refer to that method. In general,  $m_t$  is the last element of the sequence.

An arbitrary sequence of method calls is not necessarily executable. For instance, if the first method of a method call sequence is a non-static method, it cannot be executed since no target object exists for that method. Therefore, it is necessary to distinguish between *feasible test sequences*, that are, test sequences that can be executed, and *infeasible test sequences*, that are, test sequences that cannot be executed.

Feasible test sequences exhibit the properties of both *formal feasibility* and *dynamic feasibility*. Infeasible test sequences do not exhibit the property of *dynamic feasibility*. While dynamic feasibility is a sufficient property, formal feasibility is a necessary property for a test sequence to be executable. These properties are discussed and defined in the following.

Given an arbitrary test sequence, a static analysis can decide whether it is not executable, for instance, due to contained instance methods for which no preceding object-creating methods are present. This decision is based on the satisfaction of *method call dependences* and can be made without the need to attempt executing the sequence. The property exploited for the decision is the formal feasibility, which is defined as follows:

**Definition 3.2.4.** Let  $r : M_C \rightarrow C \cup \{\otimes\}$  (where  $\otimes$  is the “no class” element) be a function which assigns each method its return type (where a method returning no value (*void*) or a primitive value is assigned  $\otimes$ ). Furthermore, let  $p : M \rightarrow \mathcal{P}(C)$  (where  $\mathcal{P}(C)$  is the power set of  $C$ ) be a function which assigns each method the set of required class parameter types. Finally, let  $t : M \rightarrow C \cup \otimes$  be a function which assigns each method its declaring class if it is an instance method, or  $\otimes$ , if it is a static method or a constructor. Test sequence  $T = (m_1, m_2, \dots, m_j)$  is **formally feasible** if

$$\forall i \in 1, 2, \dots, j : \forall c \in \{t(m_i)\} \setminus \{\otimes\} : \exists k < i : c = r(m_k) \quad (3.3)$$

otherwise, it is **formally infeasible**. The **space of formally feasible test sequences** for test cluster  $C$  is  $\Theta_C^{Ff}$ . The **space of formally infeasible test sequences** for test cluster  $C$  is  $\Theta_C^{If}$ . It holds  $\Theta_C^{Ff} \subseteq \Theta_C$ ,  $\Theta_C^{If} \subseteq \Theta_C$ , and  $\Theta_C^{Ff} \cap \Theta_C^{If} = \{\}$ .

A formally feasible test sequence is not necessarily executable. Formal feasibility assumes that each method which is declared to return an instance of a particular class, actually returns an instance and does not return the `null` reference. It cannot be statically decided whether or not a method returns an instance of the type that it declares. To account for this, the property of *dynamic feasibility* is defined as follows:

**Definition 3.2.5.** Let  $M_C$ ,  $t$ , and  $p$  be defined as in Definition 3.2.4. Furthermore, let  $\iota : M_C \rightarrow C \cup \otimes$  be a function that assigns each method of a test sequence either the class of the returned instance, if it returns an instance during execution, or the  $\otimes$  element, if it returns the `null` reference during execution. Test sequence  $T = (m_1, m_2, \dots, m_j)$  is **dynamically feasible** if

$$\forall i \in 1, 2, \dots, j : \forall c \in \{t(m_i)\} \setminus \{\otimes\} : \exists k < i : c = \iota(m_k) \quad (3.4)$$

If it is formally feasible but not dynamically feasible, it is **dynamically infeasible**. The **space of dynamically feasible test sequences** for test cluster  $C$  is  $\Theta_C^{F_d}$ . The **space of dynamically infeasible test sequences** for test cluster  $C$  is  $\Theta_C^{I_d}$ . It holds  $\Theta_C^{F_d} \subseteq \Theta_C^{F_f}$ ,  $\Theta_C^{I_d} \subseteq \Theta_C^{I_f}$ , and  $\Theta_C^{F_d} \cap \Theta_C^{I_d} = \{\}$ .

This thesis makes the assumption that only test sequences which are dynamically feasible are actually executable. Sequences that are not dynamically feasible are considered to be not executable.

The definitions of test sequence feasibility refer to syntactical properties and not to semantical properties. To distinguish test sequences that respect all method preconditions and therefore comply with the specification of the involved classes from those that violate some method preconditions, the notion of *legality* is used.

A representation of test sequences can refer to static properties of test sequences only. Therefore, formal feasibility will be considered for representation design in the following, while dynamic feasibility will be considered for objective function design later. The term *feasibility* refers to formal feasibility in the following two sections.

### 3.3 Representation by Method Call Trees and Number Sequences

Wappler (2004) elaborates on a representation of test sequences by number sequences (cf. Section 2.2.2 on page 28). A drawback of this representation is that it allows the generation of infeasible test sequences, because it encodes the whole space of test sequences:  $\Phi = \Theta_C$ . The reason that infeasible test sequences can be generated when using the representation of number sequences is that the call dependences which exist among the methods of the test cluster classes are not taken into account by neither this representation nor the evolutionary operators working on it. To cope with this shortcoming, Wappler integrates additional penalty functions into the definition of the objective functions. Some of the suggested penalty functions are indirectly based on the number of disregarded call dependences. Although the objective function is defined for

all points in the genotype search space due to the inclusion of a penalty function, the evolutionary search is not efficient since the search space contains regions of infeasible individuals, the number and size of which depend on the number of call dependences of the methods of the test cluster classes. As a result, the evolutionary search must first discover the regions containing feasible test sequences before it can focus on exploring these regions in more depth in order to find a covering test sequence. Even worse, the evolutionary operators for mutating and crossing over number sequences imply a neighborhood structure of the phenotype search space where a slight change of a genotype individual might suddenly change the feasibility of the decoded test sequence (the phenotype individual), hence resulting in a significant change of the objective value. For instance, assume that one candidate test sequence is a constructor call of class  $c$ , followed by method calls that are applicable to  $c$  (that is, they require an instance of class  $c$ ). Suppose that this sequence achieves a relatively good objective value since it approaches the test goal relatively close. Now it may happen that during mutation, the number representing the constructor call (the first sequence element) is changed so that the decoded sequence has as first element whatever non-constructor call. Then, the new test sequence is infeasible, requiring the penalty function to be applied which results in a relatively bad objective value. This means in conclusion, that two neighboring individuals (where neighborhood is defined in terms of the mutation operator) may be assigned very unequal objective values. As a result, the objective function landscape for the search contains many discontinuities, making the search inefficient, and even ineffective if it does not succeed in exploring the right search space regions before one of the specified termination criteria applies.

It can be learned from the work of Wappler that, in order to effectively cope with test sequence infeasibility and consequently allow an efficient evolutionary search, the underlying representation must be designed so as to the call dependences are accounted for, thereby preventing infeasible test sequences from being generated or at least minimizing the probability of their generation. Consequently, the desire is to define genotype space  $\Gamma$  so that  $\delta(\Gamma) = \Theta^F$ .

Before designing a representation, it is necessary to understand the call dependences and how they affect test sequence feasibility. The next section deals with these call dependences and introduces a graphical representation: the method call dependence graph.

### 3.3.1 The Method Call Dependence Graph

Calling an instance method of class  $c$  requires that an instance of class  $c$  is available. This thesis refers to this fundamental principle as *call dependence* among the methods of a set of classes.

**Definition 3.3.1.** *Let  $M_C$  be the set of candidate methods of test cluster  $C$ . Furthermore, let  $r$ ,  $t$ , and  $p$  be defined as in Definition 3.2.4. Method  $m_i \in M_C$  is **call-dependent***

**on method**  $m_j \in M_C$ , denoted by  $m_i \xrightarrow{d} m_j$ , iff

$$m_i \xrightarrow{d} m_j \Leftrightarrow r(m_j) \in \{t(m_i)\} \cup p(m_i) \quad (3.5)$$

This means,  $m_i \xrightarrow{d} m_j$  if the return type of  $m_j$  is compatible to either the target object type of  $m_i$  or to one of the parameter types of  $m_i$ . The case  $r(m_j) = \otimes$  is excluded. Compatible return type means compatible in terms of polymorphism. Note that, according to this definition a method can be call-dependent on itself. Usually, a method is call-dependent on more than one method, depending on the number and types of its parameters. Also note that the definition of call dependence is very strict with respect to the parameter types; a call dependence of a method does not only refer to methods that can provide a target object, but also refers to methods that can provide parameter objects.

It is also reasonable to define call dependence of methods on classes:

**Definition 3.3.2.** Let  $M_C$  be the set of candidate methods of test cluster  $C$ . Furthermore, let both  $t$  and  $p$  be defined as in Definition 3.2.4. Method  $m \in M_C$  is **call-dependent on class**  $c \in C$ , denoted by  $m \xrightarrow{d} c$ ,

$$m \xrightarrow{d} c \Leftrightarrow c \in \{t(m_i)\} \cup p(m_i) \quad (3.6)$$

This means, method  $m$  requires an instance of class  $c$  when being called. A method can be call-dependent on the same class more than once (one dependence for each required instance of the class).

Additionally, the notion of *call contribution* is needed in order to formally define method call dependence graphs later. Both methods and classes can be *call-contributing*.

**Definition 3.3.3.** Let  $M_C$  be the set of candidate methods of test cluster  $C$ . Method  $m_i \in M_C$  is **call-contributing to method**  $m_j \in M_C$ , denoted by  $m_i \xrightarrow{c} m_j$ , iff  $m_j \xrightarrow{d} m_i$ .

Hence, call contribution with respect to methods is the inversion of call dependence; it holds:

$$m_i \xrightarrow{c} m_j \Leftrightarrow m_j \xrightarrow{d} m_i \quad (3.7)$$

Call contribution of classes to methods is to be defined as well:

**Definition 3.3.4.** Let  $M_C$  be the set of candidate methods of test cluster  $C$ . Class  $c \in C$  is **call-contributing to method**  $m \in M_C$ , denoted by  $c \xrightarrow{c} m$ , iff  $m \xrightarrow{d} c$ .

Hence, call contribution with respect to classes is the inversion of call dependence with respect to classes; it holds:

$$m \xrightarrow{d} c \Leftrightarrow c \xrightarrow{c} m \quad (3.8)$$

The *method call dependence* graph for the set of candidate methods  $M_C$  of test cluster  $C$  is defined as follows:

**Definition 3.3.5.** The *method call dependence graph*  $G_{M_C}^d$  is the tuple  $(N_M, N_C, E_{N_M \rightarrow N_C}, E_{N_C \rightarrow N_M})$ , which defines a bipartite, directed graph with two types of nodes: the method nodes  $N_M = \{n_{m_1}, n_{m_2}, \dots, n_{m_o}\}$ , each of which represents a method  $m \in M_C$ , and the type nodes  $N_C = \{n_{c_1}, n_{c_2}, \dots, n_{c_p}\}$ , each of which represents a class  $c \in C$ . The edges  $E_{N_M \rightarrow N_C} \subseteq M \times C$  (those starting at a method node and going to a type node) and  $E_{N_C \rightarrow N_M} \subseteq C \times M$  (those starting at a type node and going to a method node) have different semantics:

Two nodes  $n_i \in N_M$  and  $n_j \in N_C$  are connected by an edge  $e \in E_{N_M \rightarrow N_C}$ , iff the method represented by  $n_i$  is call-dependent on the type represented by  $n_j$ . Two nodes  $n_k \in N_C$  and  $n_l \in N_M$  are connected by an edge  $e \in E_{N_C \rightarrow N_M}$ , iff the method represented by  $n_l$  is call-contributing to the class represented by  $n_k$ .

It holds:

$$(n_i, n_j) \in E_{N_M \rightarrow N_C} \subseteq M_C \times C \wedge (n_k, n_l) \in E_{N_C \rightarrow N_M} \subseteq C \times M_C \Leftrightarrow \mu(m_i) \xrightarrow{d} \mu(m_l) \quad (3.9)$$

where  $\mu : N_M \rightarrow M_C$  is a function which assigns each method node the method that it represents.

The method call dependence graph has the following implications:

1. If the method represented by the method node  $n_i$  is to be invoked, at least one instance of each class represented by a node which is connected by an edge to  $n_i$ , must be available. (One can accept unsatisfied call dependences for the parameter objects if the `null` reference is feasible.) This implication corresponds to formal test sequence feasibility as defined in Section 2.1.3 on page 10. The edges starting at method nodes have an *AND* semantic.
2. An instance of the type represented by node  $n_j$  can be obtained by calling *one* of the methods which are represented by a node which is connected by an edge to  $n_j$ . Hence, the edges starting at type nodes have an *XOR* semantic.

Figure 3.2 on the following page shows the method call dependence graph for the test cluster  $C = \{Object, Integer, IntegerRange\}$ . (The source code of the classes is shown in Listing A.1, and Listing A.2, respectively.) If a method is call-dependent on a class more than once, only one dependence edge is shown, along with a number that indicates the number of call dependences. The method call dependence graph for a given test cluster can be constructed completely automatically.

The method call dependence graph  $G^d$  for a test cluster gives exact information as to which methods are relevant for a particular method call to become feasible. This means, if for instance method `IntegerRange.growUp()` should be invoked, the graph tells which other methods must be invoked first for this method to become callable. For the example, this is either `IntegerRange.combine(IntegerRange, IntegerRange)`, or `IntegerRange(Integer, Integer)` (which requires further methods to be called in advance). The graph explicitly models the call dependences among the methods of the test cluster classes.

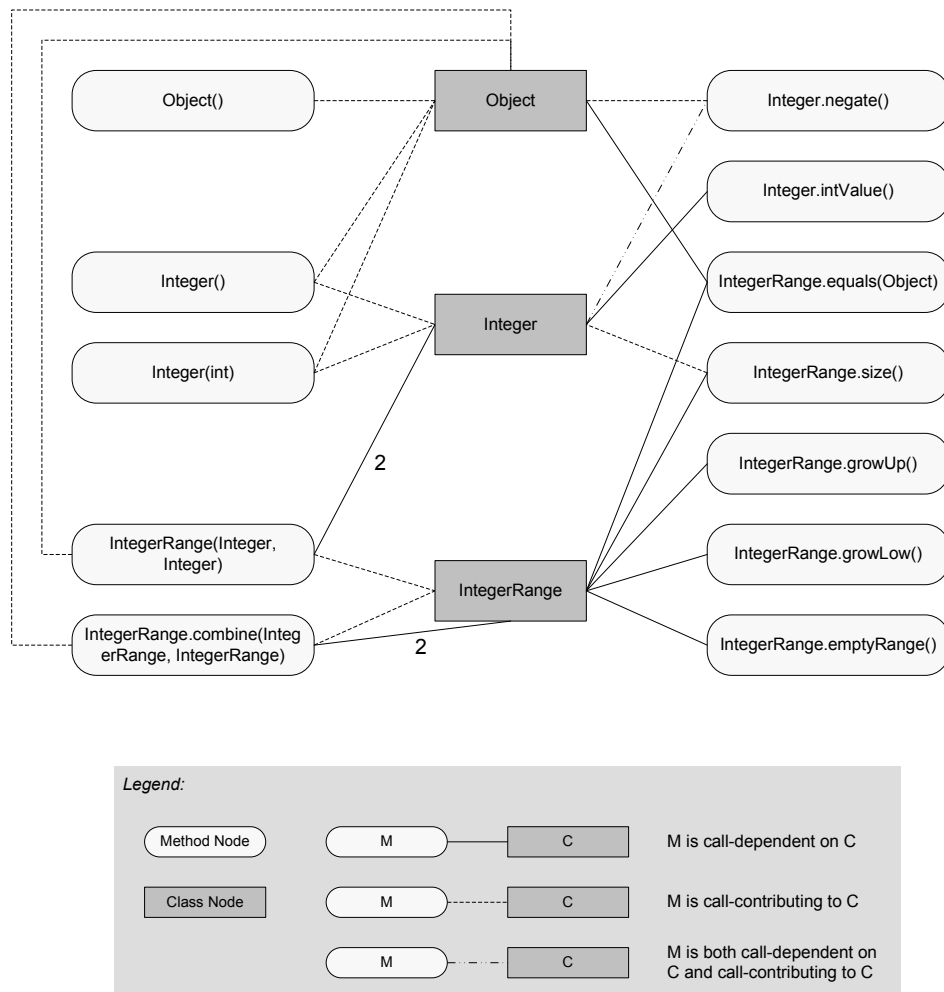


Figure 3.2: Method call dependence graph

From a test sequence generation point of view, the method call dependence graph enables the creation of feasible test sequences. Generating a feasible test sequence can be accomplished by traversing the graph in a particular way. Assume that a test sequence is to be found that covers a test goal that belongs to method  $m_i$ . Then, the graph traversal starts at the method node  $n_i$  which represents method  $m_i$ . Now all edges must be explored in order to make the preparations that  $m_i$  can be called. For instance, by a depth-first traversal algorithm, all edges that start at  $n_i$  and go to class nodes  $n_j, n_k, \dots, n_l$  must be visited. Note that all edges *must* be explored; there is no choice as to the selection of the edges (recall the *AND* semantics of these edges), but only on the *order* of their selection. When class node  $n_j$  is visited, exactly *one* edge must be traversed that starts at  $n_j$  and goes to method node  $n_p$ . Note that there is a degree of freedom concerning the *choice* of the edge and consequently the method node  $n_p$  (recall the *XOR* semantics of these edges). Also note that one such edge resolves the call dependence, it suffices to traverse one edge, even though the traversal of several of such edges would not affect the feasibility of the resulting test sequence (however, this sequence would contain more elements than needed). Graph traversal stops once the exploration of all branches has reached nodes which do not possess consecutive edges (these nodes represent constructors or static methods, respectively). During traversal, all method nodes encountered are put into a list. After traversal, the method nodes are replaced by the methods they represent. By doing so, a feasible test sequence is obtained.

The algorithm just described has to deal with two degrees of freedom: the first is the order of the selection of edges going to class nodes to be visited next, and the second is the selection of an edge going to a method node when a class node is visited. It is assumed to be reasonable to deal with the former by applying a deterministic selection strategy, such as depth-first, since the influence of the ordering on the semantics of the resulting test sequence is supposed to be low: the ordering implies *at which position* a subsequence appears. However, for the latter the influence on the resulting test sequence is supposed to be very high since the selection implies *which concrete methods* will appear in the sequence. There are several ways to deal with this second degree of freedom: for instance, exhaustive search can be applied, meaning that all possible selections are regarded which will result in a high number of graph traversals and consequently in a high number of considered test sequences. Yet, this thesis will follow a heuristic way.

A decision made concerning the edge to be followed from a class node to a method node strongly impacts all subsequent decisions. For instance, imagine a class node from which two edges go to two different method nodes. One of these two method nodes has no outgoing edges, while the other method node has an outgoing edge. This edge goes to a class node which again has two outgoing edges to method nodes. If during traversal the algorithm decides to select the edge going to the first method node, no subsequent decision is to be made. In contrast, if the decision is made in favor of the edge going to the second method node, an additional decision will be required when selecting the edge to follow from the subsequent class node. As a result, each decision strongly affects all subsequent decisions.

From a representation-designing point of view, the method call dependence graph implicitly defines part of the overall test sequence search space (the phenotype search space). Note that the method call dependence graph accounts for object-creating methods only, additional methods that may change the state of an object are not (yet) dealt with. A candidate solution, which is a point in this search space, can be represented by the accumulation of all decisions made concerning the class edges to follow. Such an accumulation possesses strong internal dependencies, due to the impact of a decision on the subsequent decision as just described. Also, the number of decisions typically varies for different traversals. The question to be answered is how to encode the mechanism of decision making and which evolutionary operators to use for mutation and crossover that work on such an encoding. The idea of this work is to represent the information of the decisions made by *method call trees*.

### 3.3.2 Method Call Trees

A method call tree consists of method nodes, each of which represents a method which will later appear in the decoded test sequence. In formal terms, a method call tree is an acyclic subgraph of the method call dependence graph. It is rooted; the root node represents the method under test.

**Definition 3.3.6.** A method call tree  $\Psi$ , defined by the tuple  $(N_M, E)$ , is an acyclic directed graph, where  $N_M$  is the set of nodes representing the test cluster methods, and  $E \subseteq N_M \times N_M$  is the set of the edges connecting the method nodes.

Each edge of a method call tree represents a decision made in favor of a particular call-contributing method (if no decision is to be made since only one call-contributing method is present, the edge to this only method represents the inevitable “decision”).

The method call tree shown in Figure 3.3 represents a (incomplete) test sequence relating to the method call dependence tree from Figure 3.2 on page 60. The tree

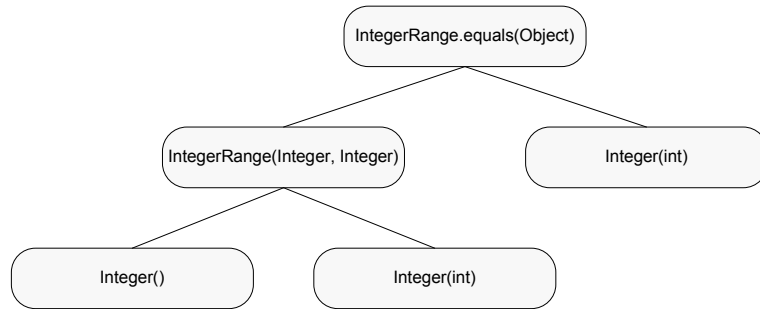


Figure 3.3: Method call tree

explicitly shows the decisions made when traversing the method call dependence tree. The root node represents the method under test, in this case method `equals(Object)` of class `IntegerRange`. In terms of the method call dependence graph, two call-dependence



edges start at the node representing the `equals` method which must be followed. The first edge goes to class node `IntegerRange`, modeling the dependence on the target object for the call. From that class node, two edges go to the method node representing the constructor of class `IntegerRange` and to the method node representing the only static method of that class. The tree shows that the decision was made in favor of the constructor (left subtree). In turn, two call-dependence edges start from that constructor node which must be satisfied. Both go to class node `Integer`. This implies that two decisions are required to choose between the default constructor of class `Integer` and the constructor requiring an `int` argument. The tree shows that for the first parameter of the constructor `IntegerRange(Integer, Integer)` the default constructor was chosen while for the second parameter, the alternative constructor was chosen. No further decisions are to be made concerning the left subtree – graph traversal has reached nodes from which no further call dependence edges start. The call dependence edge starting at method node `IntegerRange.equals(Object)` and going to class node `Object` must also be traversed. Several call-contributing edges start at `Object`; as the tree shows, the one going to method node `Integer(int)` was chosen.

By this interpretation, a particular method call tree represents a particular set of decisions made with respect to the call-contributing nodes of the corresponding method call dependence tree. Building method call trees in this way ensures that the test sequences, which are obtained by linearizing the trees using in-order tree linearization, satisfy the call dependences among the methods, meaning that the required instances are created for all methods of the sequence to become executable.

By doing so, test sequences including object-creating methods can be obtained only. However, some test goals require the participating instances to be in particular states. Therefore, it must be possible to also include state-setting methods into the method call trees. Since the signature of a method does not indicate whether it changes the state of the target object, all instance methods (non-static methods) are assumed to be potentially state-changing and will be considered in the following.

Factoring state-changing methods into the method call trees is accomplished by factoring them into the corresponding method call dependence graph, leaving the algorithm to derive method call trees from the graph unchanged. This is simply possible by realizing the convention that *each instance method of class `c` is call-contributing to `c`*. This means that each instance method of a class implicitly returns an instance of that class. This convention is reasonable since each instance method, once it has been called, can “pass” the target object it used by itself to subsequent instance method calls.

The convention does not change the definition of the method call dependence graph; rather, it implies additional call-contributing edges. Figure 3.4 on the next page shows the method call dependence graph from Figure 3.2 on page 60, extended by the appropriate additional edges. For instance, the graph now tells that method `Integer.intValue()` is call-contributing to class `Integer`, meaning that it can be used to acquire an instance of class `Integer`. This statement is correct because when traversing the graph further from node `Integer.intValue()`, inevitably a method that creates an instance of class `Integer` will be encountered eventually. Therefore, method `Integer.intValue()` is at

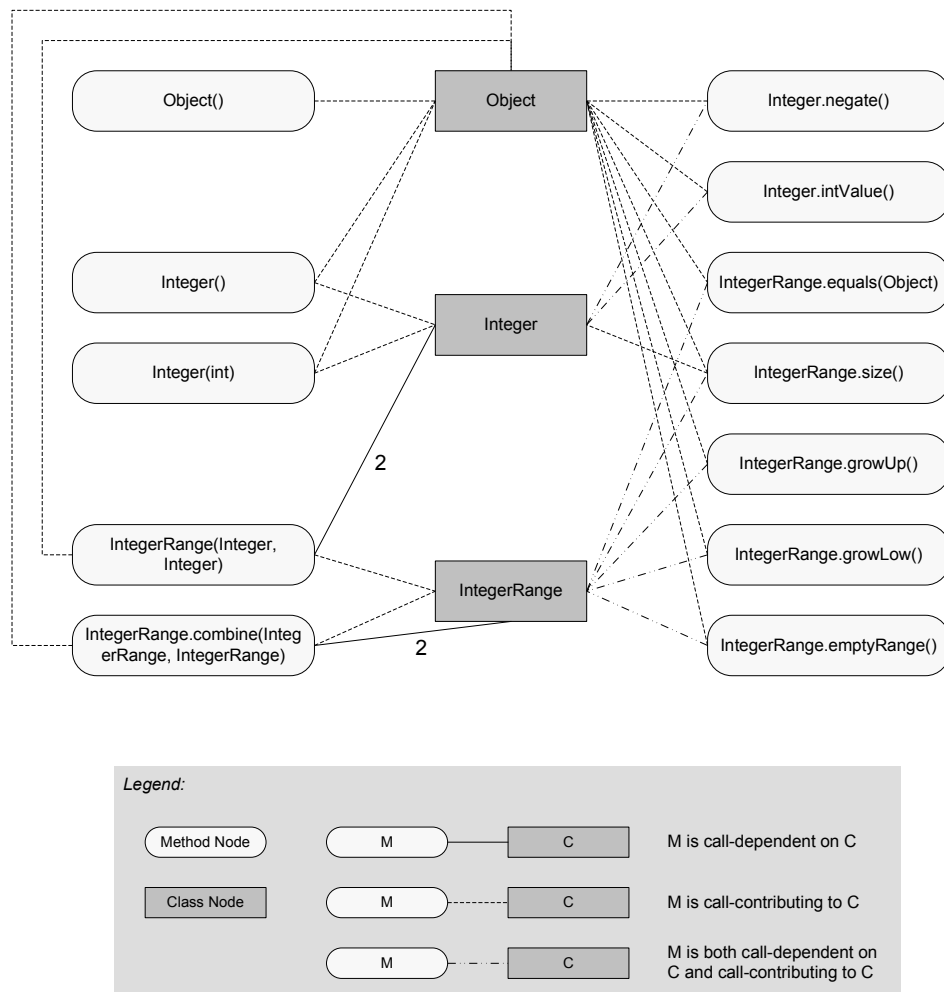


Figure 3.4: Method call dependence graph with additional call-contributing edges

least indirectly call-contributing to class `Integer`.

An example of a method call tree derived from the method call dependence tree with the additional edges is shown in Figure 3.5. In the figure, the edges are labeled with

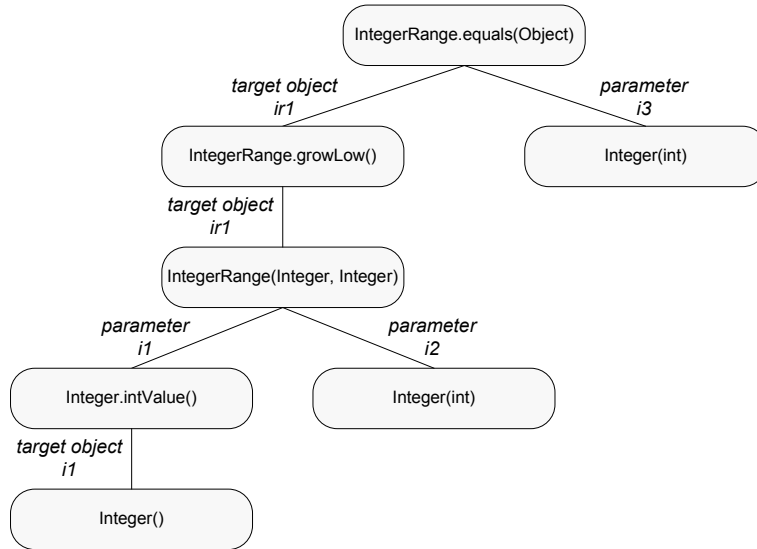


Figure 3.5: Method call tree containing state-changing methods; with annotated instances and their roles

identifiers for the instances that can be imagined to be passed among the methods represented by the nodes of the tree. Thereby, an instance can be used in the role of either the target object or a parameter object. The tree reads best when starting from the leftmost leaf node: method `Integer()` creates instance *i1* of the `Integer` type. This instance is then used as the target object for method `Integer.intValue()`. Note that the implementation logic of this method does not actually change the state of the passed target object; however, it basically may. Then, the same instance *i1* (in a potentially new state) is passed on to the constructor `IntegerRange(Integer, Integer)` as the first parameter object. The second parameter object *i2* is obtained by the constructor `Integer(int)`. Note that at this point, primitive arguments are not yet dealt with. The instance *ir1*, created by constructor `IntegerRange(Integer, Integer)`, is then passed to method `IntegerRange.growLow()`, which actually changes the state of the instance. Finally, this instance is further passed on to the actual method under test, `IntegerRange.equals(Object)`, to be used as the target object. The formally required parameter is obtained by method `Integer(int)`, which delivers instance *i3*.

Listing 3.1 shows the linearization of the method call tree from Figure 3.5 where the appropriate instance identifiers are used. Primitive arguments and primitive return values are not accounted for by the method call dependence graph, which leads to test sequences that may be incomplete, as it is the case in Listing 3.1. There, the sequence lacks two concrete integer values in line 3 and line 6, respectively, which are required as arguments for the constructors. Primitive arguments will be dealt with in the following

Listing 3.1: Linearized method call tree

---

```

1 Integer i1 = new Integer();
2 i1.intValue();
3 Integer i2 = new Integer(int);
4 IntegerRange ir1 = new IntegerRange(i1, i2);
5 ir1.growLow();
6 Integer ir3 = new Integer(int);
7 ir1.equals(ir3);

```

---

in conjunction with the issue of *object reuse*.

To summarize the concepts of the method call dependence graph and the method call trees it can be said that these notions define a genotype search space which encodes fragmental test sequences – *test sequence fragments* – which are test sequences with undefined primitive arguments. Due to the tree structure of the points in this search space, evolutionary operators for exploring this space can easily be defined as will be discussed in Section 3.3.4 on page 69. In turn, an evolutionary algorithm – more precisely, a genetic programming algorithm – searching the space of test sequence fragments can be defined for each test cluster by considering the corresponding method call dependence graph. Thereby, the well-established genetic programming operators for tree building, tree crossover, and tree mutation, as described in Section 2.3.3 on page 46 can be applied to the evolution of method call trees. The appropriate configuration of the search algorithm is described in Section 3.3.4 on page 69. The variation operators used ensure that only feasible test sequence fragments are explored.

### 3.3.3 Primitive Arguments and Parameter Object Selectors

In an early phase of the research on which this thesis reports it was considered reasonable to formulate the search for a covering test sequence by recognizing two “dimensions” of a test sequence: the first dimension is the method call sequence, and the second are the corresponding arguments. (As will be shown later, this view increases the complexity of the search significantly, which might result in an inefficient and even ineffective overall evolutionary search). Consequently, a hybrid evolutionary algorithm is to be constructed which primarily searches the space of fragmental test sequences, and for each test sequence fragment encountered, it searches the corresponding parameter space, defined by the concrete test sequence fragment. Given the method call tree from Figure 3.5 on the preceding page (which is equivalent to the test sequence fragment from Listing 3.1), the parameter space is defined by the two integer arguments. Therefore, an evolutionary search exploring this two-dimensional parameter space is carried out. For each explored point, the test sequence fragment is completed by inserting the integer arguments appropriately. Then, the objective value is calculated by assessing the execution of the completed test sequence.

However, it is beneficial to define the parameter space not only in terms of the missing

*primitive* arguments, but also by the (already present) non-primitive arguments. The benefit of this is due to the support of *object reuse* which would not be possible otherwise. Object reuse in this context means that one instance can be passed to multiple methods as an argument, or multiple times to the same method as arguments. The method call dependence graph, as defined in Definition 3.3.5, does not allow for object reuse. For instance, generating a test sequence that evaluates the first decision of method `equals(Object)` of class `IntegerRange` to `true`, is not possible when strictly adhering to the method call dependence graph. Figure 3.5 on page 65 exposes this deficiency: it is not possible to pass the same object as both target object *and* parameter object to this method in order to make the address comparison of the first decision evaluate to `true`. Therefore, the strictness of the edges concerning the instances passed must be loosened. Two alternative approaches suggest themselves for this loosening: (1) the definition of the method call trees and the traversal algorithm of the method call dependence graph is changed so that it allows creating edges to method nodes which are already part of the tree, or (2) additional informational items are introduced which make it possible to change the assignment of the instances to the formal parameters of the methods. The former approach appears attractive at first view. However, when studying the consequences with respect to the evolutionary operators for mutation and crossover, the idea turns out to make the implementation of these operators – and in turn the evolutionary search – much more complicated. For instance, imagine that the method call tree generation algorithm allows creating the tree shown in Figure 3.6. This

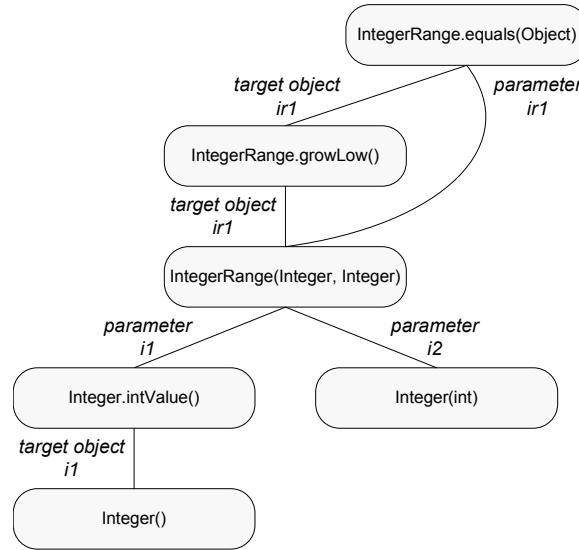


Figure 3.6: Method call tree, generated by loosened tree creation algorithm

tree differs from that of Figure 3.5 on page 65 in that it reuses instance *ir1* as the parameter object when calling method `IntegerRange.equals(Object)`. Now imagine that crossover should occur. Recall that crossover of trees means to exchange randomly selected subtrees. It would be problematic to exchange the subtree rooted by the method

node `IntegerRange.growLow()` with an arbitrary subtree from another tree individual. All edges would be required to be considered and to be relocated, involving additional decisions to be made as to how to connect the existing edges to nodes of the new subtree.

In order to avoid the reformulation of the well-established evolutionary operators and to keep the representation and in turn the evolutionary algorithm simple, the second idea based on additional informational items will be further considered. This idea does not accomplish a loosening of the interpretation of the edges of a method call tree, but rather a loosening of the parameter object assignments during (or even after) tree linearization. Reconsider the test sequence from Listing 3.1. Parameter objects occur in lines 4 and 7. It would now be possible to change the instances that appear as method arguments in order to obtain a new, different test sequence. For instance, for the method call in line 7 (`IntegerRange.equals(i3)`), it can be said which other objects may serve as a parameter object. These are all instances that are created so far and are of the formally required parameter type (which is `Object` in this case). Instances *i1*, *i2*, *i3*, and *ir1* might appear as the parameter object for the method call. Consequently, an additional degree of freedom can be introduced into the sequence by allowing the variation of the parameter objects to be actually passed when calling a particular method. The selection of which instance to be passed can be modeled by an additional variable that will be referred to as *parameter object selector* in the following. For the example sequence, there are three points at which an additional variation of the parameter objects is possible: the two parameter objects in line 4, and the parameter object in line 7. While in line 7, all available instances would be candidates, in line 4, only the instances *i1* and *i2* are candidates since only these are present at the moment of the call and match the required formal parameter type `Integer`. For each of these points, a parameter object selector is defined, leading to three additional variables to be part of the parameter search space. Listing 3.2 demonstrates the idea from an implementation point of view. It shows a modified version of Listing 3.1.

Listing 3.2: Test sequence, augmented by framework methods

---

```

1 Integer i1 = new Integer();
2 ObjectPool.addInstance( i1 );
3 i1.intValue();
4 Integer i2 = new Integer(int);
5 ObjectPool.addInstance( i2 );
6 IntegerRange ir1 = new IntegerRange(
7     ObjectPool.getInstance(Integer.class, int),
8     ObjectPool.getInstance(Integer.class, int)
9 );
10 ObjectPool.addInstance( ir1 );
11 ir1.growLow();
12 Integer ir3 = new Integer(int);
13 ir1.equals( ObjectPool.getInstance(Object.class, int) );

```

---

Note the additional framework method calls referring to class `ObjectPool`. This class maintains all instances that are created during the execution of the test sequence.

Additionally, this pool is consulted if a parameter object is required for a method call. Instead of directly passing a concrete object, the parameter object selector serves as an index for the object pool to identify the actual instance among all available instances of the required type. By doing so, the mechanism of assigning instances to the parameters of the method calls is transformed to the search for suitable object pool indices, that is, for integers. The value ranges of a selector directly depend on the number of elements in the object pool. For instance, the object pool for the parameter object selector for line 7 would include the 4 instances mentioned above, leading to a value range of  $[1, 4]$ . Since the number of instances potentially available for each class at any point in the test sequence can be obtained via a static analysis, the value ranges for the search for appropriate index values can be accurately defined, leading to valid selector values only in most cases.

Introducing object selectors also allows for easy handling of `null` references as parameter objects: the value range of a selector is defined as  $[0, |objectpool|]$ , where  $|objectpool|$  is the size of the appropriate object pool. Then, the value 0 identifies the `null` reference by convention.

In summary, the parameter space for a given test sequence fragment is defined by both the primitive method arguments and the parameter objects selectors for all non-primitive method arguments. A point in this space is a vector of numerical values. As a result, the search in this space can be carried out by a genetic algorithm that works on vectors of numerical values as presented in Section 2.3.2 on page 44. Note again that each candidate test sequence fragment defines its own parameter space.

In the following section, the searches with regards to the two dimensions – sequence space and parameter space – are put together, resulting in the definition of the hybrid evolutionary algorithm *TCGen1*.

### 3.3.4 Test-Sequence-Generating Algorithm TCGen1

Listing A.6 sketches the test-sequence-generating hybrid evolutionary algorithm referred to as *TCGen1*. The algorithm hybridizes a genetic programming algorithm, evolving test sequence fragments, and a genetic algorithm, evolving the required parameter information. While the configuration of the latter is straightforward and occurs in accordance with Section 2.3.2 on page 44, the configuration of the genetic programming algorithm demands a closer look.

As described in Section 2.3.3 on page 46, a strongly-typed genetic programming algorithm requires both the *type set* and the *function set* to be specified. Using a strongly-typed algorithm allows easily reflecting the polymorphic relationships of the classes of the test cluster. Both the type set and the function set can be defined completely automatically based on the method call dependence graph. The type set is directly derived from the class nodes that belong to the method call dependence graph of the test cluster at hand: each class becomes a type of the type set, the inheritance relations are expressed by the appropriate formulation mechanism of the genetic programming algorithm used. For instance, in the case of the genetic programming system ECJ (Wilson, McIntyre and Heywood, 2004), the inheritance relations are specified by defining *set*

*types*. Table 3.1 shows the type set derived from the method call dependence graph from Figure 3.4 on page 64. The function set is derived from the method call dependence

type name	compatible types
IntegerRange	-
Integer	-
Object	IntegerRange, Integer

Table 3.1: Example type set

graph, too. A function in genetic programming terms requires specifying a function *name*, the *parameter types*, and the *return type*. The following strategy is applied in order to obtain the functions that constitute the function set. All call-contributing edges are considered. Each method represented by a method node at which such an edge starts, becomes a function. The name of the method is directly used as the name of the function. The type associated with the class node to which the edge goes becomes the return type of the function. The types associated with the class nodes that are connected to the method node via call-dependence edges, become the parameter types of the function. Note that by doing so, multiple functions can be derived from one method. Also note that redundancy might occur concerning polymorphism. For instance, constructor `Integer()` would be inserted twice, once with return type `Object`, and once with return type `Integer`. If the underlying genetic programming algorithm already cares for polymorphism, the former function is superfluous and can be eliminated from the function set. Table 3.2 shows the function set derived from Figure 3.4 on page 64. As the table shows, primitive arguments and return types are completely ignored. It

function name	parameter types	return type
<code>Object()</code>	-	<code>Object</code>
<code>Integer()</code>	-	<code>Integer</code>
<code>Integer(int)</code>	-	<code>Integer</code>
<code>Integer.intValue()</code>	<code>Integer</code>	<code>Integer</code>
<code>Integer.negate()</code>	<code>Integer</code>	<code>Integer</code>
<code>IntegerRange(Integer,Integer)</code>	<code>Integer</code> , <code>Integer</code>	<code>IntegerRange</code>
<code>IntegerRange.equals(Object)</code>	<code>IntegerRange</code> , <code>Object</code>	<code>IntegerRange</code>
<code>IntegerRange.growLow()</code>	<code>IntegerRange</code>	<code>IntegerRange</code>
<code>IntegerRange.growHigh()</code>	<code>IntegerRange</code>	<code>IntegerRange</code>
<code>IntegerRange.emptyRange()</code>	<code>IntegerRange</code>	<code>IntegerRange</code>
<code>IntegerRange.size()</code>	<code>IntegerRange</code>	<code>IntegerRange</code>
<code>IntegerRange.size()</code>	<code>Integer</code>	<code>IntegerRange</code>
<code>IntegerRange.equals(Object)</code>	<code>IntegerRange</code> , <code>Object</code>	<code>MUT</code>

Table 3.2: Example function set



can also be seen that each instance method requires an additional parameter; the passed instance will be used as the target object for the method call, as described above. The last function is a modified duplicate of method `IntegerRange.equals(Object)`. It is defined using the particular return type `MUT`. Recall that this method contains the test goal at hand for this discussion. When instructing the genetic programming algorithm to create trees with the return type of the root node being `MUT`, it is ensured that the method under test appears at least once in all generated test sequences. Otherwise it may happen that a generated test sequence does not issue a call to the method under test which does not allow calculating the objective value reasonably.

Concerning the evolutionary operators for crossover and mutation, the operators described in Section 2.3.3 on page 46 can be applied, which are subtree crossover, demotion, and mutation. ERC mutation is not applicable since the method call trees do not involve ERCs. The application of the operators preserve formal feasibility, defined in Section 3.2 on page 54. The semantical impact of each operator on the test sequences is well-defined. For instance, demotion of a method call tree corresponds to the insertion of a new method call into the test sequence, including the potential insertion of method calls to create the arguments for this newly inserted method call. Promotion corresponds to the removal of a method, including the removal of all depending method calls. Subtree crossover of two method call trees corresponds to the exchange of subsequences, where each subsequence does not have any direct dependence on other parts of the remaining sequence. Rather, the subsequences are isolated parts of the overall test sequence, intended to create an object in a particular state.

In experiments with four test objects of rather lower complexity, TCGen1 succeeded in creating test suites that achieve full method/decision coverage. However, as already remarked in Section 3.3.3 on page 66, working with two separate search spaces implies a particular complexity of the evolutionary search. Since for each test sequence fragment an individual parameter search is carried out, the worst-case effort  $E$  (in terms of fitness function evaluations) for optimizing a test sequence is

$$E = n_{I_S} \cdot n_{G_S} \times n_{I_P} \cdot n_{G_P} \quad (3.10)$$

where  $n_{I_G}$  is the maximum number of generations for the sequence space search,  $n_{I_S}$  is the number of individuals per generation of the sequence space search,  $n_{G_P}$  is the number of generations for the parameter search, and  $n_{I_P}$  is the number of individuals per generation of the parameter search. This means that potentially a high number of objective function evaluations will occur. A more crucial drawback of the algorithm is that good parameter values will be discarded after the parameter search for a test sequence fragment, requiring to search again from scratch when completing the next test sequence fragment which might be similar to that one just assessed.

Due to these drawbacks, the desire arises to simplify the evolutionary search. Since the workflow of the algorithm directly depends on the underlying representation, the next section discusses an improvement of the representation.

### 3.4 Representation by Extended Method Call Trees

The previous section finished by pointing out the drawbacks of the test-sequence-generating algorithm TCGen1 caused by the separation of sequence space and parameter space. This separation necessitates two different representations, one for test sequence fragments and one for the parameters. These two different representations, in turn, require different evolutionary operators for crossover and mutation and consequently a hybrid search algorithm. This section elaborates on an improved representation which unifies sequence space and parameter space, hence implying a much simpler – and in turn, more efficient – search algorithm.

Without the separation of sequence space and parameter space, all information describing a complete test sequence must be put into one individual. It appears natural to simply have the primitive parameters included in the functions of the function set and let the genetic programming algorithm also evolve them using type-specific mutation operators. However, the parameter information concerning the parameter object assignments must also be encoded in order to deal with the issue of object reuse (as described in Section 3.3.3 on page 66). In the following, at first primitive arguments are dealt with, afterwards, an approach to incorporating the evolution of the parameter object assignments into the genetic programming search is suggested.

#### 3.4.1 Incorporating Parameter Space into Sequence Space

So far, the data types of the primitive method arguments are not regarded as classes, thus they do not appear in the method call dependence graph of a test cluster. One way to incorporate primitive arguments into the test sequences based on a concrete method call dependence graph is to incorporate the primitive types into the method call dependence graph. Then, the definition of call dependence also involves primitive types. Figure 3.7 on the next page shows the method call dependence graph from Figure 3.4 on page 64, augmented by the additional type `int`. Method `Integer(int)` is now call-dependent on class `int`. Actually, the graph should also include a method which is call-contributing to class `int`. However, since primitive values can easily be created randomly without the need to call a test cluster method, there is not need for an `int`-constructing method in the graph. One can think of an additional call-dependence edge going from each primitive type node to a framework method which can generate a value of the corresponding type.

In order to deal with the parameter object assignments, a new, artificial type is introduced: the *selector*. Actually, a selector value is an integer value as discussed in Section 3.3.3 on page 66, the value of which is used to decide which parameter object will be passed as the parameter object if multiple parameter object candidates are available. The new type is used to make the particular interpretation of the integer value of a selector more explicit. An additional call dependence on a selector is established in the method call dependence graph from a method to type `selector` for each class-type parameter of the method. In Figure 3.7 on the facing page, methods `IntegerRange(Integer,Integer)`, `IntegerRange.combine(IntegerRange,IntegerRange)`, and `IntegerRange.equals(-`

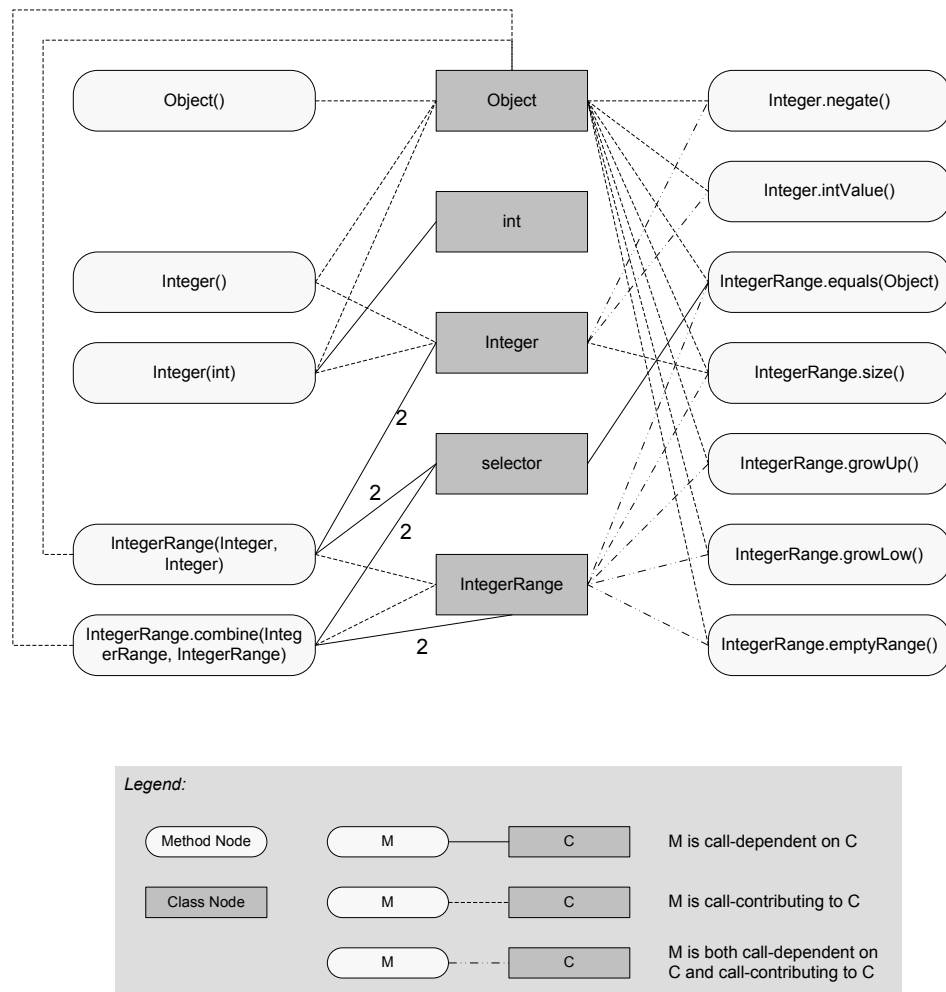


Figure 3.7: Method call dependence graph, augmented by primitive types

Object) exhibit this additional dependence for each of its formal class-type parameters.

With the selector type introduced in this way, the test sequences based on the method call dependence graph are now complete, meaning that they include both the values for primitive arguments as well as the parameter object selector values to address the issue of object reuse. Therefore, no need exists to further explore an additional parameter space since all parameter information is already present. Consequently, a one-level search algorithm can be applied instead of a two-level hybrid search algorithm, as will be shown in the next section. However, a new issue to deal with is the definition of the value ranges for the parameter object selectors. The hybrid evolutionary algorithm is able to derive the value ranges based on the given test sequence fragment. Now, the definition of the value ranges must occur more general, suitable for all test sequences, since no information is initially available as to how many candidates will exist for a particular method call. Several approaches are conceivable to deal with the absence of this information. The next section includes a description of one approach. The section describes the one-level test-sequence-generating evolutionary algorithm *TCGen2*.

### 3.4.2 Test-Sequence-Generating Algorithm TCGen2

Listing A.7 sketches out the test-sequence-generating evolutionary algorithm referred to as *TCGen2*. It consists of one overall search loop, as opposed to *TCGen1* which included a nested search for the appropriate parameters.

The definition of both the type set and the function set occurs in accordance with *TCGen1*. However, due to the additional class nodes of the method call dependence graph, the functions now have additional parameter types, as can be seen in Table 3.3 on the next page, which shows the function set derived from the method call dependence graph from Figure 3.7 on the preceding page. Note that each class-type parameter of the functions is followed by a selector parameter. Since the actual value range of such a selector parameter cannot be defined accurately (as pointed out in Section 3.4.1 on page 72), it is reasonable to allow a sufficiently large range which can be unambiguously mapped to the actual range later when linearizing the method call trees. A suitable mapping can be accomplished by applying a modulo operator to the actual selector value, as will be shown next.

Figure 3.8 on the facing page shows a method call tree that *TCGen2* could have produced. It is similar to the tree in Figure 3.3 on page 62; in addition to that, it also comprises the parameter information (shown by the dark nodes). When linearizing this tree, the selector values are used to decide which actual parameter objects to be passed as arguments. For instance, when method node `IntegerRange(Integer,Integer)` is being processed, the value 54 is used to decide which parameter object from the list of candidate objects (the object pool) to pass. However, at that moment only one instance is available, namely that created by the preceding call to `Integer()`. Therefore, the value 54 will be mapped to the range  $[0, 1]$  using modulo 2, where the value 0 represents the `null` reference and the value 1 the instance produced when `Integer()` was called. In the case of 54, the `null` reference would be chosen ( $54 \bmod 2 = 0$ ). When considering the second parameter of `IntegerRange(Integer,Integer)`, the selector value of 34 is

function name	parameter types	return type
Object()	-	Object
Integer()	-	Integer
Integer(int)	-	Integer, int
Integer.intValue()	Integer	Integer
Integer.negate()	Integer	Integer
IntegerRange(Integer,Integer)	Integer, selector, Integer, selector	IntegerRange
IntegerRange.equals(Object)	IntegerRange, Object, selector	IntegerRange
IntegerRange.growLow()	IntegerRange	IntegerRange
IntegerRange.growHigh()	IntegerRange	IntegerRange
IntegerRange.emptyRange()	IntegerRange	IntegerRange
IntegerRange.size()	IntegerRange	IntegerRange
IntegerRange.size()	Integer	IntegerRange
IntegerRange.equals(Object)	IntegerRange, Object, selector	MUT

Table 3.3: Extended type set

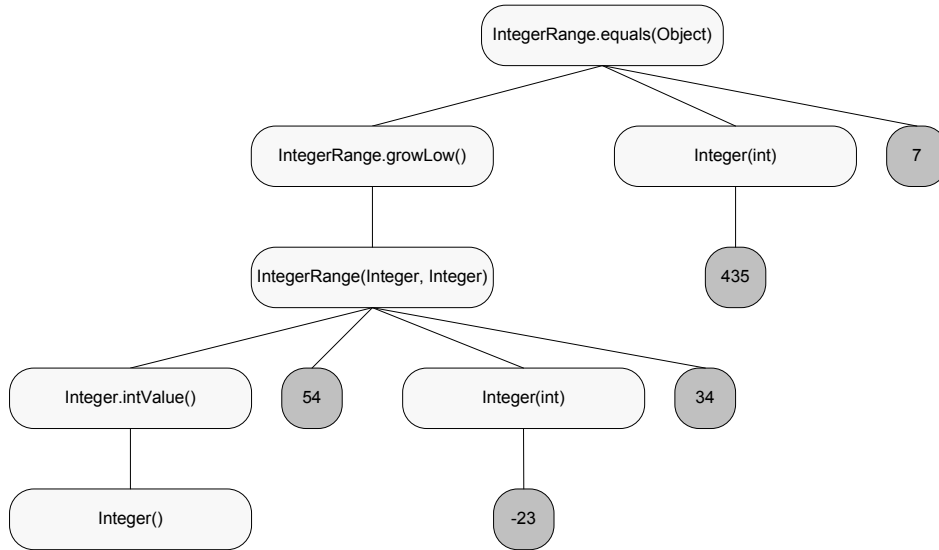


Figure 3.8: Method call tree including parameter information

Listing 3.3: Linearized method call tree

---

```

1 Integer i1 = new Integer();
2 i1.intValue();
3 Integer i2 = new Integer( -23 );
4 IntegerRange ir1 = new IntegerRange( null, i1 );
5 ir1.growLow();
6 Integer i3 = new Integer( 435 );
7 ir1.equals( ir1 );

```

---

used to select among the set of candidates, which are the instances produced by both `Integer()` and `Integer(-23)`, and the `null` reference. As a result, 34 modulo 3, which is 1, selects the instance produced by `Integer()` to be passed as the second argument. Finally, the test sequence shown in Listing 3.3 will be obtained. The example shows, however, that sequences created in this way might contain method calls that do not contribute to achieving the test goal at hand. For instance, both objects `i2` and `i3` are never used. An additional post-processing phase is required in order to make the resulting test sequences more compact by eliminating unneeded method calls. This post-processing is not addressed by this thesis but is an item of future work.

In conclusion, the integration of both primitive types and the selector type into the method call dependence graph allows defining a representation that encodes all information necessary to completely describe a test sequence. The suggested encoding by means of extended method call trees defines the genotype search space  $\Gamma$  so that  $\delta(\Gamma) = \Theta^F$ , meaning that each point in the search space represents a formally feasible test sequence. Both method calls and the corresponding parameters are encoded in one individual. The encoding allows for the application of a genetic programming algorithm with well-established genetic operators.

### 3.5 Objective Function Construction

This section discusses the definition of objective functions for the search for a test sequence that covers a particular branch. The strategy of objective function design for evolutionary structural testing of procedural software is not sufficient for object-oriented software. This is due to runtime exceptions that can occur during test sequence execution. If such an exception occurs, the method under test might not be called, hence preventing the traditional distance metrics control dependence distance and branch distance from being computable with respect to the test goal. To cope with this, an additional distance metric will be introduced. Additionally, other peculiarities of object-oriented software, such as non-public methods, will be dealt with in this section.

At first, Section 3.5.1 on the facing page elaborates on the situations that can arise during test sequence execution. It presents a classification of the possible scenarios. Afterwards, Section 3.5.2 on page 78 to Section 3.5.6 on page 83 address these scenarios in detail and describe an approach to objective function design in the respective cases.

Finally, Section 3.5.7 on page 84 puts the objective functions for the different situations together.

### 3.5.1 Classification of Execution Flows

In order to obtain the objective value for a candidate test sequence, this test sequence is executed using an instrumented version of the source code. The execution process is monitored and the distance between the path taken during execution and the desired test goal is quantified. When trying to execute a candidate test sequence, different situations may arise which require particular treatment on the part of the objective function. For instance, a runtime exception could abort the evaluation of a candidate test sequence, as just described above. Figure 3.9 classifies the possible situations; they

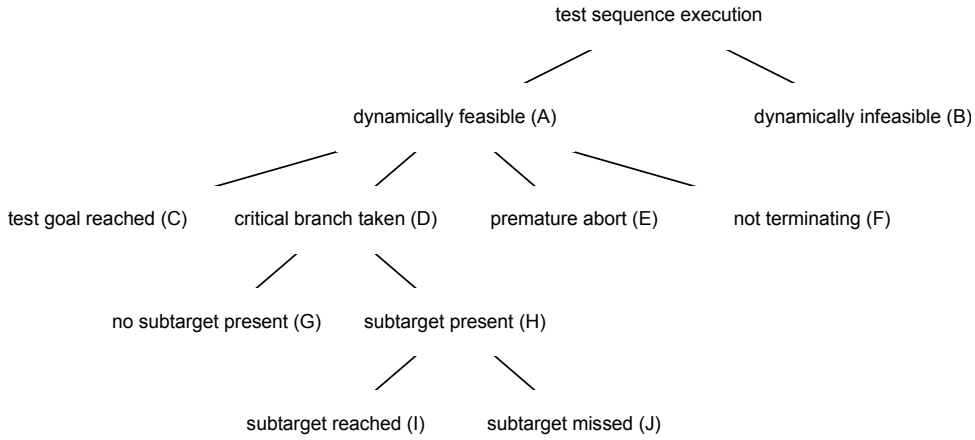


Figure 3.9: Classification of test sequence executions

will be considered in more depth in the next sections. The classification refers to the space of formally feasible test sequences  $\Theta^F$ . Due to the variety of situations it is not possible to give one overall “formula” for a general objective function.

The execution of a test sequence can be dynamically feasible (case A) or dynamically infeasible (case B). An infeasible test sequence cannot be executed, since not all call dependences are satisfied. The representations discussed in both Section 3.3 on page 56 and Section 3.4 on page 72 guarantee that the generated test sequences are formally feasible; however, in some cases methods intended to return an object of a particular type return the `null` reference instead, thus failing to provide an instance to be used as the target object for a following method call. Then, although the test sequence is formally feasible, it is dynamically infeasible. Section 3.5.2 on the next page deals with this case. A feasible test sequence either reaches the desired test goal (case C), or the execution diverged down a critical branch (case D), or the test sequence execution was prematurely aborted before reaching the method under test due to a runtime exception (case E), or in some rare cases the execution does not terminate (case F) due to an endless loop. While case C is perfect and does not require further consideration, cases D, E, and F

demand a closer look. Thereby, case F, caused by an endless loop, indicates either a severe programming error, or (more likely) an unsuitable configuration of a mock object that participates in the test (mock objects are discussed in Section 3.6.1 on page 86). A special penalty applies to test sequences leading to endless loops, as described in Section 3.5.3. The most interesting cases are those that miss the test goal due to a critical branch taken (case D), and those that do not execute the method containing the test goal due to a runtime exception. Section 3.5.4 on the facing page deals with unfavorably evaluated conditions (case D); this case is very similar to conventional evolutionary structural testing of procedural software. Both cases G and H further distinguish between the situation where a *subtarget* is to be reached (case H), and where no subtarget is to be reached (case G). Section 3.5.6 on page 83 discusses the notion of subtargets in the context of non-public methods. Section 3.5.5 on page 80 discusses the evaluation of runtime exceptions that may either prematurely terminate test sequence execution (case E), or cause a critical branch to be taken (case D).

### 3.5.2 Dynamic Test Sequence Infeasibility

Both representations of test sequences described in Section 3.3 on page 56 and Section 3.4 on page 72, respectively, ensure that the generated test sequences are formally feasible. However, when being executed, a formally feasible test sequence may turn out to be not completely executable, due to missing target objects for method calls that appear in the sequence. Consider Listing 3.4 which shows a formally feasible test sequence.

Listing 3.4: Statically feasible but dynamically infeasible test sequence

---

```

1 IntegerRange range = IntegerRangeFactory.getInstance();
2 Integer i = range.getSize();

```

---

However, method `IntegerRangeFactory.getInstance()` – supposed to provide an instance to be used for the following method call – might return the `null` reference. In this case, the method call to `IntegerRange.getSize()` cannot be performed; the sequence is dynamically infeasible.

Dynamic test sequence infeasibility can be dealt with by several approaches. For instance, the number of successfully executed method calls can be considered, in analogy to the penalty functions suggested by Wappler (2004); another penalty metric might be the number of methods that cannot be executed due to the missing instance. However, in this thesis the simple approach is implemented which introduces a constant penalty  $p_{dyn}$  for dynamically infeasible test sequences:

$$\forall T \in \Theta^{I_d} : \omega = p_{dyn} \quad (3.11)$$

### 3.5.3 Endless Loops

An endless loop occurring during test sequence execution must be detected in order to interrupt it, for instance after a timer has expired or another termination criterion applies. Otherwise, the objective function would never return, causing the search algorithm to run



forever. Critical is the formulation of the termination criterion, which must distinguish between test sequences that are so complex that they involve a huge number of (sub-) method calls and require thus a particularly long time, and those which actually cause an endless loop. A mock object can be set up unfavorably in such a manner that an endless loop occurs. An example is a mock class that inherits from type `Enumeration`. Assuming a naive mocking concept, it allows freely configuring the return value of method `hasNext()`. In unfavorable cases, the test sequence configures it to always return `true`, causing a loop that incorporates the return value of `hasNext()` into its termination condition to run endlessly.

A heuristic way to detect endless loops is to examine the execution path produced by the test sequence in question. A large number of repetitions of the same path fragment suggests the presence of an endless loop. However, more pragmatically and easier to realize is the definition of an upper bound on the allowed length of the execution path. Then, if that bound is reached, the objective function assumes that an endless loop, or at least some abnormal behavior, occurs, and terminates the execution of the test sequence. It penalizes the sequence using a constant penalty  $p_{loop}$ .

$$\forall T \in \Theta^L : \omega = p_{loop} \quad (3.12)$$

where  $\Theta^L$  is the space of test sequences implying endless loops.

### 3.5.4 Unfavorably Evaluated Conditions

Section 2.2.2 on page 20 explains how the two distance metrics *control dependence distance* and *branch distance* are used to guide the evolutionary search for test data in the context of procedural software testing. The same metrics are applied for measuring the distance between the execution path produced by a candidate test sequence and the targeted test goal. However, this requires that the method containing the test goal is actually invoked by the test sequence. Then the control flow graph of this method as well as its predicates are used to calculate the respective distances.

For object-oriented software, a modification is required concerning the control node distance: potentially, each statement of a method can throw an exception (Section 3.5.5 on the following page deals with exceptions in more detail); as a result, each node of the control flow graph of the method possesses an additional *exceptional branch*. This branch is connected to either the suitable catch block – if one is present – or is not yet connected since it is not known in which context the method will be called when it is used in a program. Figure 3.10 on the next page shows the control flow graph of method `testCase1` of class `Stack`, the code of which is shown in Listing A.4. The graph shows the additional exceptional branches (indicated by dashed edges). The first exceptional branch, originating from node  $n_1$ , accounts for an exception that may occur when calling the constructor of class `Stack`. When considering this class in isolation, it is not clear to which node to connect this branch. Later, when the class is integrated into an application, it might be connected to a higher-level catch block or to the exit node of the complete program. The second exceptional branch, originating from node

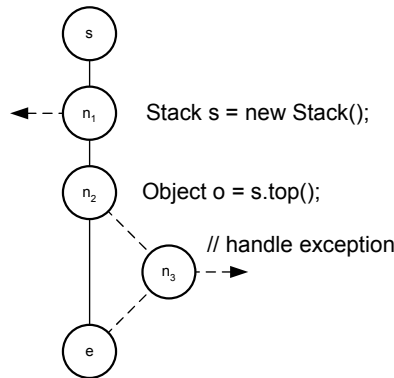


Figure 3.10: Control flow graph including exceptional branches

$n_2$ , is connected to the first statement of the catch block. Control of execution will be transferred to this statement if the declared exception occurs. Note that also an exceptional branch originates from node  $n_3$  since during exception handling a new exception might occur or the caught exception might be propagated to the caller of method `testCase1` via a `throw` statement. These additional branches, which are critical branches in most cases, affect the calculation of the control node distance. Recall that the metric refers to the number of critical nodes on the shortest path from the problem node to the test goal. With the exceptional branches, nearly each node of the control flow graph is a critical node.

The distance functions for calculating the metric branch distance must also be reconsidered. Table 2.1 on page 24 lists the distance functions for the various relational and logical operators. However, object-oriented programming languages usually provide additional, object-orientation-specific operators, such as `instanceof`. For these operators, the distance function used for a Boolean predicate can be applied. For instance, the predicate `(s instanceof String)` can be regarded as `((s instanceof String)==true)`, allowing to apply the Boolean distance function. However, in some cases the guidance by these distance functions is not sufficient. Ideas for defining improved distance functions are outlined in Section 5.3 on page 136.

The definition of the objective function in the case the target has been missed will be provided in the next section after the discussion of runtime exceptions.

### 3.5.5 Runtime Exceptions

Recent object-oriented programming languages, such as Java or C#, support integrated exception handling. Methods can be declared as throwing a particular type of exception. For instance, a method connecting to a network resource might be declared as throwing an exception in the case that the connection cannot be established. The caller of a method having an exception declared must provide additional code that is activated when an exception actually occurs. Listing A.4 shows the source code of class `Stack` which realizes a simple stack that manages instances of class `Object`. Method `Stack.top()`

is declared as throwing an `Exception` if the method is called on an empty stack. In consequence, each caller of that method must ensure that a catch block, compatible to the declared exception, is present which will handle an occurring exception.

However, there are also exceptions which need not be declared because each method is potentially able to throw it. For instance, `NullPointerException` needs not be declared but may be thrown – explicitly or implicitly. Explicitly means that in the code of the method, a new exception instance is created and thrown. Implicitly means that the runtime system generates the exception itself when an inconsistent situation is encountered. In Listing A.4, method `top()` might throw a `NullPointerException` if the class member `elements` would not have been initialized to a valid instance. Then, during the evaluation of the array access, an exception would be thrown by the runtime system.

Undeclared runtime exceptions usually indicate programming errors or situations that have not been foreseen. Thereby, a programming error is, for instance, when a method is called with arguments that are not valid for the method (the arguments violate implicit method preconditions). For instance, the exception type `IllegalArgumentException` is used to indicate that a method has been called with unsuitable arguments. With respect to evolutionary test sequence generation, where argument values are created by random, it happens that the randomly generated arguments are not accurate for a method call; an argument may violate a method precondition. For instance, in the case of method `elementAt(int)` of class `Stack` shown in Listing A.4, negative integers would lead to an implicitly thrown `ArrayIndexOutOfBoundsException`. However, if an exception occurs before all methods of a test sequence are completely executed and presuming that the test goal has not been achieved yet, test sequence execution must terminate. Because it is not clear whether or not the system is still in a consistent state now, it is not advisable to continue execution of the test sequence. As a result, it might happen that the method that contains the current test goal has not yet been called. This means in turn that no execution path information is available on which the conventional distance metrics control node distance and branch distance can be calculated.

Nevertheless, the objective function must – irrespective of an exception – provide an objective value which sufficiently guides the evolutionary search in such a way that test sequences emerge which lead to the execution of the method containing the test goal. For doing so, this work suggests that the objective function incorporates the additional distance metric *method call distance*  $d_M$ . This metric corresponds to the number of method calls of the test sequence which could not be executed due to a runtime exception. It applies only if the method containing the test goal has not been executed. It is defined as follows:

**Definition 3.5.1.** *Let  $T = (m_1, m_2, \dots, m_e, \dots, m_n)$  be a test sequence where the call of  $m_e$  produces an exception. Then, the number of not executed methods  $n - e$  is the method call distance for  $T$ .*

Listing 3.5 shows a test sequence intended to cover a code element of method `IntegerRange.equals(Object)`.

Listing 3.5: Exceptional test sequence

---

```

1 Integer i1 = new Integer(0);
2 Integer i2 = new Integer(-33);
3 IntegerRange ir1 = new IntegerRange(i1, i2);
4 IntegerRange ir2 = ir1.clone();
5 ir1.equals(ir2);

```

---

This sequence raises a runtime exception in line 3 where unsuitable arguments are passed to the constructor of class **IntegerRange**. In this case,  $d_M = 2$ , since two method calls could not be executed because of an exception.

One could think of a normalization of the method call distance, as it occurs with the branch distance (which is normalized to  $[0, 1)$ ). So it appears to be intuitively reasonable to divide the number of unexecuted methods by the total number of method calls present. However, this normalization has severe drawbacks. Imagine that instead of the two **Integer** instances created by the test sequence in Listing 3.5, some more **Integer** instances are created. In turn, the exception would occur later, leading to a lower normalized value of  $d_M$ . However, the situation is actually the same – this extended test sequence fails to reach the final method call in equal measure. The evolutionary algorithm would learn that the objective value of a test sequence can be improved by prepending (unnecessary) method calls. At the same time, the algorithm would learn that it is beneficial to remove all the method calls after the method that produced the exception. However, this would be profoundly counterproductive. Therefore, the absolute number of properly executed methods is used, multiplied by an additional weight constant as will be explained next.

In the case that an exception occurred in a method that does not contain the test goal, both metrics control dependence distance and branch distance are calculated with respect to the exit node of the control flow graph of the method which produced the exception. For the test sequence in Listing 3.5 this method is the constructor **IntegerRange(Integer, Integer)**. The first condition of this method, which checks whether the first **Integer** value is lower than the second, would be identified as the problematic condition. Branch distance would be calculated with respect to this condition. This strategy works well for explicit exceptions (since they are usually control-dependent on a predicate for which a distance function can be applied). However, in the case of implicit exceptions, such as a **NullPointerException**, no gradual distance metric can be applied. Then, branch distance is by convention 1, which is the maximum value. To further distinguish an exceptional branch distance from an unexceptional one (if the exception occurred in the last method of the test sequence), an additional weight  $\lambda_C$  is introduced so as to allow for consistent integration of the two distance metrics: the elementary objective function  $w_t$  for test goal  $t$  is then defined as follows:

$$w_t(T) = \lambda_C \cdot d_C(T, t) + d_B(p) \quad (3.13)$$

where  $T$  is the test sequence under evaluation, and  $p$  is the problem node. Branch distance  $d_B$  is either the conventional branch distance (in the case that either no exception occurred at all or, if an exception occurred, it did not occur at the problem

node), or  $d_B = \lambda_C - 1 + d_E$  (in the case that an exception occurred at the problem node), where  $d_E$  is the exception-specific distance. For instance, in the case of an `ArrayIndexOutOfBoundsException`,  $d_E$  is the distance of the invalid index (due to which the exception was raised) to the closest valid one, normalized into the range  $[0, 1)$ . Both the factor  $\lambda_C$  as well as term  $\lambda_C - 1$  are required in order to distinguish between situations where no exception occurred at the problem node and those where an exception occurred. Without this factor, it would not be clear whether the effective “branch distance” refers to the conventional branch distance, or to the distance related to the exception. An exception occurring at the problem node is considered “worse” than execution diverging down the critical branch at the problem node.

Note that this objective function guides the evolutionary search to avoid runtime exceptions. From a testing point of view this strategy appears to be problematic since exceptions indicate situations of particular interest for the tester. But in fact it is not: each evolutionary search addresses an individual test goal with the primary aim to find a covering test sequence, and not test sequences that raise exceptions. Such sequences could be found easily, most of which are, however, not interesting since they violate implicit method preconditions. Finding interesting test sequences that raise particular types of exceptions is the focus of *robustness testing* (Csallner and Smaragdakis, 2004) and is not primarily dealt with during structural testing. However, test sequences which raise interesting exceptions can be stored and provided later to the tester. They are a byproduct of the evolutionary class testing approach.

When the code under test contains catch blocks, test sequences will be sought which cover the code elements of these catch blocks. To this end, the evolutionary search must evolve test sequences that raise a type of exception which is compatible to the type declared to be caught by the catch block under consideration. The objective function must answer the question as to how far a candidate test sequence is away from producing a particular exception. Since in most cases this question cannot be answered accurately, the objective function calculates the metric control dependence distance with respect to the last encountered control node of the try block. Branch distance is then 1 by convention.

The objective function for test sequence  $T$  that causes a runtime exception in a directly called method which does not contain the test goal is defined as follows:

$$\omega_t(T) = \lambda d_M + w_e(T) \quad (3.14)$$

where  $e$  is the exit node of the method in which the exception occurred.

### 3.5.6 Non-Public Methods

Non-public methods (private and protected ones) cannot be directly called, hence calls to them cannot appear in a test sequence. Non-public methods are usually called by the public methods of the class. The basic mechanism for objective function construction for a test goal that belongs to a non-public method is that test sequences that do not indirectly call the non-public method in question are penalized, as opposed to those that include a call.

Before the evolutionary search, a static analysis is carried out which identifies all statements that involve a call to the desired non-public method. These statements will be referred to as *call points* in the following. Sometimes a non-public method  $m_1$  is only called by another non-public method  $m_2$ . Then, in order to provoke a call to  $m_1$ , a call to  $m_2$  must be attained first. In this case, *chains* of call points result.

An individual objective function is constructed for each call point (or chain of call points, respectively). A separate evolutionary search is carried out for each objective function unless a covering test sequence is found or all call points have been tackled without success.

Each call point is considered a *subtarget* of the search. If the test sequence under evaluation misses the subtarget (that is, the non-public method in question is not called), the distance metrics are calculated with respect to that subtarget (that is, the call point). Otherwise, if the subtarget is reached, the distance metrics are calculated with respect to the actual test goal. In the former case, a constant penalty  $p_s$  will be added to the objective value.

### 3.5.7 Putting it all Together

Figure 3.11 shows the classification of possible scenarios that may occur when attempting to execute a test sequence, annotated with the objective functions that apply in the

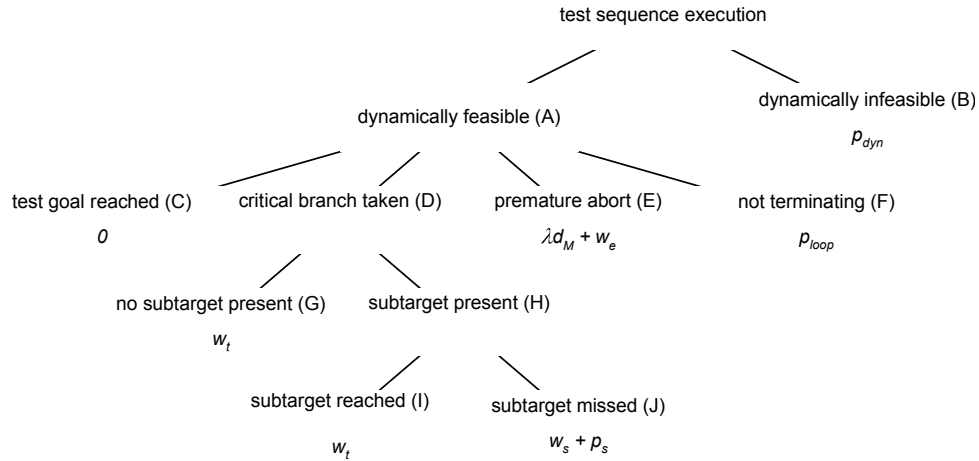


Figure 3.11: Objective functions for the different situations

cases. The overall objective function  $\omega_t(T)$  for test goal  $t$ , where  $T$  is the test sequence to evaluate, refers to the elementary objective function  $w$  in some cases ( $w$  is defined in Equation 3.13). Note that in case E,  $w$  refers to the exit node  $e$  of the function in which the exception occurred, whereas in case J,  $w$  refers to the subtarget  $s$  (the call point).

The general definition of the objective function  $\omega_t$  involves a number of constants which represent either penalty values or weights. Even though constant  $\lambda$  is problem-specific (it must ensure that the method call distance weighs more than control node distance

and branch distance together; the maximum value of these two depends on the number of control nodes of the method with the largest control flow graph), all constants can be globally adjusted without the need to incorporate static or dynamic knowledge of the software under test. However, the following relations must be satisfied in order for the objective function not to be misleading:

$$\lambda_C > 1 \quad (3.15)$$

$$\lambda > \max(w) \quad (3.16)$$

$$p_{dyn} > \max(\lambda d_M + w + p_s) \quad (3.17)$$

$$p_{loop} > \max(\lambda d_M + w + p_s) \quad (3.18)$$

$$p_s > \max(\lambda d_M + w) \quad (3.19)$$

where  $\max(x)$  is the maximum possible value occurring for  $x$ .

### 3.6 Test Cluster Definition

In most cases, an object-oriented class is not *standalone*, rather, it is associated to other classes whose services are drawn upon. Hence, generating test sequences for a given class under test dynamically involves the creation and manipulation of the associated classes – or of surrogates of them. The set of all classes relevant for testing a particular class  $c$  is referred to as *test cluster*  $C_c$  of  $c$ , as introduced in Section 2.1.3 on page 10. The test cluster for class  $c$  can be obtained by performing a transitive static analysis of the signatures of the public methods of this class. Each type (class or interface) encountered during this analysis is added to the test cluster. After all methods of the class under test have been considered, the analysis continues by analyzing all public methods of the test cluster types which have not yet been considered. Once all method signatures have been analyzed in this manner, the test cluster contains all relevant types.

However, the test cluster created via static analysis is not necessarily accurate for test sequence generation due to the following reasons:

- The test cluster may contain abstract classes and interfaces, respectively. However, neither abstract classes nor interfaces can be instantiated. Therefore, concrete classes are required that replace these types within the test cluster.
- For the purpose of isolation, it might be desired to replace some of the test cluster classes by *dummy classes* (or *mock classes*) which possess a much simpler implementation and are known to contain no errors.
- The test cluster does not account for arrays since no explicit array classes exist that can be included.

Consequently, the definition of the test cluster requires further consideration. The following sections will address these issues. At first, Section 3.6.1 on the following page introduces mock classes which are suited to serve as a surrogate to complex application

classes. At the same time, they can be used to replace interfaces and abstract classes within the test cluster. This will be dealt with in Section 3.6.2 on the next page. Finally, Section 3.6.3 on page 88 elaborates on arrays.

### 3.6.1 Mock Classes

For the purpose of testing, it is sometimes advisable to replace particular test cluster classes by artificial, *mock classes* (Beck, 2003). Then, during test execution instances of the mock classes are used instead of instances of the genuine classes. A mock class exhibits the same interface as the class that it replaces. However, the implementation of the method and thus the behavior of a mock object might be completely different from that of the replaced instance. Mock objects are used when the genuine class possesses behavior which is hard to reproduce (for example provoking a network error), when it is very complex and slow (for instance waiting for certain timers), when it requires a particular environment to work properly (for instance depending on a database that must contain data sets of a particular scheme).

Different approaches exist to automatically create a mock class from a given genuine class, for example Mock Objects (2006). Typically, the generated mock class possesses additional methods that allow configuring the mock object appropriately with respect to the intended test scenario. This configuration comprises for instance the definition of fixed return values of the public methods. Listing 3.6 shows part of class **DatabaseAdapter** which allows establishing a connection to a database and querying data items.

Listing 3.6: DatabaseAdapter class to be replaced

---

```

1 class DatabaseAdapter
2 {
3     public DatabaseAdapter() {...}
4     public boolean connect(String address,
5                             String user,
6                             String pwd)
7         {...}
8     public Object query(String query) {...}
9 }
```

---

If the class under test depends on class **DatabaseAdapter**, a database must be present and set up appropriately in order to realize a particular scenario during testing. However, testing will be much simpler if class **DatabaseAdapter** is replaced by a mock class which does not require a concrete database to be present but exhibits the behavior that class **DatabaseAdapter** would. Listing 3.7 shows the mock version of class **DatabaseAdapter**.

Listing 3.7: DatabaseAdapter mock class

---

```

1 class DatabaseAdapter
2 {
3     private Vector retVals__connect = new Vector();
4     private Vector retVals__query   = new Vector();
5     private int  retVal__connect = 0;
```

---



```

6   private int retVal__query    = 0;
7
8   public DatabaseAdapter() { }
9
10  public boolean connect(String address ,
11                          String user ,
12                          String pwd) {
13      return ((boolean)returnValues__connect.get(
14          retVal__connect++ \%
15          retVal__connect.size ())).boolValue ());
16  }
17
18  public void addRetVal__connect(boolean bool) {
19      retVal__connect.add( new Boolean(bool) );
20  }
21
22  public Object query(String query) {
23      return retVal__query.get(
24          retVal__query++ \% retVal__query.size ( ) );
25  }
26
27  public void addRetVal__query(Object o) {
28      retVal__query.add( o );
29  }
30 }

```

Note that the mocking concept behind the shown mock is very simple and naive. The mock class possesses very simple method implementations which rely on data structures that contain the return values. For instance, method `connect()` returns a Boolean value that indicates whether or not the connection has been successfully established. The value to be returned can be configured using method `addRetVal__connect(boolean)`. The mock supports the definition of multiple return values for a method. On each call, the index for the return value vector is incremented. When reaching the end of the vector of predefined return values, it is again started with the first element. The mock class does not demand an underlying database. However, it requires a suitable configuration of the return values of its methods. Note that the mocking concept discussed is only one approach to replacing a complex or non-instantiable class.

In terms of the evolutionary search, the configuration of a mock object is part of the search process since the mock configuration methods can appear in a test sequence. Suitable configurations of the participating mock objects are searched which enable the attainment of the test goal at hand.

### 3.6.2 Interface Implementers and Abstract Class Implementers

Mock classes, as previously described, can be used to replace interfaces and abstract classes within the test cluster. From both interfaces and abstract classes no instances can be created. However, when using a mock class instead of an interface or an abstract

class, instances complying with a particular interface or abstract class can be obtained. Therefore, all interfaces of the test cluster are replaced by appropriate mock classes. All abstract classes of the test cluster are replaced by mock classes which implement either all methods – thereby replacing already existing ones – or only the abstract methods, thus preserving the original implementations of the non-abstract methods.

### 3.6.3 Array Generators

Arrays of arbitrary dimensions can also be part of the test cluster, since they can appear in the signature of a public method. It must be enabled that arrays can be constructed from any subset of objects and values available at the moment of calling a test sequence method which expects an array-type argument. In order to allow for a high flexibility of array constructions, for each array type of the test cluster, an *array generator* is added to the test cluster which enables the creation of arbitrary arrays. For instance, in the case of a one-dimensional array of the `Integer` type, the array generator shown in Listing 3.8 is used.

Listing 3.8: Array generator for class `Integer`

---

```

1 class ArrayGenerator_Integer
2 {
3     private Vector elements = new Vector();
4
5     public void addElement(Integer i) {
6         elements.add( i );
7     }
8
9     public Integer[] getArray() {
10         return (Integer[]) elements.toArray(
11             new Integer[elements.size()]);
12     }
13 }

```

---

This array generator maintains an internal vector of `Integer` instances. Elements can be added to this vector using the `addElement` method. An array is retrieved by calling method `getArray()` which constructs an array from the elements currently present in the internal vector.

In analogy to the configuration of the mock objects, the setup of the array generators as well as the retrieval of array instances are part of the test sequence and will be evolved during the evolutionary search.

## 3.7 Function-Assigned Flags

The objective function for a given test goal is essentially composed of the three metrics *method call distance*, *control dependence distance*, and *branch distance*, as described in Section 3.5 on page 76 (besides the various constants). In many cases, the combination

of these metrics leads to a smooth *objective function landscape*. This landscape is a multi-dimensional visual representation of the objective function: each point in the phenotype search space is assigned its objective value in an additional dimension. The entirety of the objective values forms a landscape possibly containing ascents, descents, peaks, ridges, and plateaus. Note that the neighborhood relation of two points in the phenotype search space is indirectly defined by the variation operators of the evolutionary search in the genotype search space. An objective function is particularly suited to effectively guide the evolutionary search if the corresponding landscape is smooth, meaning that from each point it can be obtained whether or not a neighboring point improves the objective value. If plateaus are present in the landscape, this information cannot be obtained easily by exploring the close neighborhood of a point. As a result, plateaus hinder the evolutionary search, in grave cases even causing it to fail.

In the field of evolutionary testing, sometimes suboptimal objective functions implying landscapes with plateaus occur: in the case a Boolean predicate must be satisfied for a particular path to be traversed during test sequence execution, the metric branch distance is a binary function, returning either 0 (if the condition is satisfied as desired) or 1 (otherwise). This binary characteristic of the metric results in plateaus in the objective function landscape.

In the literature, this issue is known as the *flag problem*. Several approaches to the flag problem have been suggested, all of which address procedural programming languages only. However, in the area of object-oriented programming, a particular type of flag occurs relatively frequently which is not dealt with by the suggested approaches: *function-assigned flags*. This section deals with function-assigned flags in particular in order to enable more efficient evolutionary searches in their presence.

A function-assigned flag is a Boolean variable which is assigned the return value of a Boolean-returning method; in addition, the flag value is used by a predicate.

Listing 3.9: Example of function-assigned flag

---

```

1 // fragment A
2 if ( stack.isFull() )
3     // target

```

---

Listing 3.9 shows a code fragment with a function-assigned flag. Variable `stack`, supposed to be of type `IStack` (shown in Listing 3.11), is referenced in the condition to call the `isFull` method, whose return value controls the outcome of the condition. Hence, the predicate represents a function-assigned flag, leading to a plateau in the objective function landscape: for all cases where the stack is not full, the same objective value occurs.

The following section discusses how the existing approaches to the flag problem (neither one addresses function-assigned flag explicitly) deal with function-assigned flags. Afterwards, two new approaches which particularly address function-assigned flags, but are applicable to other flag problem as well, will be presented: the first, referred to as *method substitution*, relies on the substitution of particular method calls, while the

second, referred to as *Boolean variable substitution*, relies on a more complex substitution of Boolean variables. Both approaches apply code transformations which create modified versions of the original software under test. These modified versions are used during the process of test sequence generation only; afterwards they become useless and will be discarded.

### 3.7.1 Existing Approaches to Flag Removal

The flag removal algorithm of Harman, Hu, Hierons, Baresel and Sthamer (2002b) relies on a code transformation that substitutes the flag within a condition by the flag-defining expression. This code transformation is, however, not beneficial in the case of function-assigned flags: the flag-defining expression is a function call itself; its usage instead of the flag variable does not change the objective function landscape. Moreover, the transformation might even change the application logic of the implementation which could lead to inconsistent states of the system under test.

Listing 3.10: Problematic flag transformation

---

```

/* original version */
boolean connected = database.connect();
if( connected ) ...
else ...

/* transformed version */
boolean connected = database.connect();
if( /* replaced flag:*/ database.connect() ) ...
else ...

```

---

Listing 3.10 shows the original implementation with a flag at the top and the modified version, transformed according to the approach of Harman et al. at the bottom. In the transformed version, the `connect()` method is called twice which might be a problem, depending on the implementation of the database logic.

Bottaci (2002) describes the general idea of using the distance of the conditions that control a flag assignment as a replacement distance for the distance of the condition that involves the flag. He introduces additional variables storing these intermediate distance values. However, he does not discuss how the idea would extend to function-assigned flags.

Baresel and Sthamer (2003) perform a static analysis in order to classify all occurring flag assignments into the categories *desired* and *undesired*. They include the conditions controlling the assignments into the fitness calculation: the negated conditions controlling the undesired assignments are added to the flag condition which is used to calculate the branch distance. However, in the presence of function-assigned flags the static analysis is limited in that it cannot decide whether a flag definition is desired or undesired – this depends on the actual return value of the flag-assigning function at runtime.

Baresel, Binkley, Harman and Korel (2004) deal with the issue of loop-assigned flags. They also classify the flag assignments according to desired and undesired. Additionally,

they introduce local fitness functions and additional local variables. They reformulate the flag condition using the additional variables. The approach is equally limited in the presence of function-assigned flags as the approach of Baresel and Sthamer (2003).

Liu, Liu, Wang, Chen and Cai (2005a) introduce a fitness variable for each flag variable, the value of which is calculated with the help of aggregation rules. They also classify the flag assignments according to desired and undesired. Whenever a flag condition is to be satisfied, the fitness value of the additionally introduced fitness variables is used. Again, due to the classification into the categories desired and undesired, the approach is limited in the presence of function-assigned flags.

McMinn and Holcombe (2006) combine the idea of *chaining* (Ferguson and Korel, 1996) and evolutionary testing to cope with the problem of internal variables and internal states, including the flag problem. A transitive static analysis identifies all statements that manipulate the variables controlling a flag assignment. Chains of such definitions are considered systematically to define the objective function that then rewards the execution of definition statements so as to eventually enable the desired flag assignment. However, the approach of McMinn and Holcombe (2006) is not sufficient in the presence of dynamic method binding. Reconsider the flag example in Listing 3.9 and also consider the different stack types in Listing 3.11.

Listing 3.11: Polymorphic stack types

---

```

1 interface IStack {
2     // standard stack operations
3 }
4
5 class SimpleStack implements IStack {
6     ...
7     boolean isFull() {
8         if( elements.size() == MAX_SIZE )
9             return true;
10        else return false;
11    }
12 }
13
14 class ExtendedStack implements IStack {
15     ...
16     boolean isFull() {
17         if( keepInLimit() )
18             if( elements.size() == MAX_SIZE )
19                 return true;
20             else return false;
21         else return false;
22    }
23 }
```

---

The static analysis performed by McMinn and Holcombe must statically decide what the definition of the return value of method `isFull` is. However, multiple candidates are available to be included in the calculation of the objective value. The first class,

`SimpleStack`, compares the number of currently stored elements to a maximum size bound in order to calculate the return value of method `isFull`. The second class, `ExtendedStack`, allows exceeding the maximum size unless it is configured to keep the size within the limit. With respect to code fragment A, it depends on the runtime type of variable `stack` – which is supposed to be either `SimpleStack` or `ExtendedStack` – which predicates must be incorporated into the flag fitness calculation: in the first case, the predicate `(elements.size()==MAX_SIZE)` is relevant, whereas in the latter case, both predicates `(keepInLimit())` and `(elements.size()==MAX_SIZE)` are relevant. The right choice concerning the class to be considered can be made at runtime only when the effective method binding will be detected, meaning when it is clear of which type variable `stack` actually is. Additionally, the chaining approach does not know about various instances of the same type. Executing a definition statement for an arbitrary instance might not be helpful; rather, the elements of a chain must include the information as to which instance a definition must occur for.

In general, most of the previous approaches incorporate the knowledge of what a desired and what an undesired flag assignment is into a code transformation or calculation rule. They also require the conditions controlling the calculation of a flag value to be present in the context of the considered function meaning that the information as to which condition controls a flag assignment is statically accessible. As shown by an example, this requirement cannot be satisfied in the presence of dynamic method binding and instantiation, which are fundamental features of object-oriented software.

### 3.7.2 Method Substitution

*Method substitution* is a code transformation which applies to “well-known” methods returning Boolean values or having very small return value ranges. `Object.equals(-Object)` and `Comparable.compareTo(Comparable)` are examples of such well-known methods. A call to such a method in the source code of a test cluster class will be replaced by a call to an alternative implementation provided by the testing framework. In contrast to the method it replaces, the framework method does not have a Boolean return types, but a numeric one. Consider the code fragment in Listing 3.12.

Listing 3.12: Original predicate

---

```

1 public void m(Integer i1, Integer i2) {
2     if ( i1.equals(i2) )
3         // do something
4     ...
5 }
```

---

It shows method `m` which takes two `Integer` arguments. The predicate `i1.equals(i2)` is a Boolean predicate (a function-assigned flag) since method `equals` returns a Boolean. In this case, the branch distance would be 0 if `i1` equals `i2`, and 1 otherwise. However, this branch distance would not give any hint as to how close the condition is to be satisfied (or not).

To cope with this kind of predicates, the well-known `equals` method of type `Integer` is replaced by a method which returns a more meaningful value: the absolute difference of the two integers. Additionally, the condition must be adapted to ensure that the expression evaluates to a Boolean. The modified version of the method is shown in Listing 3.13.

Listing 3.13: Modified predicate

---

```

1 public void m(Integer i1 , Integer i2) {
2     if( INTEGER.abs( i1 , i2 ) == 0 )
3         // do something
4     ...
5 }
```

---

The framework method `INTEGER.abs` is used instead of the original `equals` method. Its return value is compared to 0. This reformulation is semantically equivalent to the original equality-checking condition; however, it implies a smoother objective function landscape due to the non-Boolean distance function that is applied now (the distance function of the equality operator for integers).

The strategy of method replacement can be applied to various well-known types and methods. In the case of class `String`, the framework method for `equals` implements a similarity metric for strings which indicates how equal two strings are. For instance, the metrics discussed by Alshraideh and Bottaci (2005) can be employed.

### 3.7.3 Boolean Variable Substitution

Another approach to improving the objective function landscape in the presence of function-assigned flags is the substitution of Boolean variables. The general idea is to replace the `Boolean` type of the Boolean variables in the code by a more meaningful type, such as `double`. Then, presuming a suitable convention, the value of such a “Boolean” variable indicates whether it is actually `true` or `false`, and also how far it is away from being the opposite value (`false` or `true`, respectively).

In the following, the Boolean variable code transformation will be detailed. It consists of three steps, referred to as *tactics*. These tactics will be described using a running example. This example refers to the methods `func1` and `func2` of a class, whose complete definition is not relevant here.

#### Tactic 1: Branch Completion

This tactic completes the “invisible” `else` branches of all conditions and adds tautological flag assignments. Table 3.4 on the following page shows a sample program on the left. On the right, it shows the same program after tactic 1 has been applied. All branches have been completed and tautological assignments of the flag in question have been inserted (lines 6 and 7). The intention of this tactic is to make a flag assignment occur irrespective of the path taken during execution of the function. This tactic along with tactic 3 ensures that a guiding distance value can always be calculated. The scope of this

original program:	tactic 1 applied:
<pre> 1 void func1(int a, int b) 2 { 3     boolean flag = false; 4     if( a == 0 ) 5         flag = func2(b); 6 7 8 9     if( flag ) 10        // target 11 } 12 13 int func2(int b) 14 { 15     if( b==0 ) 16         return true; 17     else 18         return false; 19 }</pre>	<pre> void func1(int a, int b) {     int flag = 0;     if( a == 0 )         flag = func2(b);     else         flag = flag;      if( flag )         // target }  int func2(int b) {     if( b==0 )         return true;     else         return false; }</pre>

Table 3.4: Tactic 1

tactic are all conditions that control a flag assignment. If there are nested conditions, all parent conditions are also considered and their respective alternative branches are also expanded.

### Tactic 2: Data Type Substitution

This tactic substitutes the data type `Boolean` with the data type `double` in all flag declarations. This comprises not only local variable declarations but also the return type of functions that are supposed to return a Boolean value.

The data type substitution is the most essential tactic of the overall code transformation. Positive values indicate that the flag is `true` whereas negative values indicate that it is `false`. It assumes the value 1.0 to be “maximum true” and the value  $-1.0$  to be “maximum false”. By convention, 0 is a (unused) neutral value. The flag value’s amount expresses how far it is away from being the opposite value. For example, a flag value of  $-0.5$  indicates that the flag is `false` and is 0.5 “away” from being true. The distance that the flag value represents is later used during the evolutionary search to calculate the objective value. The gradual values (that substitute `true` and `false`, respectively) produce a smooth objective function landscape which provides better guidance to the evolutionary search than the objective landscape for the untransformed source code.

The data type substitution also requires that all conditions involving the flag are modified. When the flag value is compared to `true`, e.g. `(flag == true)`, the predicate will be changed to `(flag > 0)`. The modified predicate makes use of a relational operator defined



for real values instead of the equality operator defined for Boolean values. Analogously, a comparison to false would be changed to  $(\text{flag} < 0)$ . Short-hand predicates, such as  $(\text{flag})$  would be completed to  $(\text{flag} == \text{true})$  first before applying the transformation.

The tactic also demands a special treatment of the negation operator. All occurrences of it are replaced by the  $-$  operator (the negation operator for real values). For instance, a condition  $\text{if}(\text{!flag})$  would then be modified to  $\text{if}(-\text{flag} > 0)$ .

tactic 1 applied:	+ tactic 2 applied:
<pre> 1 void func1(int a, int b) 2 { 3     int flag = false; 4     if( a == 0 ) 5         flag = func2(b); 6     else 7         flag = flag; 8 9     if( flag ) 10        // target 11 } 12 13 int func2(int b) 14 { 15     if( b==0 ) 16         return true; 17     else 18         return false; 19 }</pre>	<pre> void func1(int a, int b) {     double flag = -1;     if( a == 0 )         flag = func2(b);     else         flag = flag;      if( flag &gt; 0 )         // target }  double func2(int b) {     if( b==0 )         return 1;     else         return -1; }</pre>

Table 3.5: Tactic 2

### Tactic 3: Local Instrumentation

The local instrumentation assigns gradual distance values to the flag variables. Therefore, each right-hand operator of a flag assignment will be instrumented meaning that the actual right-hand expression is replaced by a call to a distance function. In Table 3.6 on the next page, the constants have been replaced by calls to function `dist` which returns the distance for the expression passed to it. The angle brackets used for the second argument passed to the `dist` function shall mean that the formal expression as well as the actual values of the concerned variables are passed. The call to `func2` has been replaced by a call to function `map`. The following listing shows the pseudo-code of the `dist` function.

---

```

1 double dist(double assignedValue, expression exp,
2             int nestingLevel) {
3     double distance = branch_dist(exp);
```

tactic 2 applied:	+ tactic 3 applied:
<pre> 1 void func1(int a, int b) 2 { 3     double flag = -1; 4     if( a == 0 ) 5         flag = func2(b); 6 7 8     else 9         flag = flag; 10 11 12     if( flag &gt; 0 ) 13         // target 14 } 15 16 double func2(int b) 17 { 18     if( b==0 ) 19         return 1; 20 21     else 22         return -1; 23 24 }</pre>	<pre> void func1(int a, int b) {     double flag = -1;     if( a == 0 )         flag = map(             func2(b),             2);     else         flag = dist(flag,             &lt;a==0&gt;, 1);      if( flag &gt; 0 )         // target }  double func2(int b) {     if( b==0 )         return dist(1,             &lt;b==0&gt;,1);     else         return dist(-1,             &lt;b==0&gt;,1); }</pre>

Table 3.6: Tactic 3

```

4   distance = map(distance , nestingLevel);
5
6   if( assignedValue < 0 )
7       distance = -distance;
8   return distance;
9 }

```

Initially, the conventional branch distance will be calculated based on the passed expression. This calculation depends on the applied relational operator; for each operator, a particular distance function is defined, as described in Section 2.2.2 on page 20. Then, the distance is mapped to a particular range using the `map` function. This function realizes the idea of interval bisection which can be regarded as the inversion of the control dependence distance approach. Interval bisection allows the integration of several pieces of information into one real value. Here, this information consists of the actual branch distance of the condition that controls the flag assignment and the *nesting level*. The nesting level of a statement corresponds to the number of conditions that control this statement; this number is used instead of the control dependence distance since the latter cannot be calculated by the local objective function unambiguously using static analysis. This is due to the dynamic function binding as mentioned above. The number of nesting levels may differ from function call to function call depending on how many levels the called function possesses.

Equation 3.20 shows the relationship between the original distance ( $d_{orig}$ ) and the resulting mapped distance ( $d_{mapped}$ ) where  $l$  is the nesting level. The `map` function implements this formula.

$$d_{mapped} = \text{sign}(d_{orig}) \cdot \frac{1 + |d_{orig}|}{2^l} \quad (3.20)$$

An example shall demonstrate the idea behind the `map` function.

The argument for calling `func1` is a pair  $(a, b)$  of integers. Table 3.7 shows the nesting level (column  $n/l$ ), the branch distance (column  $b/d$ ), the flag value, and a graphical representation of the absolute amount of the flag value (called *interval*) that the arguments  $(1, 1)$ ,  $(1, 0)$ ,  $(0, 1)$ , and  $(0, 0)$  would achieve when being passed as arguments to `func1`. Pair  $(1, 1)$  and  $(1, 0)$  do not satisfy the first condition of `func1`,

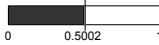

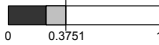
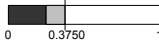
(a,b)	n/l	b/d	flag value	interval
(1,1)	1	0.0005	-0.5002	
(1,0)	1	0.0005	-0.5002	
(0,1)	2	0.0005	-0.3751	
(0,0)	2	0.0005	+0.3750	

Table 3.7: Arguments for `func1` and the resulting flag values

hence leading to the traversal of the alternative branch and achieving nesting level

1. The branch distance 0.0005 indicates how close execution was to evaluating the first condition to *true*. This value originates from the corresponding distance function ( $d_{==}$ ). The flag value is negative in these cases, indicating a **false** flag outcome. The corresponding intervals in Table 3.7 on the preceding page show a solid lower half which can be regarded as a reserved area for higher nesting levels. The branch distance of 0.0005 has been mapped to the upper half, resulting in the absolute flag value 0.5002. Pair (0,1) satisfies the first condition of function **func1** and leads to the traversal of the alternative branch of the decision in function **func2**, hence achieving a nesting level of 2.

2. The miss of the **true** branch of this condition has been taken into account by the branch distance of 0.0005. As the interval shows in this case, the lower part is bisected as compared to the first two intervals, and the branch distance is mapped into the upper one of these new halves. Finally, pair (0,0) satisfies both conditions and leads to an assignment of **true** to the flag variable. Therefore, the sign of the flag value is positive, indicating the **true** value. The absolute value of the flag indicates how close execution was to avoiding the **true** outcome.

### 3.8 Summary

The chapter presented the evolutionary class testing approach based on genetic programming. At first, formal definitions relating to test sequences and their feasibility were given. These definitions built the basis for the subsequent considerations on two representations of test sequences. The aim when designing the representations was to restrict the space of test sequences searched by an evolutionary algorithm to executable ones. The first representation relies on a tree representation of test sequences, where the trees encode the method calls only and the parameters for the single calls are not encoded. Such tree is said to represent a *test sequence fragment*. Test sequence fragments are evolved via genetic programming which applies selection operators and variation operators on trees. The required parameters are represented using vectors of primitive values, such as integers and doubles. For each test sequence fragment, a second evolutionary search via a genetic algorithm working on vectors of primitives is carried out. The definition of the vector (its size and its elements' data types) depends on the test sequence fragment for which the parameters are sought. Put together, a test sequence fragment along with a corresponding concrete parameter vector build an executable test sequence. In order to simplify this expensive two-level approach – which is due to the search space separation into the subspaces *sequence space* and *parameter space* – the second representation relies exclusively on trees. Both method call information and parameter information is encoded in one tree. The genetic operators of the genetic programming algorithm that evolves the trees are applicable to both the method information and the parameter information. The two representations do not require the definition of new genetic operators, neither for crossover, nor for mutation. Well-researched and well-established operators can be applied for test sequence generation. The representations focus the search on exploring executable test sequences only, formally infeasible sequences are not part of the search space. However, it is not possible to prevent the search from encountering dynamically

infeasible test sequences.

A strategy for objective function construction was discussed. It integrates the two conventional distance metrics branch distance and control dependence distance into a comprehensive objective function framework that can handle all possible situations that may arise during the execution of a test sequence. Runtime exceptions are coped with by introducing the additional distance metric *method call distance*. The concept of *call points* was introduced in order to cope with non-public methods. A call point is a statement which involves a call to the targeted non-public method. When a test goal belonging to a non-public method is addressed, the evolutionary search consists of two phases: in the first phase, the call point is tackled, meaning the search aims at finding a test sequence which indirectly invokes the method in question. Then, in the second phase, test sequences involving a call to the non-public method are further evolved as to finally find a test sequence which covers the targeted test goal within the non-public method. The objective functions defined by the approach effectively handle runtime exceptions and non-public methods, as will be demonstrated in Section 4.2 on page 106.

For the discussion of both the representations and the objective function it was assumed that the underlying test cluster consists of classes which can be instantiated. However, since this is not generally the case, approaches to replacing non-instantiable types by instantiable types were suggested. Mock classes were introduced to replace interfaces, abstract classes, and user-selected classes, respectively. In addition, array generators were introduced in order to allow the evolution of array-type method arguments.

The last part of this chapter revisited objective function design. Boolean predicates, often relying on the Boolean return value of a method, imply plateaus in the objective function landscape, hence hampering the evolutionary search. Two strategies to cope with *function-assigned flags*, a particular type of Boolean predicates, were suggested: *method substitution* and *Boolean variable substitution*. The former relies on the replacement of well-known methods returning Boolean values by framework methods returning more meaningful values, such as integers. The latter relies on the replacement of the data type `Boolean` when used as return type of a method. This replacement requires further modifications of the source code, for instance the insertion of local objective functions. Both strategies replace Boolean values by more gradual values with a larger value range that allow for the application of better distance functions and consequently lead to better objective functions. The improvement will be empirically demonstrated in the following chapter.

With regards to the limitations of automatic test generation approaches identified in Section 2.2.4 on page 31, the evolutionary class testing approach has the properties shown in Table 3.8 on the next page.

Evolutionary class testing is a dynamic, search-based approach to automating the generation of test sequences. It does not apply symbolic execution and constraint solving. It transforms the task of test generation to a set of search problems. For each search problem, both the search space and the respective objective function are defined automatically. Points in such a search space are test sequences that include the creation and configuration of parameter objects. Therefore, the approach is applicable to testing

Limitation	Evolutionary class testing
Symbolic execution and constraint solving	Search-based approach not involving symbolic execution and constraint solving
Support of Class-Type Parameters	Generation of test sequences that include the creation and setup of parameter objects
Maintainability and Usage	Generation of test sequences that call only public methods
Inexecutable test sequences	Usage of a tree-based representation which significantly reduces the probability of occurrence of inexecutable test sequences during the search
Complex predicates	Application of distance functions for the various relational and logical operators that a predicate can make use of
Runtime exceptions	Incorporation of an additional distance metric that accounts for runtime exceptions during the search
Non-public methods	Incorporation of an additional penalty mechanism that accounts for non-public methods
Additional user input	No need; yet, the specification of irrelevant classes and methods is beneficial

Table 3.8: Properties of evolutionary class testing with respect to the limitations

classes with methods having class-type arguments. In addition, the encapsulation of the objects is not broken since a test sequence includes calls to public methods only.

A novelty of evolutionary class testing is that it relies on a tree-based representation of test sequences. This representation accounts for the call dependences that exist among the methods of the test cluster. As a result, the probability of occurrence of inexecutable test sequences, from which other search-based approaches suffer, can be reduced significantly.

The approach to constructing the objective function builds on the distance-based approach to objective function design in the area of procedural evolutionary structural testing. Distance functions tailored to the relational and logical operators occurring in a predicate are used, which enables effective evolutionary searches for relatively complex predicates also. The idea for handling runtime exceptions is that the objective functions include a special metric that refers to the number of unexecuted method calls of the candidate test sequence. This metric ensures that sufficient guidance is provided to the evolutionary search so as to overcome the exception and finally reach the test goal.

Test goals belonging to non-public methods are dealt with by incorporating a particular penalty mechanism into the objective functions. A penalty applies if the non-public method has not indirectly been called. In this case, the evolutionary search is guided so that test sequences are encountered which call the non-public method and finally reach the test goal.

Apart from the source code of the class under test and optionally the source codes of the other involved classes, the evolutionary class testing approach does not require additional user input. The configuration of the genetic programming algorithm can be globally defined once and needs not be changed for every new process. Yet, an additional specification of which classes are not relevant and which methods might be omitted during the search can be beneficial with respect to search space reduction and hence the performance of the approach (cf. Section 5.2 on page 134).





## 4 Experiments

This chapter reports on the experiments performed with test sequence generating algorithm *TCGen2*, described in Section 3.4.2 on page 74. This algorithm is implemented by the test sequence generator *EvoUnit*. EvoUnit is presented in short in Section 4.1. The general intention of the experiments was to empirically investigate the effectiveness of the evolutionary class testing approach with respect to the elaborated limitations. Therefore, a case study consisting of 34 real-world Java classes, taken from 5 different open-source development projects, was carried out. The results were contrasted with the results from a random testing approach as well as with the results from the two commercial test sequence generators CodePro (described in Section 2.2.3 on page 30) and Jtest (described in Section 2.2.3 on page 31). These experiments, including the setup of the test sequence generators and the achieved results, are described in Section 4.2 on page 106. The effectiveness of the approach to testing non-public methods was studied in particular. The results of experiments demonstrating the effectiveness of the evolutionary class testing approach with respect to generating tests for non-public methods are summarized in Section 4.3 on page 125. Section 4.4 on page 127 reports on the results of the case study that analyzes the effectiveness of the strategy *Boolean variable substitution*, described in Section 3.7.3 on page 93. Finally, a summary of the experiments concludes this chapter in Section 4.5 on page 130.

### 4.1 Implementation of EvoUnit

The architecture of EvoUnit is depicted in Figure 4.1 on the following page. The input to the system is a text file that specifies the properties of the test sequence generation process. These properties for instance include the name of the class under test, the settings for the evolutionary search, and the classes to be replaced by mock classes. It can also be specified which classes are to be used as a replacement for interfaces and abstract classes. Additionally, reduction patterns which filter out some of the methods when defining the function set can be specified for each class. This helps reducing the search space size. The output of EvoUnit is a JUnit class that implements a test case method for each generated test sequence.

Three major components can be identified in the figure: Runner, Optimizer, and Class Loader. Runner is the component that controls the overall process. It uses Class Loader to acquire all relevant classes in Java byte format. Optimizer is employed to carry out the evolutionary searches for the individual test goals. These components and the main flow will be explained in more detail below.

Runner initializes the system by reading in the user-specified property file. Depending

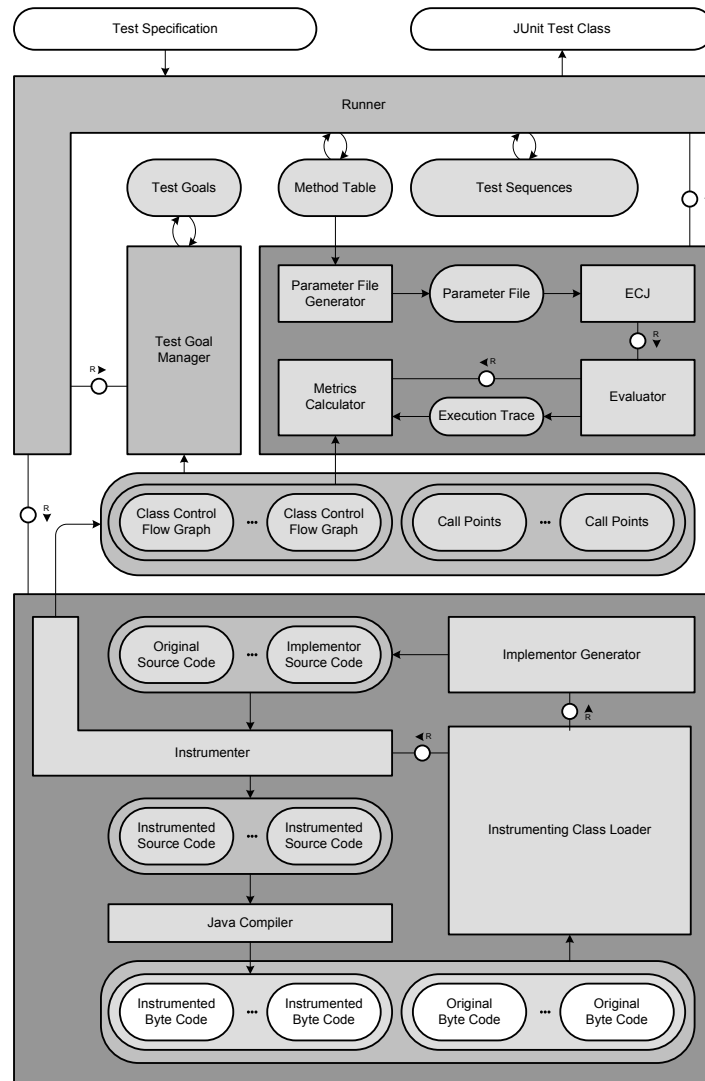


Figure 4.1: EvoUnit System Architecture

on the specified class under test and the specification of other classes, it creates a method table that contains all classes and their methods that are considered relevant for testing the class under test. Creating the method table requires the classes in question to be loaded. This happens via a particular class loader. This class loader behaves like the standard system class loader except that it instruments the classes that it should load on-the-fly. This necessitates that the source code of the class to be loaded is available. If it is, the instrumentation process starts. Otherwise, the standard class loader is used, loading the uninstrumented byte code.

The instrumentator parses the source code using the OpenJava parser (Tatsubori, Chiba, Killijian and Itano, 2000). Trace statements are inserted into the methods that will create an execution trace when the methods are executed later. Simultaneously, Instrumentor also constructs the control flow graphs of the methods of the class being loaded. The control flow graphs of all classes are stored in a global repository. Once the source code of a class is instrumented, the Java compiler translates it to byte code. The resulting byte code is then loaded by the instrumenting class loader. Both the instrumented class and the control flow graphs are stored persistently, allowing skipping instrumentation and construction of the control flow graphs if the system can find them in the repository. It may happen that Runner instructs to load an abstract class or an interface when creating the method table. In this case, Implementor Generator automatically creates an implementing mock class and returns the corresponding byte code instead of the requested interface or abstract class, respectively. This ensures that the method table contains methods of concrete types only.

Once the method table is created, it is the task of Test Goal Manager to create and maintain the list of test goals (that is, all decisions/branches of the control flow graphs of the methods of the class under test). Afterwards, Runner iterates over the list of test goals via Test Goal Manager. For each test goal to do, the following procedure is performed:

First, the method table is modified according to the current test goal. This involves the specification of the method under test, that is, the method which contains the current test goal. Parameter File Generator compiles a parameter file to be used by ECJ (Wilson et al., 2004). This parameter file contains the specification of both the type set and the function set. Elements of the type set are extracted from the signatures of the methods of the method table, whereas the methods of the method table become elements of the function set. Then, the optimization process is started by invoking ECJ. It carries out the evolutionary search by using Evaluator as a callback for acquiring the objective value of a given individual. Evaluator executes the test sequence represented by the given individual; by doing so, an execution trace is created. Metrics calculator, which implements the algorithms for computing the distance metrics mentioned in Section 3.5 on page 76, calculates the objective value based on the execution trace and the control flow graph information. Once a covering test sequence is found, Runner inserts this sequence into the global list of relevant test sequences. Finally, if all test goals have been done, the JUnit class is created based on the set of stored test sequences.

EvoUnit does not only return the JUnit class file. Information about the test genera-

tion process including the development of the objective values and the corresponding individuals are collected during the run and returned afterwards as well. This allows comprehending the test generation process even after its termination, which is particularly useful when there are unachieved test goals.

## 4.2 General Effectiveness Case Study

In order to empirically examine the effectiveness of the evolutionary class testing approach, a case study with 34 classes was conducted. These classes were taken from five different Java development projects to consider a broad spectrum of types of test objects. Section 4.2.1 describes the class selection criteria and the resulting assortment of classes.

The degree of method/decision coverage (cf. Section 2.1.4 on page 11) achieved by the set of test sequences for each class was taken as the main criterion for effectiveness assessment. Since the degree of coverage achieved is not suggestive of effectiveness when considered without any reference, the coverages were contrasted to the coverages achieved by a random approach. In addition, the coverage results were also contrasted to the coverage results achieved by the two commercial test sequence generators CodePro and Jtest. Achieving higher coverages than the random approach would suggest the value of the evolutionary class testing approach in general, and achieving higher coverages than the commercial generators would suggest even greater value.

The setup of the experimental framework, including the setup of the generators, is described in Section 4.2.2 on page 111. The results obtained and their analysis is given in Section 4.2.3 on page 116.

### 4.2.1 Test Objects

#### Selection Criteria

The following criteria were applied for the selection of the projects from which to choose test objects as well as for the selection of the individual classes to be tested:

- *Availability.* The source code of the classes must be freely available in order to ensure reproducibility of the experiments. Additionally, the source code must be available and accessible easily.
- *Relevance and Popularity.* The projects must also possess a particular relevance and popularity, respectively. Relevance is defined by the importance of the originating party while popularity is defined in terms of references.
- *Complexity.* The projects and classes must not be too simple, meaning that covering the code of a particular class must not be trivial but rather require an effort. (The results of the random approach suggest the triviality of a class.)
- *Dependencies.* The candidate project must not depend on too many other projects that require an extensive configuration of the system on which the test sequence

generator runs. Note that performing dynamic test generation includes that the class under test will be instantiated, requiring all code dependencies on other classes to be resolved.

- *Type.* Projects with a primary focus on graphical components (such as GUI libraries) are considered to be not relevant. This is due to the fact that GUI classes require special treatment when they are tested.
- *Usage of string literals.* Projects and classes that make strong use of strings are excluded. For instance, all XML-related projects are not considered since the test sequence generation would need to create string inputs that correspond to the XML tag names of a particular XML schema.
- *Usage of remote resources.* Projects that are intended to primarily work on network resources or local file system resources such as file packagers are no good candidates since they would require extensive stubbing and the creation of valid resource identifiers, such as file names.

### Selected Test Objects

The following projects were selected to provide test objects in accordance with the selection criteria from above:

- J2SDK, version 1.4.2\_08, the Java 2 Standard Development Kit,  
<http://java.sun.com/j2se/1.4.2/download.html>
- Quilt, version 0.6a5, a code coverage measurement tool working on Java byte code level,  
<http://quilt.sourceforge.net>
- JFreeChart, version 1.0.1, in conjunction with JCommons, version 1.0.4, a library for chart drawing,  
<http://www.jfree.org/jfreechart>
- Colt, version 1.2.0, a library for high performance and technical computing developed in the context of CERN (European organization for nuclear research),  
<http://dsd.lbl.gov/~hoschek/colt>
- Apache Commons Math, version 1.1, a library providing various data structures and algorithms for mathematical computing,  
<http://jakarta.apache.org/commons/math>

Table 4.1 on the following page lists the classes taken from the selected projects. The names of the classes are shown in the first column of the table. The other columns show the values of the following complexity metrics: ELOC is the *number of executable lines of code* which corresponds to the size of the class in terms of the source code. NOM means *number of methods*, while NOA means *number of attributes*. DIT is the *depth of*

Test object	ELOC	NOM	NOA	DIT	NBD	CYC
java.security.CodeSource	236	11	3	1	5	27
java.util.BitSet	486	32	2	1	3	33
java.util.HashMap	504	35	6	3	4	8
java.util.LinkedList	317	30	2	4	4	6
java.util.Stack	34	6	0	4	2	2
java.util.StringTokenizer	108	12	8	1	2	8
java.util.TreeSet	108	22	2	3	3	8
org.quilt.cover.stmt.Ephemera	48	6	3	1	2	6
org.quilt.framework.QuiltTest	194	50	22	1	3	6
org.quilt.graph.Directed	137	14	7	1	3	5
org.quilt.graph.Edge	74	9	2	1	2	6
org.quilt.graph.Entry	34	6	0	2	2	2
org.quilt.graph.Exit	30	6	0	2	2	2
org.quilt.graph.UnaryConnector	22	5	1	2	2	2
org.quilt.graph.Vertex	90	15	4	1	3	5
org.quilt.graph.Walker	96	4	6	1	5	12
org.quilt.reg.Registry	83	11	1	1	3	7
org.quilt.report.Msg	68	10	7	1	2	2
org.jfree.chart.JFreeChart	834	65	15	5	4	16
org.jfree.chart.axis.AxisSpace	186	19	4	1	3	9
org.jfree.chart.axis.NumberAxis	720	34	6	3	6	16
org.jfree.data.DefaultKeyedValues	150	18	1	1	4	9
org.jfree.data.DefaultKeyedValues2D	234	23	4	1	5	13
org.jfree.data.Range	144	11	2	1	3	4
org.jfree.data.time.TimeSeries	415	44	6	2	4	12
org.jfree.ui.RectangleInsets	255	28	5	1	2	8
org.jfree.util.ObjectTable	200	20	5	1	4	9
cern.colt.bitvector.BitMatrix	196	29	3	2	6	24
cern.colt.bitvector.BitVector	316	34	2	2	6	21
cern.colt.buffer.DoubleBuffer	35	5	5	2	2	2
cern.colt.matrix.impl.SparseDoubleMatrix1D	72	15	1	5	2	3
org.apache.commons.math.analysis.BrentSolver	105	3	0	2	4	16
org.apache.commons.math.analysis.SecantSolver	72	3	0	2	4	9
org.apache.commons.math.complex.Complex	141	14	2	1	4	10

Table 4.1: Test objects; general complexity metrics

*inheritance tree* and indicates how many super classes the class possesses. NBD is the *nested block depth*. CYC is McCabe's *cyclomatic complexity* (McCabe, 1976). Note that the table shows the maximum values for both NBD and CYC since each method of the class implies a value for these metrics.

Table 4.2 shows the aspects of the selected classes that relate to the limitations from Section 1.1 on page 3. Column *Loops* gives the number of loops occurring in the source

Test object	Loops	Arrays	Class Args	Cmplx Preds.	Non-Public	Abstr.
CodeSource	9	9	7/11	43	4/11	2
BitSet	22	73	7/39	129	11/39	0
HashMap	20	16	21/40	41	8/40	4
LinkedList	14	4	17/30	31	5/30	0
Stack	0	0	2/6	2	0/6	2
StringTokenizer	4	0	4/12	13	3/12	0
TreeSet	1	0	13/22	3	3/22	4
Ephemera	0	0	2/6	4	0/6	
QuiltTest	4	0	14/50	13	0/50	0
Directed	1	0	9/16	7	3/16	1
Edge	0	0	6/10	6	0/10	
Entry	0	0	2/6	2	0/6	1
Exit	0	0	2/6	1	1	1
UnaryConnector	0	0	2/5	1	0/5	1
Vertex	1	0	5/16	6	1/16	1
Walker	2	0	3/4	9	2/4	2
Registry	1	2	5/11	6	0/11	0
Msg	1	0	8/10	1	0/10	0
JFreeChart	7	4	36/66	63	6/66	5
AxisSpace	0	0	7/19	24	0/19	0
NumberAxis	2	0	20/38	54	11/38	2
DefaultKeyedValues	0	0	3/7	1	0/7	0
DefaultKeyedValues2D	6	0	10/24	23	0/24	2
Range	0	0	6/17	12	1/17	0
TimeSeries	8	0	23/44	47	0/44	2
RectangleInsets	0	0	8/28	22	0/28	0
ObjectTable	8	16	7/22	30	11/22	0
BitMatrix	7	2	9/29	32	4/29	1
BitVector	20	18	10/35	64	4/35	1
DoubleBuffer	0	1	2/5	3	0/5	
SparseDoubleMatrix1D	0	0	4/15	2	4/15	9
BrentSolver	1	0	1/3	13	0/3	1
SecantSolver	1	0	1/2	8	0/2	1
Complex	0	0	5/14	19	0/14	0

Table 4.2: Test objects; properties related to limitations

code (**for**, **while**, and **do-while**). Column *Arrays* gives the number of array accesses that occur in the source code. Column *Class args* gives the number of methods that possess at least one non-primitive argument type (in relation to the total number of methods). Column *Cmplx Preds.* gives the number of complex predicates. A predicate

is considered complex if it is composed using the **AND** or **OR** operator, or if it is a method call. Column *Non-Public* gives the number of non-public methods, in relation to the total number of methods. Finally, column *Abstr.* gives the number of interfaces or abstract classes that the class under test involves (directly or indirectly).

Table 4.3 shows the characteristics of the selected test objects that refer to the evolutionary search and are suggestive of the complexity of the search. It shows the

Test object	Branches	Types	Functions	IMPL	AG
CodeSource	40/0/70	19	62	2	2
BitSet	157/0/91	7	36	0	0
HashMap	47/0/9	18	145	4	1
LinkedList	57/0/11	12	59	0	1
Stack	8/0/0	16	118	2	1
StringTokenizer	12/0/17	8	20	0	0
TreeSet	24/0/3	17	144	4	1
Ephemera	13/0/0	8	16	0	0
QuiltTest	64/0/0	17	90	0	0
Directed	20/5/0	18	77	1	0
Edge	19/0/0	18	77	1	0
Entry	8/0/0	18	101	1	0
Exit	6/1/0	18	100	1	0
UnaryConnector	6/0/0	19	84	1	0
Vertex	22/1/0	18	77	1	0
Walker	30/0/0	21	87	2	0
Registry	18/0/0	11	30	0	1
Msg	14/0/0	10	23	0	0
JFreeChart	157/26/16	55	177	5	1
AxisSpace	63/0/0	10	41	0	0
NumberAxis	83/72/0	44	212	2	0
DefaultKeyedValues	44/0/0	15	43	0	0
DefaultKeyedValues2D	72/0/0	14	57	2	0
Range	38/0/4	7	22	0	0
TimeSeries	135/0/0	24	94	2	0
RectangleInsets	70/0/0	12	58	0	0
ObjectTable	36/21/8	8	20	0	0
BitMatrix	78/7/0	10	38	1	0
BitVector	129/6/0	12	50	1	1
DoubleBuffer	8/0/0	24	136	4	2
SparseDoubleMatrix1D	14/7/0	39	286	9	3
BrentSolver	27/0/0	7	24	1	0
SecantSolver	17/0/0	7	24	1	0
Complex	51/0/0	5	28	0	0

Table 4.3: Test objects; properties related to the evolutionary search

following metrics: *Branches* shows the number of branches to cover according to the method/decision coverage criterion (cf. Section 2.1.4 on page 11). This number estimates the number of individual optimizations that must be carried out by the test sequence generator. The number of branches is split up into the branches that belong to (a)



public methods, (b) protected methods, and (c) private methods. *Types* shows the number of types used by the genetic programming algorithm, while *Functions* shows the number of function set entries (cf. Section 2.3.3 on page 46). *IMPL* shows how many abstract classes and interfaces are involved, hence requiring the test sequence generator to automatically create implementers for them (cf. Section 3.6.2 on page 87). Finally, *AG* shows the number of array generators (cf. Section 3.6.3 on page 88) that the test sequence generator uses.

#### 4.2.2 Setup and Realization

The experimental framework for the general effectiveness case study comprises several tools and configurations. It was used to accomplish the following steps:

1. EvoUnit was used in optimizing mode to obtain JUnit test classes for the 34 test objects.
2. EvoUnit was used in random mode to obtain JUnit test classes for the 34 test objects.
3. CodePro was used to obtain JUnit test classes for the 34 test objects.
4. Jtest was used to obtain JUnit test classes for the 34 test objects.

EvoUnit creates coverage reports based on method/decision coverage. Therefore, the comparison between the evolutionary class testing approach and the random approach is based on the degree of achieved method/decision coverage. However, both CodePro and Jtest have particular implementations of coverage measurement. None of them measures method/decision coverage. Therefore, the independent coverage measurement tool clover (Cenqua Pty. Ltd., 2007) was used in order to generate comparable coverage reports. Clover measures method coverage (the relative amount of methods tested), statement coverage, conditional coverage (that is, decision coverage, but not method/decision coverage), and an accumulated coverage, referred to as *Clover coverage* in this thesis. Clover coverage *TC* is defined as follows:

$$TC = \frac{B_T + B_F + SC + MC}{2 \cdot B + S + M} \quad (4.1)$$

where  $B_T$  is the number of branches that evaluated to true at least once,  $B_F$  is the number of branches that evaluated to false at least once,  $SC$  is the number of statements covered,  $MC$  is the number of methods entered,  $B$  is the total number of branches,  $S$  is the total number of statements, and  $M$  is the total number of methods. Clover coverage was used as the decisive coverage criterion for the comparison of the approaches.

#### EvoUnit Setup and Realization

**Optimizing Mode** In general, the default settings of ECJ – the genetic programming system that EvoUnit uses – were kept unchanged for the experiments. While the ECJ

documentation (Luke, 2007) provides the detailed general settings, Table 4.4 summarizes the specific major parameters (refer to Section 2.3 on page 35 for an explanation of the parameters). Figure 4.2 shows the evolutionary *pipeline* that ECJ uses. The figure is

Parameter	Value
individuals	50 in 1 global population
max. generations	200
tree generation strategy	Uniform
max. tree size	17
selection strategy	tournament selection
tournament size	7
crossover strategy	subtree crossover
mutation strategies	ERC mutation, demotion, promotion
population update strategy	pure reinsertion

Table 4.4: Settings of the genetic programming system ECJ

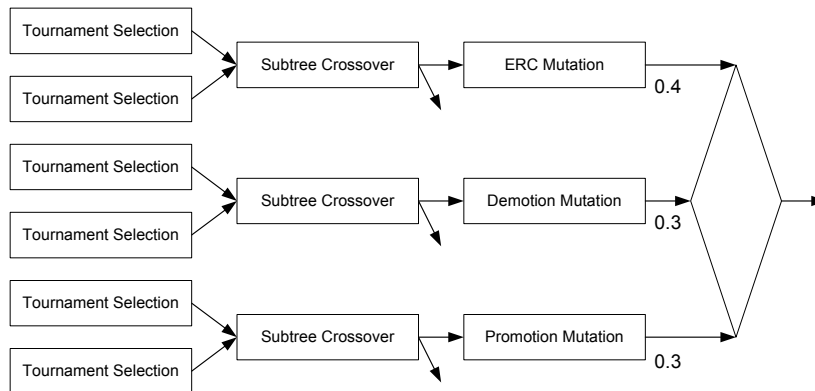


Figure 4.2: Experimental ECJ pipeline

best read from right to left: a new individual is obtained by using either ERC mutation, demotion mutation, or promotion mutation, respectively. The numbers at the edges indicate the probability of the choice of the corresponding mutation operator. These probabilities were chosen arbitrarily. ERC mutation was favored (value of 0.4) in order to explore the dimension of primitive values more intensively than the dimension of method calls. The “input” for the selected mutation operator is provided by the crossover operator. Note that although the crossover operator creates two offspring individuals, only one of them survives. The crossover operator acquires the parent individuals by using tournament selection.

A set of parameter-tuning experiments was carried out in order to find accurate values for the parameters *population size*, and *tournament size*, since these parameters appeared to have significant impact on the efficiency of the evolutionary search. Figure 4.3 shows

the summary of experiments performed with class `State1` (Listing A.3) for finding an accurate population size. Population sizes ranging from 10 to 150 individuals were tried. The task was to find a covering test sequence for the `true` branch of method `test` (line 24). Each run was carried out 50 times. Tournament size was set to 7 (ECJ's default). The figure shows the average number of objective function evaluations that indicate the efficiency of the search. The figure also shows the standard deviation achieved by the 50

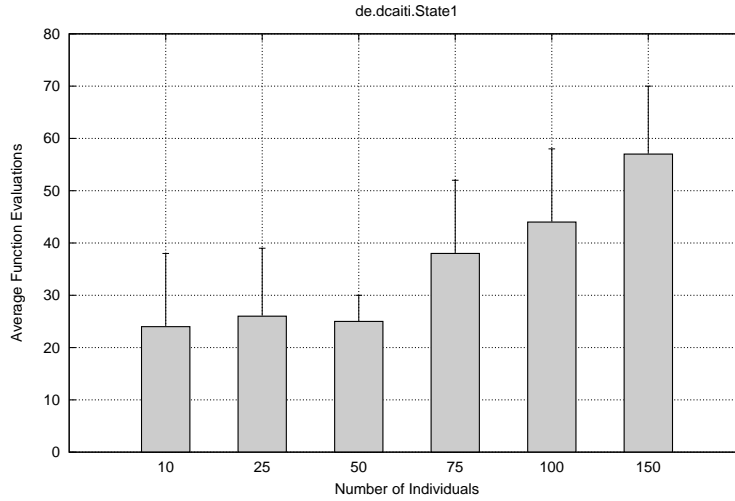


Figure 4.3: Results for parameter number of individuals

runs. The population size of 50 individuals achieved the best results in terms of required function evaluations and standard deviation. The results of experiments with varying tournament size are depicted in Figure 4.4 on the following page. Again, test object `State1` was used in 50 repeated runs. Population size was adjusted to 50. Tournament sizes were varied from 1 (which implies random selection) to 10. The chart shows the average number of objective function evaluations including the standard deviation. It is worth mentioning that the best tournament size 7 obtained from the experiments equals the default tournament size of ECJ. Although one parameter-setting experiment does not allow drawing general conclusions, the found parameters were used as starting points for the experiments.

**Random Mode** The random mode of EvoUnit was realized by switching off selection and variation. A new generation is then created by generating fresh individuals from scratch. The Uniform initializer is supposed to ensure that the probabilities of occurrence of the candidate test sequences is distributed equally.

**General Settings** Table 4.5 on the next page shows the settings for the various ERC types. Except for the types `integer` and `long`, the default data type ranges were used. The ranges of the two mentioned types were restricted in order to boost the evolutionary

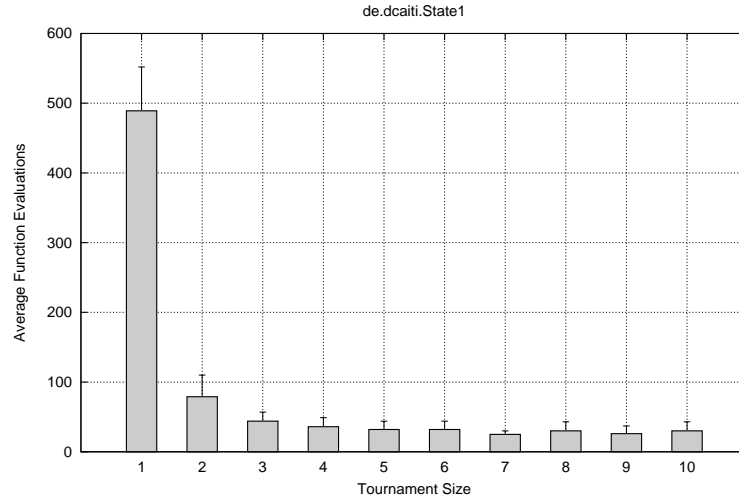


Figure 4.4: Results for parameter tournament size

ERC type	min. value	max. value
byte	-128	127
char	0	255
short	-32768	32767
integer	-100000	100000
long	-100000	100000
float	-3.4028235e+38f	3.4028235e+38f
double	-1.7976931348623157e+308	1.7976931348623157e+308

Table 4.5: ERC value ranges

search. These restrictions turned out to be convenient since the selected test objects do not require particular values beyond the specified ranges.

The execution times were also measured during the experiments. The test system was a PC with an Intel Pentium P4 CPU with 2.8 GHz and 2GB main memory. This PC ran Windows XP Professional 2002, Service Pack 2. Apart from the test sequence generator and the standard operating system maintenance processes no other applications were active on it.

EvoUnit was configured to run each test sequence generation process 50 times and report the results on each run. For some test objects, manual assignments of concrete classes to interfaces and to abstract classes were provided, along with method filters that prevent some of the classes' methods to be transformed to functions of the function set in order to keep the resulting function sets compact.

### **CodePro Setup and Realization**

The Eclipse plugin of CodePro Analytix, version 4.6.1, was used. CodePro allows adjusting several parameters. The only parameter which appears to have an impact on the degree of code coverage is the maximum allowed number of test sequences to generate per method of the class under test. By default, this value is set to 30, meaning that CodePro would generate at most 30 test sequences for a single method of the class under test if required. According to CodePro's documentation, a value of 0 should indicate that no such maximum limit exists, allowing CodePro to generate an unlimited number of test sequences. However, setting the value to 0 delivered empty test classes without any test method in preceding experiments. Preceding experiments also showed that the coverage of a sample class could be improved by increasing the parameter value from 30 to a 200. The value was set to 200 since the focus is primarily on code coverage and not on the number of test cases. Thereby, 200 appeared to be sufficiently high. Since the test-case-generating algorithm of CodePro is deterministic, exactly one run was performed.

### **Jtest Setup and Realization**

The Eclipse plugin of Jtest, version 8.0.122, was used. Similar to CodePro, Jtest allows specifying several parameters. However, the default settings were left unchanged except for the configuration of methods for which Jtest should generate test sequences: by default, private methods are not addressed; thus, the settings were adapted in order to make Jtest address them, too.

Jtest generated the same number of test sequences for a test object in repeated runs in preceding experiments with the test objects taken from the J2SDK project. Furthermore, the test sequences generated during different runs of Jtest led to exactly the same degrees of code coverages. Therefore, only one run was performed for each test object.

### 4.2.3 Results

At first, the results achieved by EvoUnit in optimizing mode are presented, followed by the results achieved by EvoUnit in random mode. Afterwards, the results of CodePro and Jtest are discussed.

#### EvoUnit Results

The results achieved by EvoUnit in optimizing mode are shown in Table 4.6 on the facing page. The values are averaged over the 50 runs (except the standard deviations in parentheses). Column *Test object* lists the names of the test objects. Column  $D^+COV$  shows the achieved method/decision coverage. Column  $\sigma_c$  gives the standard deviation of the coverage. Both  $D^+COV$  and  $\sigma_c$  give values in percent. Column *Evals* shows the average number of objective function evaluations. Column  $\sigma_e$  shows the corresponding standard deviation. Column *Time* shows the time needed on average in the format `minutes:seconds`. Column  $\sigma_t$  shows the corresponding standard deviation in seconds. When required, values were rounded according to the standard rounding rules.

The results demonstrate that EvoUnit was able to generate test sequences that lead to high code coverages in most cases. Note that the coverage relates to all methods of the classes, not only to the public ones. Relatively high coverages were obtained, ranging from 65.7% (Edge) to 100% (Stack, Entry, Exit, Registry, Msg, SecantSolver). On average, 91.6% were achieved. The relatively small standard deviations suggest the stability of the results and hence the stability of the approach in general. The number of objective function evaluations ranges from 10 (Exit) to 390,146 (BitVector). A higher number indicates a greater difficulty of the test goals associated with a test object. The execution times range from one second (Registry, Msg), suggesting that the search was trivial, to 201 minutes (NumberAxis). However, the execution times do not always correlate to the number of objective function executions (longer execution times do not necessarily indicate higher numbers of evaluations). This is due to the fact that the execution time of one evaluation differs from test goal to test goal and is different for each test object. For instance, test objects with a large number of attributes that require intensive memory management, such as in the case of class BitSet, one evaluation takes a relatively long time.

An analysis of the results suggests the following reasons for uncovered test goals:

In many cases, the classes possess private serializing methods (`readObject` and `writeObject`) for which no call points exist since the methods are called by the Java serialization framework. The presence of these methods that sometimes include several decisions, decreases the achievable degree of coverage.

Some classes possess protected methods that they do not call themselves. These methods are provided to subclasses to allow access to some internals (for example, in the case of `ObjectTable`). Test goals that belong to such protected methods could not be covered since no call points exist.

Predicates are present primarily in the mathematical test objects that involve exact comparisons of floating point values of double precision. Finding an exactly matching

Test object	$D^+$ -COV	$\sigma_c$	Evals	$\sigma_e$	Time	$\sigma_t$
CodeSource	87.0	(1.3)	152853	(13457)	31:38	(284)
BitSet	98.5	(0.5)	56673	(12723)	26:35	(707)
HashMap	89.1	(1.1)	102593	(9685)	09:42	(140)
LinkedList	98.3	(0.6)	27398	(4936)	01:23	(45)
Stack	100.0	(0.0)	946	(967)	00:02	(2)
StringTokenizer	93.7	(2.4)	23299	(8423)	01:38	(34)
TreeSet	92.7	(0.7)	19803	(1124)	01:02	(13)
JFreeChart	92.9	(0.3)	141911	(5213)	47:58	(337)
AxisSpace	98.4	(0.0)	10041	(17)	00:18	(2)
NumberAxis	95.4	(2.2)	73983	(2512)	201:00	(913)
DefaultKeyedValues	92.6	(2.0)	35868	(7472)	01:24	(19)
DefaultKeyedValues2D	86.4	(3.3)	99162	(22724)	03:04	(43)
Range	97.6	(0.0)	10006	(5)	00:26	(9)
TimeSeries	89.7	(1.0)	119658	(10150)	06:07	(41)
RectangleInsets	99.4	(0.9)	13918	(6508)	01:44	(36)
ObjectTable	80.0	(0.0)	120000	(1)	03:10	(6)
Ephemera	92.3	(0.0)	10096	(33)	00:17	(1)
QuiltTest	96.9	(0.0)	21511	(663)	01:42	(7)
Directed	82.0	(3.3)	45558	(7329)	02:45	(19)
Edge	65.7	(3.7)	67223	(5894)	03:46	(17)
Entry	100.0	(0.0)	13	(6)	00:00	(1)
Exit	100.0	(0.0)	10	(6)	00:00	(1)
UnaryConnector	83.3	(0.0)	10288	(395)	00:23	(1)
Vertex	87.0	(0.0)	30000	(0)	00:52	(6)
Walker	70.7	(1.4)	87800	(4185)	03:17	(10)
Registry	100.0	(0.0)	226	(72)	00:01	(1)
Msg	100.0	(0.0)	31	(13)	00:01	(1)
DoubleBuffer	99.8	(1.8)	4843	(2839)	02:03	(74)
BitVector	71.1	(0.0)	390146	(145)	11:44	(9)
BitMatrix	97.2	(0.7)	33735	(5701)	01:06	(12)
SparseDoubleMatrix1D	90.5	(0.0)	20000	(0)	33:15	(210)
BrentSolver	96.3	(0.0)	10190	(213)	01:36	(4)
SecantSolver	100.0	(0.0)	1208	(1120)	00:05	(6)
Complex	90.2	(0.3)	50013	(665)	02:56	(59)
Average	91.6	(0.8)	52677	(3976)	11:51	(90)

Table 4.6: Results from EvoUnit (optimizing mode)

Test object	$D^+$ -COV	$\sigma_c$	Evals	$\sigma_e$	Times	$\sigma_t$
CodeSource	69.5	(0.8)	347248	(10191)	05:25	(17)
BitSet	93.4	(0.8)	198155	(16276)	53:15	(303)
HashMap	76.2	(1.8)	237531	(15906)	04:59	(23)
LinkedList	90.6	(1.0)	83901	(7082)	08:08	(18)
Stack	100.0	(0.0)	1291	(1174)	00:03	(2)
StringTokenizer	96.2	(1.0)	16122	(3307)	01:01	(14)
TreeSet	91.6	(1.7)	25509	(4150)	01:20	(14)
JFreeChart	71.3	(1.3)	141911	(5213)	44:24	(205)
AxisSpace	98.4	(0.0)	10056	(23)	00:12	(0)
NumberAxis	79.8	(0.7)	314713	(10245)	338:48	(438)
DefaultKeyedValues	83.9	(1.4)	66910	(5190)	01:40	(9)
DefaultKeyedValues2D	62.5	(1.3)	267017	(8405)	06:44	(17)
Range	97.0	(1.2)	18426	(4892)	00:24	(7)
TimeSeries	66.4	(1.5)	378990	(15192)	13:21	(32)
RectangleInsets	76.1	(1.9)	168503	(13440)	01:13	(6)
ObjectTable	78.5	(1.1)	132252	(5642)	01:47	(9)
Ephemera	92.3	(0.0)	11442	(1168)	00:17	(2)
QuiltTest	89.7	(1.1)	73340	(5273)	01:46	(5)
Directed	83.3	(3.4)	44425	(6481)	01:38	(6)
Edge	72.3	(4.2)	75733	(7390)	00:23	(3)
Entry	100.0	(0.0)	12	(6)	00:00	(1)
Exit	100.0	(0.0)	9	(4)	00:00	(0)
UnaryConnector	89.3	(8.1)	7984	(3320)	00:06	(2)
Vertex	82.9	(3.2)	40828	(7265)	00:19	(4)
Walker	67.5	(2.0)	97400	(5997)	02:52	(7)
Registry	100.0	(0.0)	738	(433)	00:02	(1)
Msg	100.0	(0.0)	31	(11)	00:15	(61)
DoubleBuffer	89.0	(4.8)	11804	(2688)	03:17	(41)
BitVector	69.0	(0.5)	419129	(6273)	12:35	(15)
BitMatrix	80.0	(1.9)	154294	(13992)	04:41	(26)
SparseDoubleMatrix1D	90.5	(0.0)	20000	(0)	189:55	(260)
BrentSolver	37.3	(24.0)	171402	(63373)	06:28	(143)
SecantSolver	29.8	(18.0)	120559	(28888)	05:33	(66)
Complex	90.2	(0.0)	50080	(63)	01:43	(11)
Average	82.2	(2.6)	109051	(8205)	21:02	(52)

Table 4.7: Results from EvoUnit (random mode)



double input is an extremely hard task due to the huge search space in the case of double parameters and the mutation precision. This parameter indirectly specifies the smallest mutation step of a double value during the evolutionary search. However, all practical precision values do not allow the smallest possible mutation step size that might be required to evolve a particular double value. This issue is also known in the area of evolutionary structural testing of procedural software.

Additionally, some of the test goals are infeasible, meaning that no test sequence exists that attains it. Infeasible test goals can have several reasons. Often, they are caused by logical programming errors or “double checks”; for instance: a setter method of a class checks whether the passed argument is the `null` reference. If it is, it declines setting the attribute; otherwise it sets the attribute value. Another method that uses the attribute value checks again for a `null` value regardless of that the value cannot be `null` due to the previous check of the setter method. This case occurs, for instance, in class `Range`.

Finally, some predicates were either too complex or the corresponding distance functions were not helpful in guiding the evolutionary search. Since many address comparisons are involved in the decisions of the test objects, the resulting objective function contains plateaus due to the binary character of the assigned distance function for address comparisons. Furthermore, several conditions require some objects – either the object under test or needed parameter objects – to be in a particular state that is very hard to attain. In these cases the approach did not succeed in setting up the required states of the instances.

Table 4.7 on the preceding page shows the results from EvoUnit running in random mode. The table has the same structure as Table 4.6 on page 117. On average, 9.4% less coverage was achieved by the random mode. This suggests that evolutionary optimization actually occurred and contributed to the attainment of some test goals. However, the relatively high coverage achieved by the random mode suggests that many test goals are trivial and random testing is also able to produce relatively good results. Both the average number of objective function evaluation and the average execution time are about the double of those of the optimizing mode. The number of evaluations ranges from 9 (Exit) to 419,129 (BitVector). The execution time ranges from under 1 second (Entry and Exit) to 338 minutes and 48 seconds (NumberAxis).

The following figures contrast the coverages achieved by both EvoUnit running in optimizing mode (labeled “EVO” in the figures) and EvoUnit running in random mode (labeled “RND” in the figures): Figure 4.5 on the next page shows the coverages for the test objects of the J2SDK project, Figure 4.6 on the following page shows the coverages for the test objects of the Quilt project, Figure 4.7 on page 121 shows the coverages for the test objects of the JFreeChart project, and Figure 4.8 on page 121 shows the coverages for the test objects of the Colt project and the Apache Math project.

The figures also show the result of the t-test by two stars, one star, or no star respectively, above each test object: two stars indicate that the assumption that the approach which achieved a higher coverage during the 50 runs *in fact* outperforms the other approach is accurate with a probability of 99.9% (statistically *highly significant*).

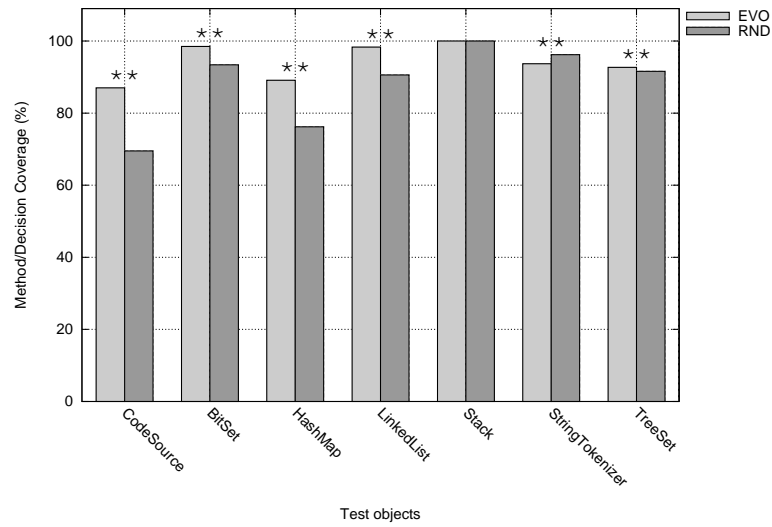


Figure 4.5: Coverage achieved by EvoUnit and random generator; J2SDK test objects

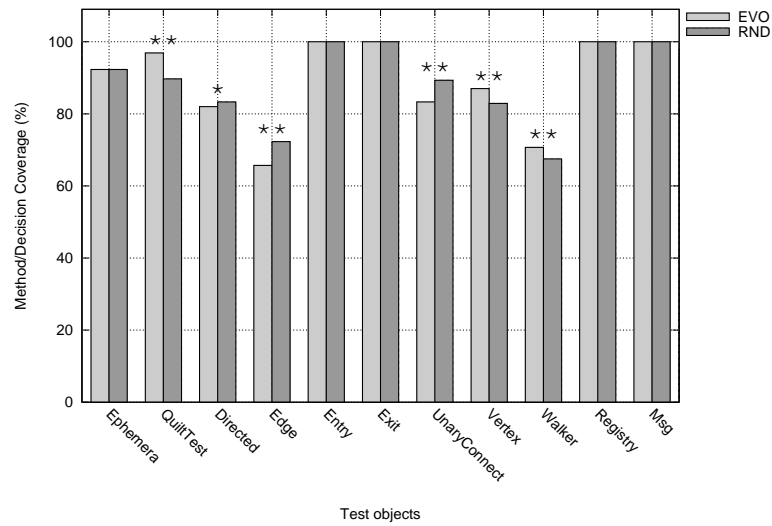


Figure 4.6: Coverage achieved by EvoUnit and random generator; Quilt test objects

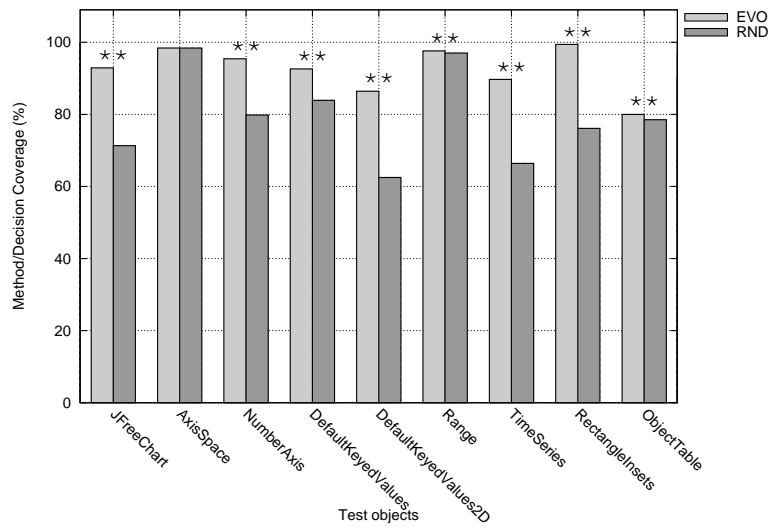


Figure 4.7: Coverage achieved by EvoUnit and random generator; JFreeChart test objects

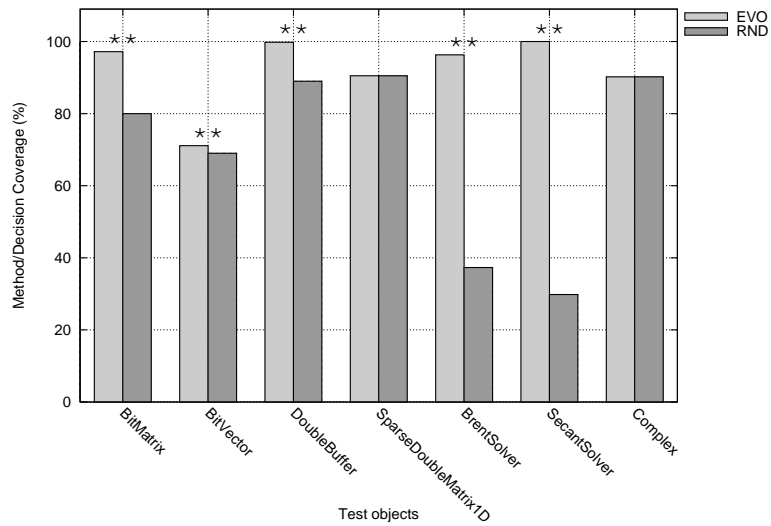


Figure 4.8: Coverage achieved by EvoUnit and random generator; both Colt and Math test objects

One star indicates that the same assumption is accurate with a probability of 99.5% (statistically *significant*). No star indicates that the probability of the accuracy is lower than 99.5%. The results suggest that the evolutionary approach outperforms the random approach in general. As can be seen in the figures, in most of the cases the differences between the two approaches are highly statistically significant. The significance indicates the stability of the results: another repetition of a run is very likely to produce the same result. Yet, the results also indicate that random testing performs relatively well, too.

However, there are test objects for which the same results are achieved and also for which the random approach was even better than the evolutionary approach (4 test objects). The latter finding is surprising in particular. Harman and McMinn (2007) also observe that random testing outperforms evolutionary testing in rare cases. An investigation into the 5 test goals of the 4 affected test objects found that the objective functions are deceptive in these cases, preventing the evolutionary search from exploring search space regions containing covering test sequences. These test objects are: StringTokenizer, Directed, Edge, and UnaryConnector. The affected test goals are state-dependent, meaning that the instances must be in particular states. However, attaining the states requires method calls that potentially throw runtime exceptions. However, a simple test sequence will be rewarded in these cases while test sequences that include many calls to state-changing methods tend to be penalized due to greater distance values caused by runtime exceptions. It is assumed that the chaining approach (McMinn, 2004a) will help in these situations since it addresses state-dependent test goals. However, this approach must be adapted to object-oriented software since it was developed for procedural software only.

### CodePro Results

Table 4.8 on the next page contrasts the coverages achieved by both EvoUnit and CodePro. The coverages were reported by the Clover tool when (1) executing that test class generated by EvoUnit, whose achieved method/decision coverage occurred most frequently during the 50 runs, and (2) executing the only test class generated by CodePro.

The differences between the method/decision coverage from Table 4.6 on page 117 and the Clover conditional coverage are due to the different definitions of the metrics and the fact that the former does not account for “short-hand” conditionals, such as  $x = (a == 0 ? 1 : -1)$ . Column *Test object* shows the names of the test objects. Columns *#* show the number of generated test sequences. Columns *MC* show the achieved method coverage. Columns *SC* show the achieved statement coverage. Columns *CC* show the achieved conditional coverage. Columns *TC* show the achieved total Clover coverage.

As the table shows, EvoUnit reached 30.5% higher Clover coverage than CodePro on average. Note that in three cases, CodePro was not able to generate any test sequence due to a runtime exception in the tool: for Registry, NumberAxis, and Complex. When excluding these test objects, EvoUnit reached 19.6% higher Clover coverage than CodePro on average. The table also shows that although CodePro generated one test sequence for class `BrentSolver` and for class `SecantSolver`, respectively, these test sequences do not

Test object	EvoUnit					CodePro				
	#	MC	SC	CC	TC	#	MC	SC	CC	TC
CodeSource	112	81.8	65.8	74.4	70.7	19	63.6	26.0	23.6	26.6
BitSet	254	97.4	98.2	98.5	98.3	96	82.1	41.9	31.3	40.1
HashMap	90	95.0	86.4	86.5	87.4	279	92.5	55.0	35.4	53.9
LinkedList	69	93.3	90.4	92.4	91.4	102	93.3	52.5	30.3	51.3
Stack	8	100.0	100.0	100.0	100.0	8	100.0	75.0	50.0	76.9
StringTokenizer	29	100.0	100.0	96.4	99.0	46	100.0	98.4	89.3	96.2
TreeSet	27	90.9	75.0	30.0	73.8	119	90.9	60.4	10.0	62.5
BitMatrix	85	96.6	98.4	97.3	97.8	1610	100.0	74.4	50.0	69.7
BitVector	135	100.0	84.0	70.5	80.6	869	100.0	66.7	56.8	66.0
DoubleBuffer	8	100.0	100.0	100.0	100.0	20	100.0	82.4	50.0	78.6
SparseDoubleMatrix1D	21	86.7	78.8	60.0	77.6	115	60.0	63.6	60.0	62.1
Ephemera	13	100.0	94.7	87.5	93.9	8	100.0	52.6	12.5	51.5
QuiltTest	64	100.0	97.1	93.3	97.3	213	100.0	85.7	43.3	82.7
Directed	25	100.0	66.7	64.3	72.8	47	100.0	76.5	42.9	75.3
Edge	19	70.0	63.6	64.3	64.9	17	100.0	97.0	100.0	98.2
Entry	8	100.0	100.0	100.0	100.0	1	16.7	13.3	25.0	16.0
Exit	7	100.0	100.0	100.0	100.0	8	100.0	91.7	50.0	90.0
UnaryConnector	6	80.0	85.7	100.0	85.7	1	20.0	28.6	50.0	28.6
Vertex	23	100	84.8	75.0	86.9	205	100.0	78.8	58.3	80.3
Walker	30	100.0	76.3	50.0	68.8	7	100.0	74.6	46.7	66.7
Registry	18	100.0	100.0	100.0	100.0	0	0.0	0.0	0.0	0.0
Msg	14	100	100.0	100.0	100.0	493	100.0	100.0	100.0	100.0
AxisSpace	64	100.0	98.8	98.0	98.7	1132	100.0	86.9	74.0	84.3
NumberAxis	151	97.4	81.4	90.2	84.0	0	0.0	0.0	0.0	0.0
JFreeChart	179	93.9	85.3	81.5	85.2	3555	97.0	81.0	64.8	78.3
DefaultKeyedValues2D	71	100.0	95.2	85.5	92.9	1083	100.0	85.6	75.8	84.3
DefaultKeyedValues	41	100.0	98.6	88.9	96.1	677	100.0	94.6	77.8	90.6
Range	42	100.0	100.0	97.1	99.1	291	100.0	76.9	67.6	77.6
TimeSeries	111	100.0	93.0	84.4	91.5	2058	97.7	44.7	33.3	48.2
RectangleInsets	70	100.0	100.0	98.4	99.6	848	100.0	81.6	71.9	81.1
ObjectTable	60	60.0	51.5	46.4	50.8	135	85.0	62.4	48.3	60.3
BrentSolver	27	100.0	98.6	92.3	97.0	1	0.0	0.0	0.0	0.0
SecantSolver	17	100.0	97.7	93.8	96.8	1	0.0	0.0	0.0	0.0
Complex	51	100.0	92.5	86.8	91.6	0	0.0	0.0	0.0	0.0
Average	57	95.4	89.4	84.8	89.1	414	76.4	59.1	45.0	58.2

Table 4.8: Clover results for EvoUnit and CodePro

increase code coverage since they both provoke a runtime exception in the constructors of the super classes of the respective classes.

### Jtest Results

Table 4.9 contrasts the coverages achieved by both EvoUnit and Jtest. The coverages were reported by the Clover tool when (1) executing that test class generated by EvoUnit, whose achieved method/decision coverage occurred most frequently during the 50 runs, and (2) executing the only test class generated by Jtest. In general the test sequences

Test object	EvoUnit					Jtest				
	#	MC	SC	CC	TC	#	MC	SC	CC	TC
CodeSource	112	81.8	65.8	74.4	70.7	43	100.0	64.4	57.5	63.1
BitSet	254	97.4	98.2	98.5	98.3	112	100.0	80.8	70.9	78.1
HashMap	90	90.0	72.7	68.8	73.6	101	97.5	70.0	46.9	66.9
LinkedList	69	93.3	90.4	92.4	91.4	67	100.0	74.3	56.1	72.4
Stack	8	100.0	100.0	100.0	100.0	10	100.0	100.0	100.0	100.9
StringTokenizer	29	100.0	100.0	96.4	99.0	25	100.0	92.2	78.6	89.4
TreeSet	27	90.9	75.0	30.0	73.8	51	100.0	95.8	50.0	91.3
BitMatrix	85	96.6	98.4	97.3	97.8	49	100.0	49.8	38.4	50.0
BitVector	135	100.0	84.0	70.5	80.6	61	100.0	60.0	36.5	57.5
DoubleBuffer	8	100.0	100.0	100.0	100.0	9	100.0	82.4	66.7	82.1
SparseDoubleMatrix1D	21	86.7	78.8	60.0	77.6	38	100.0	100.0	100.0	100.0
Ephemera	13	100.0	94.7	87.5	93.9	10	100.0	89.5	75.0	87.9
QuiltTest	64	100.0	97.1	93.3	97.3	79	100.0	96.2	86.7	95.7
Directed	25	100.0	66.7	64.3	72.8	5	62.5	41.2	14.3	40.7
Edge	19	70.0	63.6	64.3	64.9	19	50.0	51.5	50.0	50.9
Entry	8	100.0	100.0	100.0	100.0	2	50.0	60.0	50.0	56.0
Exit	7	100.0	100.0	100.0	100.0	2	50.0	50.7	0.0	45.0
UnaryConnector	6	80.0	85.7	100.0	85.7	1	20.0	28.5	50.0	28.6
Vertex	23	100	84.8	75.0	86.9	4	31.2	24.2	16.7	24.6
Walker	30	100.0	76.3	50.0	68.8	4	100.0	37.3	13.3	32.3
Registry	18	100.0	100.0	100.0	100.0	22	100.0	100.0	100.0	100.0
Msg	14	100	100.0	100.0	100.0	11	100.0	100.0	60.0	91.3
AxisSpace	64	100.0	98.8	98.0	98.7	30	100.0	79.8	58.0	75.2
NumberAxis	151	97.4	81.4	90.2	84.0	35	57.9	30.1	25.4	30.9
JFreeChart	179	93.9	85.3	81.5	85.2	35	100.0	76.4	66.7	76.4
DefaultKeyedValues2D	71	100.0	95.2	85.5	92.9	59	100.0	85.6	72.6	83.3
DefaultKeyedValues	41	100.0	98.6	88.9	96.1	32	100.0	74.3	50.0	71.1
Range	42	100.0	100.0	97.1	99.1	21	94.1	78.5	73.5	79.3
TimeSeries	111	100.0	93.0	84.4	91.5	103	100.0	83.7	77.1	83.9
RectangleInsets	70	100.0	100.0	98.4	99.6	42	100.0	64.7	48.4	64.5
ObjectTable	60	60.0	51.5	46.4	50.8	57	100.0	92.1	79.3	88.8
BrentSolver	27	100.0	98.6	92.3	97.0	5	100.0	17.1	3.8	16.2
SecantSolver	17	100.0	97.7	93.8	96.8	5	100.0	22.7	6.2	22.2
Complex	51	100.0	92.5	86.8	91.6	31	100.0	74.6	57.9	72.3
Average	57	95.2	89.0	84.3	88.7	32	88.6	68.5	54.0	66.7

Table 4.9: Clover results for EvoUnit and Jtest

generated by EvoUnit achieved higher coverages than those of Jtest. More precisely, on average, EvoUnit's test sequences cover 22% more of the code in terms of Clover

coverage than Jtest's test sequences do. For 29 out of 34 test objects EvoUnit was more effective, whereas for 2 out of 34 test objects, Jtest was more effective.

### Summary

The case study with 34 test objects demonstrated the effectiveness of the evolutionary class testing approach, implemented in the test sequence generator EvoUnit. In comparison with random testing, evolutionary class testing achieved higher code coverages on average. In comparison with the two commercial test sequence generators CodePro and Jtest, EvoUnit achieved higher code coverages on average. Figures 4.9 to 4.12 summarize the coverage results of the three generators. It must, however, be noted that no

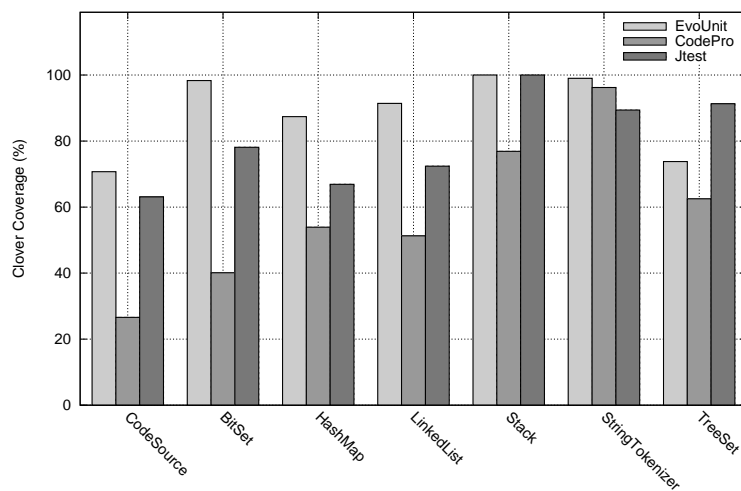


Figure 4.9: Coverage achieved by all generators; J2SDK test objects

expert knowledge was available for the application of the two commercial tools. Their configuration occurred based on the available documentation and preliminary tests with various parameter values.

## 4.3 Non-Public Method Coverage Case Study

This section reports on the results of an investigation into the coverage of non-public methods. The investigation should provide insight as to the effectiveness of the concept to address test goals belonging to non-public methods, as described in Section 3.5.6 on page 83.

An additional series of experiments was carried out for a subset of the test objects from Table 4.1 on page 108. All considered test objects possess non-public methods. For the accomplishment of the experiments, EvoUnit was used in a mode where it ignored test goals belonging to non-public methods. The results should reveal if the coverage of

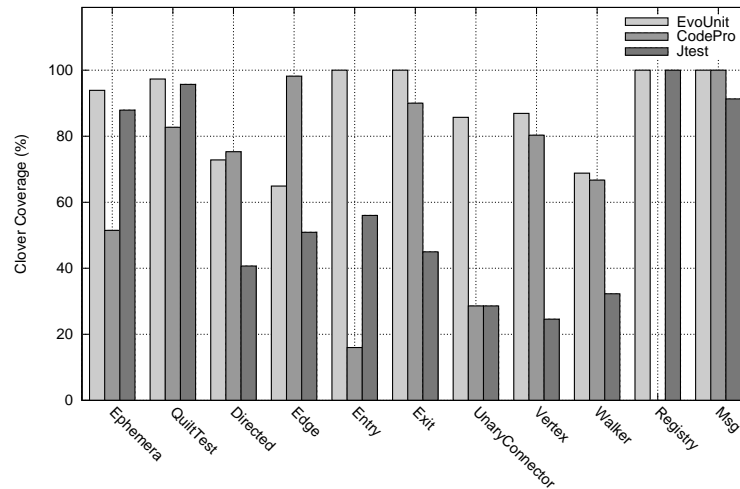


Figure 4.10: Coverage achieved by all generators; Quilt test objects

the non-public methods achieved by EvoUnit actually require the additional concept of the call points, or if non-public methods are covered in the process anyway.

Section 4.3.1 lists the test objects used for the case study. Section 4.3.2 describes the setup and the realization of the experiments, whereas Section 4.3.3 discusses the achieved findings.

### 4.3.1 Test Objects

From the selection of the 34 test objects used for the general effectiveness case study, the classes with non-public methods were selected: `CodeSource`, `BitSet`, `StringTokenizer`, `LinkedList`, `TreeSet`, `Directed`, `Exit`, `Vertex`, `Range`, `BitVector`, and `BitMatrix`. Table 4.3 on page 110 shows the number of test goals belonging to non-public methods for each of them.

### 4.3.2 Setup and Realization

The configuration of EvoUnit described in Section 4.2.2 on page 111 was used to run the experiments. Each run was carried out 50 times.

### 4.3.3 Results

The results of the comparison are shown in Figure 4.13 on page 129. The figure shows the relative degree of method/decision coverage achieved for the non-public methods. Full coverage (100%) means that all decisions of all non-public methods were covered and each non-public method has been entered at least once during test execution. As the figure shows, EvoUnit achieved higher coverages or equal coverages when it was run with



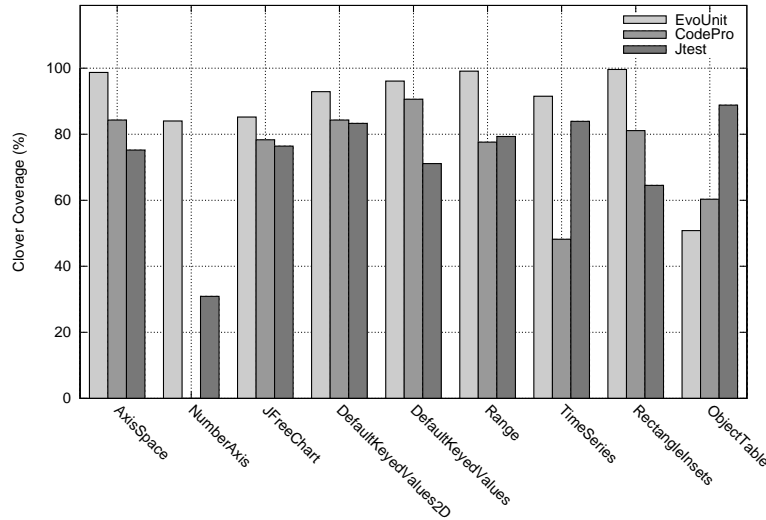


Figure 4.11: Coverage achieved by all generators; JFreeChart test objects

the support for non-public methods activated. Equal coverages were achieved for the three test objects `Exit`, `Vertex`, and `Range`. This indicates that the non-public methods can be tested in the process while testing the public methods. As expected, no case exists in which the coverage achieved without the support of non-public methods was better than with this support. In general, the results suggest that the integration of the concept of call points and the respective extension of the objective functions effectively contribute to the attainment of higher code coverages in the presence of non-public methods.

## 4.4 Function-Assigned Flag Case Study

This section describes the case study which was performed in order to assess the effectiveness of the *Boolean variable substitution*, suggested in Section 3.7.3 on page 93. The case study consists of two series of experiments performed with class `Stack` (shown in Listing A.4: at first, EvoUnit used the original version of the code to create high coverage test sequences; then, EvoUnit used the modified version.

Section 4.4.1 introduces the `Stack` class and points out the problems that the evolutionary class testing approach would have with the original source code. Section 4.4.2 on page 129 summarizes the setup of the experiments. Finally, Section 4.4.3 on page 129 discusses the results of the case study.

### 4.4.1 Test Object

Listing A.4 shows the source code of class `Stack`. This class realizes a simple stack which is able to hold `MAX_ELEMENTS` elements. Initially, the capacity is adjusted to

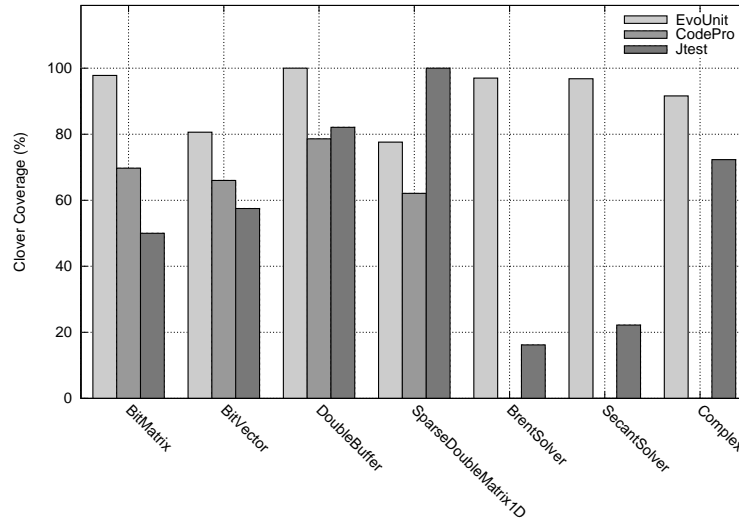


Figure 4.12: Coverage achieved by all generators; both Colt and Math test objects

10, meaning that 10 elements can be pushed onto a stack object at most. The internal implementation maintains both an array which actually contains the stack elements and a stack pointer which points to the next free index within this array. Method `add` adds an element to the stack if it is not already full. If it is called on a full stack, it throws an exception. Method `removeTop` removes the top element from the stack and returns it. If it is called on an empty stack, it throws an exception. Method `isFull` indicates whether the maximum capacity has been reached, while method `isEmpty` indicates whether the stack is empty.

The critical branch for which test sequence generation is hard is the `true` branch of the first condition of method `add` (line 11). This is due to the *information loss* caused by the method call `isFull`: the Boolean return value, which depends on the size of the array and the current pointer value to the next free element, implies a plateau in the objective function landscape, hindering the evolutionary search. A similar code construct is present at the first condition of method `removeTop` (the same kind of information loss occurs); however, since a new stack instance represents initially an empty stack, the condition can easily be satisfied as desired.

Listing A.5 shows the transformed version of the source code of class `Stack` (class `StackT`). The transformation modified the code in that it replaced the `Boolean` return type of both methods `isFull` and `isEmpty` with type `double`. Accordingly, it replaced the Boolean constants originally returned by these methods with the local objective values. Finally, it adapted the conditions which incorporate calls to these two methods. Note that for local objective value calculation, the code uses a shorthand notation: the expression in angle brackets is thought to abbreviate the respective operator-specific distance function.

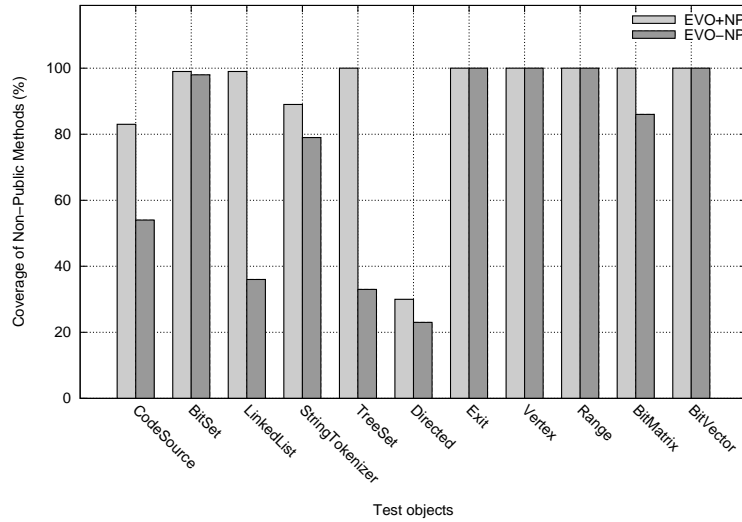


Figure 4.13: Coverage of non-public methods

#### 4.4.2 Setup and Realization

EvoUnit was used to run two series of experiments: 10 times the original code version was used, while 10 times the transformed version was used. EvoUnit was configured to only search for a test sequence which covers the true branch of the first condition of method `add`. The configuration of the genetic programming algorithm is the same as that of the effectiveness case study, described in Section 4.2.2 on page 111.

#### 4.4.3 Results

EvoUnit did not succeed in creating a covering test sequence for the targeted branch in any of the 10 runs when using the original code version (number of objective function evaluations: 10000, standard deviation: 0). In contrast, when it used the modified version it always found a covering test sequence (number of objective function evaluations: 1261, standard deviation: 786).

Figure 4.14 on the next page shows the development of the objective values during the 10 evolutionary searches with the transformed version. The thick graph represent the best objective value averaged over the 10 runs. It can be observed that the objective values incrementally improved which indicates that an optimization process took place. However, the initial objective value improvements are very small (due to the calculations of the `map` function) and cannot easily be seen in the figure. The final jumps of each graph of the single runs are due to the change of the sign of the “flag value”: once the relevant condition is satisfied, the sign of the flag value is inverted. Consequently, the distance function of the `>` operator returns the distance value 0, causing the best fitness to immediately change to 0.

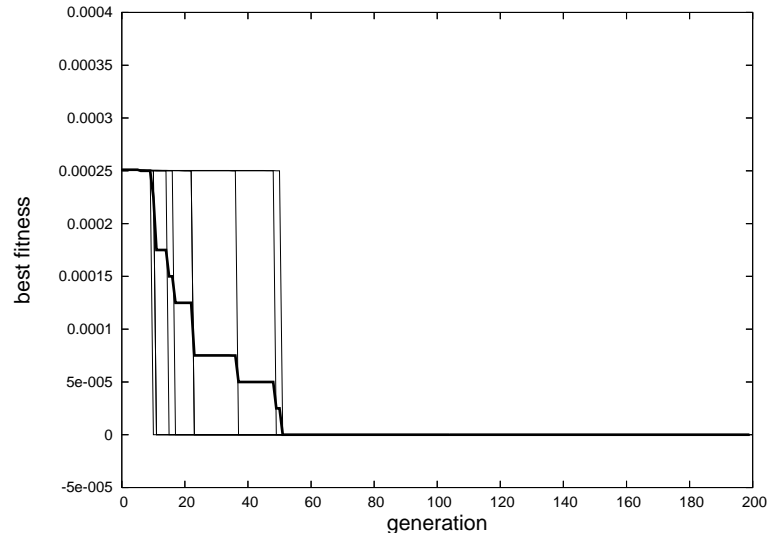


Figure 4.14: Objective value development for transformed Stack

## 4.5 Summary

This chapter presented three case studies for the empirical investigation into (1) the effectiveness of the evolutionary class testing approach in general, (2) the effectiveness of the approach in addressing the test of non-public methods, and (3) the effectiveness of objective function landscape smoothing, as described in Section 3.7 on page 88.

In the experiments for general effectiveness assessment, the evolutionary class testing approach implemented by the test sequence generator EvoUnit, successfully generated high coverage test sequences for the 34 real-world test objects taken from five different open-source development projects. Full code coverage was easily achieved in case the test objects were of low complexity. In many cases, full coverage was not possible due to infeasible branches and non-public methods for which no call points exist. On average, EvoUnit achieved higher code coverages when compared to a random testing approach. The same is true when comparing the results of EvoUnit with the results delivered by the two commercial test sequence generators CodePro and Jtest.

The second case study analyzing the coverage of non-public methods in particular indicated that the extension of the objective functions as suggested in Section 3.5.6 on page 83 is effective: without the extension, the coverage of the non-public methods achieved for 10 test objects was lower than with the extension being active. This suggests the value of the extension in general.

The case study investigating the effectiveness of the Boolean variable substitution method for smoothing the objective function landscape suggests that the method is suited to improving the evolutionary search in the presence of function-assigned flags. However, since the case study considered one problem case only, more experimentation is required to draw general conclusions.

The settings of the genetic programming algorithms used, such as the population size or the probability of mutation, were kept unchanged throughout the case studies. It is expected that the efficiency of the approach can be increased for some test objects if the settings are adapted individually.



## 5 Conclusion and Future Work

This chapter concludes this thesis by summarizing the achievements of this work in Section 5.1, discussing the restrictions and limitations of the evolutionary class testing approach in Section 5.2 on the next page, and finally giving directions for future work in Section 5.3 on page 136.

### 5.1 Summary of Achievements

The major objective of this thesis was the development of an approach to automatically generating object-oriented unit tests. The motivation for a new approach arose from the limitations of the existing work in this area. These limitations impact the applicability of the respective approach (meaning that only particular types of classes can be handled effectively) the achievable code coverage (meaning that the degree of coverage is suboptimal), and the maintainability of the results. This thesis suggested the evolutionary class testing approach to address the limitations outlined in Section 1.1 on page 3.

Evolutionary class testing is a dynamic and search-based approach that does not apply symbolic execution and constraint solving. Hence, it does not suffer from the limitations of these techniques. It generates test sequences that include the creation of class type arguments for method calls. The generated test sequences include calls to the public methods of the concerned classes only and do not involve breaking encapsulation by accessing non-public class members. As a result, the generated tests are better maintainable: they do not require the reflection mechanisms provided by a particular testing framework, nor will they be broken if refactorings involving non-public members are performed. Using the public class interfaces only ensures that the tests exclusively use objects in states that comply with their class invariants.

The representations used to encode test sequences to allow for evolutionary searches are defined in such a way that inexecutable test sequences rarely occur. Test sequences are encoded as method call trees which regard the call dependences among the methods (cf. Section 3.3 on page 56 and Section 3.4 on page 72). In general, the representations ensure that the genetic operators for crossover and mutation preserve the executability of the test sequences. As a result, no repair mechanisms are required and penalties are needed only in infrequent cases.

The objective function for a particular test goal is defined so that it implies a smooth landscape for even complex predicates in many cases (cf. Section 3.5 on page 76). This allows for an effective evolutionary search if the test goal is control-dependent on complex predicates. Furthermore, the objective function involves a distance metric that effectively deals with runtime exceptions (cf. Section 3.5.5 on page 80). As opposed to previous

search-based approaches that generally fail if randomly generated method arguments cause a runtime exception, the search is guided by this additional metric to explore regions of the search space with valid method arguments. In addition, objective functions are also defined for test goals that belong to non-public methods by introducing an additional penalty and distinguishing appropriately between test sequences that involve a call to the non-public method in question and those that do not (cf. Section 3.5.6 on page 83).

Another achievement is that the suggested approach enables the application of well-studied and well-proven genetic programming algorithms. The representation of test sequences by method call trees enables the use of off-the-shelf genetic programming toolboxes, thus facilitating the implementation of an evolutionary test sequence generator enormously. At the same time, since the approach is not dependent on a particular genetic programming algorithm, it can benefit from further improvements and ideas in the field of evolutionary computation.

Furthermore, another achievement of this thesis is an improvement of the objective functions in the presence of function-assigned flags. Function-assigned flags occur when the Boolean return value of a method appears in a predicate. Two strategies to cope with function-assigned flags were suggested. They are intended to smooth the landscape of the objective function and hence improve the guidance to the evolutionary search.

An empirical investigation in the effectiveness of the approach showed that it can outperform random testing when being allocated the same resources. In comparison with two commercial test sequence generators, the approach has been shown to be competitive and even outperformed the two commercial tools in terms of achieved code coverage. The effectiveness of the strategies in dealing with non-public methods and Boolean predicates could also be demonstrated empirically in a case study.

From a very general test-automation-point-of-view, a significant contribution of this thesis is the suggestion of an approach to generating arbitrary test sequences with high feasibility for a given set of classes. Test sequences are interesting in the context of various testing techniques, not only structure-oriented testing. For instance, random testing or robustness testing of classes also requires the generation of test sequences. The strength of the presented approach is that it allows the simple realization of a test sequence generator by means of an arbitrary off-the-self genetic programming system; the only requirement for the genetic programming system to build upon is that it supports strong typing.

## 5.2 Restrictions and Limitations

In particular circumstances, the evolutionary class testing approach has the following limitations:

1. If the test cluster is large and the number of public methods of the test cluster classes is large, the approach struggles due to the huge size of the function set of the genetic programming algorithm. The approach may become inefficient or even fail in severe cases.



2. The approach encounters difficulties if some classes of the test cluster have one or more of the following characteristics: they are GUI elements or involve GUI user interaction, they make use of concurrency concepts such as threads, they possess predicates which involve exact string matching, such as XML-processing classes, their source code is not available.
3. In the case of predicates involving pointer comparisons and type checks, the approach might be ineffective due to the binary distance functions applied in these cases.
4. If the class under test possesses protected methods which are not used by the class itself, the approach encounters problems finding covering test sequences for these methods.
5. If the states some objects must be in in order for a particular test goal to be attained and the number of functions in the function set is high, the approach may fail to find a test sequence that sets the proper states.
6. The test sequences generated by the suggested approach might include unnecessary method calls.

If the test cluster consists of many classes, the function set derived from the public methods of the test cluster classes can be relatively large. This also happens if few classes possess a high number of public methods. Then the function set might consist of thousands of functions. With an increasing size of the function set – and hence an increasing size of the search space – the probability that the “right” methods appear in a candidate test sequence decreases. At the same time, the efficiency of the evolutionary search decreases as well. For example, a large function set occurs if the class under test uses the `String` class. Then, the static analysis also adds the `Locale` class to the test cluster, causing all its methods to be added to the function set. However, in most cases, these methods are not relevant for the test of a class that makes use of strings.

If the class under test belongs to the graphical user interface of an application, the search for reasonable test sequences is relatively hard. This is due to the concepts of event handling and notifications that build the basis for the GUI elements. The evolutionary search would need to evolve reasonable events, which is hard due to the unrestrictedness of the search: no semantical dependencies are regarded during test sequence evolution, for instance that a particular method can only be called if another method has been called in advance. This issue relates to the class’ *modality* (Binder, 1999); the approach does not account for those modalities.

Threads and other related concepts can cause problems for the approach. For instance, if an object created by a candidate test sequence is put into a state where it blocks the execution of the thread that currently executes the test sequence, the overall search might be blocked as well, and hence fail.

Strings can cause problems for the approach for several reasons: either, exactly matching string parameters must be evolved in order to pass equality-checking conditions;

or, string parameters identifying valid resources must be evolved, for instance if a method interprets the string as a file name. In the case of string matching, the approach works but, depending on the length of the string to match, it might become inefficient, meaning that the search might take very long. In the worst case, a termination criterion applies before the string to match has been evolved; then the approach fails. In the case of evolving valid resource identifiers the approach tends to fail generally.

The use of library classes can lower the effectiveness of the approach. For instance, if a method of the class under test uses the `contains` method of the library class `Vector`, the distance function is purely binary. This causes plateaus in the objective function landscape, hampering the evolutionary search. The smoothing strategies, which were also discussed in this thesis, cannot be applied in these cases since the source code of the involved library classes is usually not accessible. The code transformation cannot be carried out. This generally applies for classes for which the binary code is available only.

The distance functions for predicates involving pointer address comparisons and type checks are binary functions, causing plateaus in the objective function landscape. The smoothing strategies do not help in these cases; the plateaus cannot be transformed away by them.

Protected methods are often used as a means for allowing a subclass to overwrite some of the behavior of the base class. However, such a protected method might be called by the base class only. In this case, the static analysis carried out by the evolutionary class testing approach with the intention of identifying the call points of the non-public methods ends up with no result since it analyzes the code of the class under test only. As a result, no test sequences will be generated for the protected method at hand.

Some test goals can only be achieved if some instances participating in the test are in particular states. Therefore, the search space to search for a covering test sequence includes all test sequences that call all conceivable combinations of methods. The search might experience troubles in finding a covering test sequence when the number of methods that can potentially appear in a test sequence is relatively high. This is also due to the lack of sufficient guidance provided by the objective function in this case.

Sometimes, the generated test sequences contain object creations and method calls which are not relevant for the test. This phenomenon is due to the concept of parameter object selectors. Figure 3.8 on page 75 shows an example of it: the middle child of the root node, supposed to provide the argument for the call to `equals`, creates an instance of class `Integer` which is never used because the parameter object value of 7 (right child of the root node) causes object *ir1* to be passed as the parameter object to the method call.

### 5.3 Summary of Future Work

Two main areas of future work can be identified: (1) dealing with the limitations of the evolutionary class testing approach, and (2) investigating new directions that further enhance the technical concept of the approach and fortify it by more theoretical investigations. While Section 5.3.1 on the facing page points out some ideas addressing

some of the limitations, Section 5.3.2 on the next page suggests other directions of future research.

### 5.3.1 Addressing the Limitations

Large function sets that result from large test clusters with classes that possess many methods can be attacked by several approaches: (1) the user can name classes whose methods shall not be transformed to functions of the function set, (2) a heuristic can be applied that prevents the methods from those test cluster classes that are associated to the class under test via several other classes from being transformed to functions of the function set, (3) a static analysis can eliminate all function in the function set that correspond to methods which are neither object-creating nor state-changing, and (4) a combination of these ideas. The first idea tries to exploit the user's knowledge of the class under test. All methods which are not constructors can be disregarded from classes that the user indicates to be not relevant. The second idea tries to automate the elimination of irrelevant methods. This can be accomplished by a heuristic based on the "distance" of a test cluster class to the class under test. This distance can be defined in terms of the number of associations to be traversed from the class under test to the class in question in the class diagram. The partial class diagram in Figure 5.1 shows class

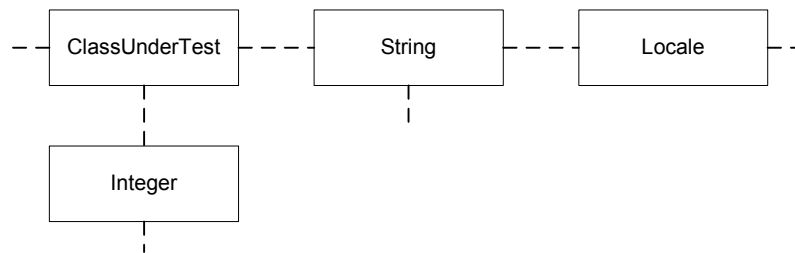


Figure 5.1: Example class diagram

**ClassUnderTest** and some associated classes. The distance to class **Locale**, defined by the number of intermediate associations, is 2. The heuristic could be defined so that it eliminates all methods that require instances of classes that have distance 2 from the class under test. In turn, the classes that are no longer required can be removed from the test cluster. By doing so, the size of the function set can be reduced. The third idea requires a thorough static analysis of the methods of the test cluster classes, aiming at the identification of methods that neither create instances nor affect the state of instances. These methods can possibly be ignored when defining the function set.

The issue of library classes can be dealt with by creating a pre-instrumented version of these classes, including the application of the suggested code transformations. In the case of Java this possibly also requires a modified version of the JVM. Then, the test sequence generator can use the primed runtime environment for test sequence evaluation and receives better guidance to the search.

Pointer-comparing predicates can be addressed by defining distance functions that

rely on an artificially introduced total ordering of the objects created by a test sequence. Listing 5.1 shows a pointer-comparing predicate in line 2 where the address of the object passed to method `equals` of class `IntegerRange` is compared to the `null` reference.

Listing 5.1: Example pointer-comparing predicate

---

```

1 public boolean equals(Object obj) {
2     if( obj == null )
3         // test goal
4     else
5         ...
6 }
```

---

Consider the test sequence shown in Listing 3.2. The parameter object selector (the integer argument) in line 13 defines which instance to be passed from the pool of available instances. The distance function for pointer comparisons can exploit the ordering of the instances within an object pool. For instance, the `null` reference is passed when the parameter object selector has the value 0. Now if the parameter object selector of the candidate test sequence has a value different from 0, the difference between the actual value and 0 indicates how close the test sequence is to pass the `null` reference to the `equals` method, hence satisfying the predicate in line 2 and attaining the test goal.

Protected methods, called by the base class only, can be dealt with by extending the static analysis of the call points to the base classes of the class under test.

McMinn (2004a) elaborates on the state problem for evolutionary structural testing of procedural software. The evolutionary class testing approach can benefit from the findings and ideas of this work. However, the approach of McMinn does not support objects and does not distinguish between object identities. Therefore, the approach must be extended; then, it can provide enormous support in the case of test goals that require objects to be in particular states.

Unnecessary method calls, caused by the concept of the parameter object selectors, can be tackled by analyzing the extended method call trees during linearization. All subtrees that do not create instances which are used by relevant method calls can be ignored and not transformed to the linearized test sequence.

### 5.3.2 Other directions

The following topics and directions for possible future research are conceivable:

- theoretical investigation into evolutionary class testing
- investigation into parameter setting
- search space reduction
- application of grammatical evolution
- usage of formal specifications for test oracle construction

- application in the context of robustness testing
- application in the context of mutation testing

A theoretical investigation can reveal why and when evolutionary class testing is an effective approach. Findings can help predict the effort of evolutionary class testing and indicate when an alternative approach is better. An intensive investigation into the empirical results with particular regard to the test goals for which no test sequence was found can give insight for further improvements.

Parameter setting is an important topic in the area of evolutionary computation. A study of the impact of different settings of the genetic programming algorithms used can provide valuable insight as to the accurate adjustment of the parameters for a given class under test.

Search space reduction aims at accelerating the search since it reduces the number of points to be potentially visited. The VADA approach exists (Harman, Fox, Hierons, Hu, Danicic and Wegener, 2002a) for evolutionary structural testing of procedural software. It identifies those input variables of the function under test which do not affect any computation related to the attainment of the test goal at hand. For instance, if the function under test possesses multiple predicates and if a parameter value is used by the last predicate only, the attainment of the test goals related to all predicates but the last is not dependent on this parameter value. Candidate test inputs need to comprise values for the remaining parameters only; one dimension of the search space can be eliminated. This speeds up the evolutionary search. Currently there is ongoing research on ideas for search space reduction for evolutionary class testing at King's College, London, UK (M. Harman, F. Islam) with particular consideration of aspect-oriented software.

Grammatical evolution (Ryan, Collins and O'Neill, 1998) is a relatively new type of evolutionary algorithm which might be an alternative to the genetic programming algorithm used in this thesis. Its application could enable the formulation of more complex and more compact code constructs used by a test sequence.

Formal specifications such as the design-by-contract annotations in the source code formulated in the Java Modeling Language (Leavens, Baker and Ruby, 1999) can provide the basis to automatically generating the test assertions for a given test sequence. Additionally, they can be used to distinguish between test sequences that violate implicit method preconditions (and are therefore illegal) and those which respect these preconditions. For instance, the approach of Visser et al. (2004) uses this kind of specification for this purpose.

Robustness testing penetrates the unit under test by executing it in a very large number of scenarios. The definition of the test oracle is simple: if the system crashes in one of such scenarios, a fault-revealing test has been found. The ideas of the evolutionary class testing approach can fertilize an approach to automatic robustness testing. JCrasher (Csallner and Smaragdakis, 2004), a robustness checker for Java, randomly explores part of the parameter space of the methods of the class under test in order to cause a system crash. However, JCrasher does not account for state-changing methods, meaning that the parameter objects are usually in initial states. In contrast, method call trees as used

in this thesis also account for state-changing method calls, enabling the creation of a greater variety of robustness tests. These might detect more crash-causing faults in the unit under test.

Mutation testing (King and Offutt, 1991) is a technique to both obtaining tests that have a high potential to reveal common programming mistakes and assessing the quality of a given set of tests. The software under test is slightly modified – *mutated* – and a test is sought which *kills the mutant* (that is, exposes the modification). Such a modification typically affects a single statement, such as a conditional statement. However, a mutant can only be killed if the modified statement is actually executed. Consequently, in the area of object-oriented mutation testing, test sequences are sought which lead to the execution of mutated code. Evolutionary class testing may help in that it facilitates the generation of test sequences that execute the mutated statement, allowing killing the mutant.

# Bibliography

- Alshraideh, M. and Bottaci, L. (2005). Automatic software test data generation for string data using heuristic search with domain specific search operators, *Proceedings of UK Test 2005*, University of Sheffield.
- Baker, J. E. (1985). Adaptive selection methods for Genetic Algorithms, *Proceedings of an International Conference on Genetic Algorithms and their Application*, pp. 101–111.
- Baker, J. E. (1987). Reducing bias and inefficiency in the selection algorithm, *Proceedings of the Second International Conference on Genetic Algorithms and their Application*, pp. 14–21.
- Banzhaf, W., Nordin, P., Keller, R. E. and Francone, F. D. (1998). *Genetic Programming – An Introduction*, Morgan Kaufmann Publishers, San Francisco, CA, USA.
- Baresel, A. and Sthamer, H. (2003). Evolutionary testing of flag conditions, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 2442–2454. July 2003.
- Baresel, A., Binkley, D., Harman, M. and Korel, B. (2004). Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach, *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*.
- Baresel, A., Pohlheim, H. and Sadeghipour, S. (2003). Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 2428–2441.
- Baresel, A., Sthamer, H. and Schmidt, M. (2002). Fitness function design to improve evolutionary structural testing, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1329–1336. 9-13th July.
- Beck, K. (2003). *Test-Driven Development by Example*, Addison-Wesley.
- Beizer, B. (1990). *Software Testing Techniques*, International Thomson Computer Press.
- Beydeda, S. and Gruhn, V. (2003). Bintest – binary search-based test case generation, *Computer Software and Applications Conference*, IEEE Computer Society Press.
- Binder, R. (1999). *Testing Object-Oriented Systems – Models, Patterns, and Tools*, Addison Wesley.

- Boehm, B. (1988). A spiral model for software development and enhancements, **21**(5): 61–72.
- Böhm, W. and Geyer-Schulz, A. (1996). Exact uniform initialization for genetic programming, in R. K. Belew and M. Vose (eds), *Foundations of Genetic Algorithms IV*, Morgan Kaufmann, University of San Diego, CA, USA, pp. 379–407.
- Boshernitsan, M., Doong, R. and Savoia, A. (2006). From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing, *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, ACM Press, New York, NY, USA, pp. 169–180.
- Bottaci, L. (2002). Instrumenting programs with flag variables for test data search by genetic algorithms, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*.
- Buy, U., Orso, A. and Pezze, M. (2000). Automated testing of classes, *International Symposium on Software Testing and Analysis (ISSTA)*.
- Cenqua Pty. Ltd. (2007). Clover, <http://www.cenqua.com/clover/doc>.
- Chellapilla, K. (1998). A preliminary investigation into evolving modular programs without subtree crossover, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pp. 23–31.
- Csallner, C. and Smaragdakis, Y. (2004). JCrasher: An automatic robustness tester for Java, *Software—Practice & Experience* **34**(11): 1025–1050.
- Darwin, C. (1859). *The origin of Species by means of natural selection*, Murray, London.
- Eiben, A. E. and Smith, J. E. (2003). *Introduction to evolutionary computing*, Springer.
- Ferguson, R. and Korel, B. (1996). The chaining approach for software test data generation, *ACM Transactions on Software Engineering and Methodology* **5**(1): 63–86.
- Fogel, L. J., Owens, A. J. and Walsh, M. J. (1965). Artificial intelligence through Simulated Evolution, *Biophysics and Cybernetic Systems*, Spartan, pp. 131–156.
- Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman.
- Goldberg, D. E. (1989). *Genetic Algorithms in search, optimization, and Machine Learning*, Addison-Wesley.
- Goldberg, D. E. and Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms, *Foundations of Genetic Algorithms*, Morgan Kaufmann Publishers, pp. 69–93.



- Gosling, J., Joy, B. and Steele, G. L. (2005). *The Java Language Specification*, Addison Wesley Longman.
- Grochtmann, M. (2000). Einführung in den systematischen test – presentation at euroforum 2000.
- Harman, M. and McMin, P. (2007). A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation, *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 73–83.
- Harman, M., Fox, C., Hierons, R., Hu, L., Danicic, S. and Wegener, J. (2002a). VADA: A transformation-based system for variable dependence analysis, *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pp. 55–64.
- Harman, M., Hu, L., Hierons, R., Baresel, A. and Sthamer, H. (2002b). Improving evolutionary testing by flag removal, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*.
- Haynes, T. D., Schoenefeld, D. A. and Wainwright, R. L. (1996). Type inheritance in strongly typed genetic programming, *Advances in Genetic Programming 2*, MIT Press, Cambridge, MA, USA, pp. 359–376.
- Hecht, M. S. (1977). *Flow Analysis of Computer Programs*, Elsevier Science Inc.
- Holland, J. (1975). *Adaptation in Natural and Artificial Systems*, University of Michigan Press.
- Instantiations, Inc. (2007). CodePro, <http://www.instantiations.com>.
- ISO (2005). ISO/WD 26262.
- ISO/IEC 9899 (1990). C language specification.
- Jones, B. F., Sthamer, H. and Eyres, D. E. (1996). Automatic test data generation using genetic algorithms, *Software Engineering Journal* **11**(5): 299–306.
- Khurshid, S., Păsăreanu, C. and Visser, W. (2003). Generalized symbolic execution for model checking and testing, *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.
- Kim, S., Clark, J. A. and McDermid, J. A. (1999). Assessing test set adequacy for object-oriented programs using class mutation, *Proceedings of Symposium on Software Technology (SoST)*, pp. 72–83.
- Kim, S., Clark, J. A. and McDermid, J. A. (2000). Investigating the applicability of traditional test adequacy criteria for object-oriented programs, *Proceedings of the ObjectDays 2000*.

- Kim, S., Clark, J. A. and McDermid, J. A. (2001). Investigating the effectiveness of object-oriented testing strategies using the mutation method, *Software Testing, Verification & Reliability* **11**(3): 207–225.
- King, J. C. (1976). Symbolic execution and program testing, *Communications of the ACM*.
- King, K. N. and Offutt, J. (1991). A Fortran language system for mutation-based software testing, *Software Practice and Experience* **21**(7): 686–718.
- Kirkpatrick, S., Gellat, C. D. and Vecchi, M. P. (1983). Optimization by simulated annealing, *Science* **220**(4598): 671–680.
- Korel, B. (1990). Automated software test data generation, *IEEE Transactions on Software Engineering* **16**(8): 870–879.
- Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, The MIT Press.
- Leavens, G. T., Baker, A. L. and Ruby, C. (1999). JML: A notation for detailed design, *Behavioral Specifications of Businesses and Systems* pp. 175–188.
- Liu, X., Liu, H., Wang, B., Chen, P. and Cai, X. (2005a). A unified fitness function calculation rule for flag conditions to improve evolutionary testing, *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, ACM Press.
- Liu, X., Wang, B. and Liu, H. (2005b). Evolutionary search in the context of object-oriented programs, *MIC2005: The Sixth Metaheuristics International Conference*.
- Luke, S. (2007). ECJ, <http://cs.gmu.edu/~eclab/projects/ecj/>.
- McCabe, T. J. (1976). A complexity measure, *IEEE Transactions on Software Engineering* **2**(12): 101–111.
- McMinn, P. (2004a). *Evolutionary Search for Test Data in the Presence of State Behaviour*, PhD thesis, University of Sheffield, UK.
- McMinn, P. (2004b). Search-based test data generation: A survey, *Journal on Software Testing, Verification and Reliability* **14**(2): 105–156.
- McMinn, P. and Holcombe, M. (2006). Evolutionary testing using an extended chaining approach, *Evolutionary Computation* **14**(1): 41–64.
- Michael, C., McGraw, G. and Schatz, M. (2001). Generating software test data by evolution, *IEEE Transactions on Software Engineering* **27**(12): 1085–1110.
- Miller, W. and Spooner, D. L. (1976). Automatic generation of floating-point test data, *IEEE Transactions on Software Engineering* **2**(3): 223–226.

- Mock Objects (2006). <http://www.mockobjects.com>.
- Montana, D. J. (1995). Strongly typed genetic programming, *Evolutionary Computation* **3**(2): 199–230.
- Muehlenbein, H. and Schliekamp-Voosen, D. (1993). Predictive models for the Breeder Genetic Algorithm: I. Continuous Parameter Optimization, *Evolutionary Computation* **1**(1): 25–49.
- Oster, N. (2007). *Automatische Generierung optimaler struktureller Testdaten für objekt-orientierte Software mittels multi-objektiver Metaheuristiken*, Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen.
- Parasoft, Inc. (n.d.). Jtest, <http://www.parasoft.com>.
- Pargas, R. P., Harrold, M. J. and Peck, R. R. (1999). Test-data generation using genetic algorithms, *Journal of Software Testing, Verification and Reliability* **9**(4): 263–282.
- Pohlheim, H. (1995). Ein genetischer algorithmus mit merhfachpopulationen zur numerischen optimierung, *at-Automatisierungstechnik* **3**: 127–135.
- Pohlheim, H. (1999). *Evolutionäre Algorithmen – Verfahren, Operatoren und Hinweise für die Praxis*, Springer-Verlag, Berlin.
- Rechenberg, I. (1971). *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*, PhD thesis, Technical University of Berlin, Germany.
- RTCA Inc. (1992). RTCA/DO-178b.
- Russell, S. and Norvig, P. (1995). *Artificial intelligence: a modern approach*, Prentice-Hall.
- Ryan, C., Collins, J. J. and O'Neill, M. (1998). Grammatical evolution: Evolving programs for an arbitrary language, *EuroGP 1998*, pp. 83–96.
- Sen, K. and Agha, G. (2006). Cute and jcute : Concolic unit testing and explicit path model-checking tools, *18th International Conference on Computer Aided Verification (CAV'06)*, Lecture Notes in Computer Science, Springer. (To Appear. Tool Paper).
- Sen, K., Marinov, D. and Agha, G. (2005). CUTE: A concolic unit testing engine for C, *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pp. 263–272.
- Staknis, M. E. (1990). Software quality assurance through prototyping and automated testing, *Inf. Softw. Technol.* **32**(1): 26–33.

- Sthamer, H. (1996). *The Automatic Generation of Software Test Data Using Genetic Algorithms*, PhD thesis, University of Glamorgan.
- Stroustrup, B. (1988). What is object-oriented programming?, *IEEE Software*.
- Stroustrup, B. (2000). *The C++ programming language*, Addison Wesley Longman.
- Tatsubori, M., Chiba, S., Killijian, M. and Itano, K. (2000). OpenJava: A class-based macro system for Java, *Lecture Notes in Computer Science 1826, Reflection and Software Engineering* pp. 117–133.
- Tillmann, N. and Schulte, W. (2006). Unit tests reloaded: Parameterized unit testing with symbolic execution, *Technical report*, Microsoft Research. MSR-TR-2005-153.
- Tonella, P. (2004). Evolutionary testing of classes, *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ACM Press, New York, NY, USA, pp. 119–128.
- Tracey, N., Clark, J. and Mander, K. (1998a). Automated program flaw finding using simulated annealing, *SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ACM Press.
- Tracey, N., Clark, J., Mander, K. and McDermid, J. (1998b). An automated framework for structural test-data generation, *Proceedings of the International Conference on Automated Software Engineering (ASE)*, IEEE, pp. 285–288.
- Tsang, E. (1993). *Foundations of Constraint Satisfaction*, Academic Press.
- Visser, W., Păsăreanu, C. S. and Khurshid, S. (2004). Test input generation with Java PathFinder, *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ACM Press, New York, NY, USA, pp. 97–107.
- Wappler, S. (2004). *Using evolutionary algorithms for the test of object-oriented systems*, Master's thesis, University of Potsdam, Germany.
- Wegener, J., Baresel, A. and Sthamer, H. (2001). Evolutionary test environment for automatic structural testing, *Information and Software Technology* **43**(1): 841–854.
- Wilson, G. C., McIntyre, A. and Heywood, M. I. (2004). Resource review: Three open source systems for evolving programs: Lilgp, ecj and grammatical evolution, *Genetic Programming and Evolvable Machines* **5**(1): 103–105.
- Xanthakis, S. E., Skourlas, C. C. and LeGall, A. (1992). Application of genetic algorithms to software testing, *Proceedings of the 5th International Conference on Software Engineering and its Applications*, pp. 625–636.
- Xie, T., Marinov, D., Schulte, W. and Notkin, D. (2005). Symstra: A framework for generating object-oriented unit tests using symbolic execution, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pp. 365–381.

# A Source Codes and Algorithms

## A.1 Source Listings

Listing A.1: Class Integer

---

```
1 class Integer
2 {
3     private final int value;
4
5     public Integer()
6     {
7         this( 0 );
8     }
9
10    public Integer(int value)
11    {
12        this.value = value;
13    }
14
15    public int intValue()
16    {
17        return value;
18    }
19
20    public Integer negate()
21    {
22        return new Integer(-value);
23    }
24 }
```

---

Listing A.2: Class IntegerRange

---

```
1 class IntegerRange
2 {
3     private final Integer lower , upper;
4
5     static IntegerRange combine(IntegerRange ,
6                                 IntegerRange)
7                                 throws Exception
8
9     public IntegerRange(Integer lower , Integer upper)
10    {
```

```

11     if( lower > upper )
12         handleInvalidRange(lower , upper);
13     this.lower = lower;
14     this.upper = upper;
15 }
16
17 public boolean equals(Object obj)
18 {
19     if( obj == this ) return true;
20     if( !(obj instanceof IntegerRange) ) return false;
21
22     IntegerRange that = (IntegerRange)obj;
23     if( this.lower.intValue() != that.lower.intValue() )
24         return false;
25     if( this.upper.intValue() != that.upper.intValue() )
26         return false;
27
28     return true;
29 }
30
31 public void growUp() throws Exception
32 {
33     if( upper.intValue() == Integer.MAX_VALUE )
34         handleOverflow();
35     else
36         upper = new Integer(upper.intValue()+1);
37 }
38
39 public void growLow()
40 {
41     if( lower.intValue() == Integer.MIN_VALUE )
42         handleUnderflow();
43     else
44         lower = new Integer(lower.intValue()-1);
45 }
46
47 public Integer size()
48 {
49     if( emptyRange )
50         return new Integer();
51     else
52         return new Integer(
53             upper.intValue()-lower.intValue());
54 }
55
56 public boolean emptyRange()
57 {
58     if( lower.intValue() == upper.intValue() )
59         return true;

```

```
60     else
61         return false;
62     }
63
64     private void handleOverflow()
65     {
66         throw new IllegalStateException(
67             "Range already at max bound.");
68     }
69
70     private void handleUnderflow()
71     {
72         throw new IllegalStateException(
73             "Range already at min bound.");
74     }
75
76     private void handleInvalidRange(Integer lower,
77                                     Integer upper)
78     {
79         throw new IllegalArgumentException(
80             "Specified 'lower' bound must be lower"+
81             " than 'upper' bound.");
82     }
83 }
```

---

Listing A.3: Class State1

---

```

1 public class State1
2 {
3     private int state;
4
5     public State1()
6     {
7         state = 0;
8     }
9
10    // dummy methods to increase search space size
11    public void m1() {}
12    public void m2() {}
13    public void m3() {}
14    public void m4() {}
15    public void m5() {}
16
17    public void next()
18    {
19        state = (state+1) % 10;
20    }
21
22    public boolean test()
23    {
24        if( state == 9 )
25            return true;
26        else
27            return false;
28    }
29 }

```

---

Listing A.4: Class Stack

---

```

1 public class Stack {
2     private static int MAX_ELEMENTS = 10;
3     private Object[] elements;
4     private int freeIndex;
5
6     public Stack() {
7         elements = new Object [MAX_ELEMENTS];
8         freeIndex = 0;
9     }
10
11    public void add(Object element) {
12        if( isFull() ) throw new Exception();
13        elements[freeIndex++] = element;
14    }
15
16    public Object removeTop() {

```



```

17     if( isEmpty() ) throw new Exception();
18     return elements[--freeIndex];
19 }
20
21 public boolean isFull() {
22     if( freeIndex >= MAX_ELEMENTS )
23         return true;
24     else return false;
25 }
26
27 public boolean isEmpty() {
28     if( freeIndex == 0 )
29         return true;
30     else return false;
31 }
32
33 public Object top() throws Exception {
34     if( isEmpty() )
35         throw new Exception( "Stack is empty." );
36     else
37         return elements[freeIndex - 1];
38 }
39
40 public Object elementAt(int index)
41 {
42     return elements[index];
43 }
44
45 public static void testCase1() throws Exception
46 {
47     Stack s = new Stack();
48     try
49     {
50         Object o = s.top();
51     }
52     catch( Exception e )
53     {
54         throw e;
55     }
56 }

```

---

Listing A.5: Class StackT

---

```

1 public class StackT {
2     private static int MAX_ELEMENTS = 10;
3     private Object[] elements;
4     private int freeIndex;
5
6     public void add(Object element) {

```

```

7      if( isFull() > 0 ) throw new Exception();
8      elements[freeIndex++] = element;
9  }
10 public Object removeTop() {
11     if( isEmpty() > 0 ) throw new Exception();
12     return elements[--freeIndex];
13 }
14 public double isFull() {
15     if( freeIndex >= MAX_ELEMENTS )
16         return T.dist( T.TRUE,
17             <freeIndex>=MAX_ELEMENTS>, 1 );
18     else return T.dist( T.FALSE,
19         <freeIndex>=MAX_ELEMENTS>, 1 );
20 }
21 public double isEmpty() {
22     if( freeIndex == 0 )
23         return T.dist( T.TRUE,
24             <freeIndex==0>, 1 );
25     else return T.dist( T.FALSE,
26         <freeIndex==0>, 1 );
27 }
28 }

```

---

## A.2 Algorithms

Listing A.6: Test-sequence-generating algorithm TCGen1

---

```

1  begin algorithm TCGen1
2      in: class to be tested c
3      out: set of test cases T
4
5      identify test cluster C for c
6      instrument source codes of C
7      collect test goals G from c
8      generate function set for C
9      generate type set for C
10
11     for each test goal g in G
12         modify function set for g
13         create initial tree individuals I
14         evaluate tree individuals I:
15         for each tree individual i in I
16             specify parameter space genotype for individual i
17             perform parameter search for i:
18                 create initial vector individuals J
19                 evaluate the vector individuals J:
20                 for each vector individual j in J

```

```

21         create a test program from the tree
22         individual i and the vector individual j
23         execute the test program, thereby
24         monitor execution flow
25         calculate fitness based on distance
26     end for
27     while termination criterion not met:
28         recombine and mutate individuals
29         evaluate offspring
30     end while
31     return fitness of best j as fitness of i
32 end for
33 while termination criterion not met:
34     recombine tree individuals
35     mutate tree individuals
36     evaluate tree individuals
37 end while
38 insert test sequence into T if g covered
39 end for
40 end algorithm

```

---

### A.2.1 TCGen2

Listing A.7: Test-sequence-generating algorithm TCGen2

---

```

1  begin algorithm TCGen2
2    in: class to be tested c
3    out: set of test cases T
4
5    identify test cluster C for c
6    instrument source codes of C
7    collect test goals G from c
8    generate function set for C
9    generate type set for C
10
11   for each test goal g in G
12     modify function set for g
13     create initial tree individuals I
14     evaluate tree individuals I:
15     for each tree individual i in I
16       create a test program from the tree
17       individual i and the vector individual j
18       execute the test program, thereby
19       monitor execution flow
20       calculate fitness based on distance
21   end for
22   while termination criterion not met:
23     recombine tree individuals

```

```
24         mutate tree individuals
25         evaluate tree individuals
26     end while
27     insert test sequence into T if g covered
28 end for
29 end algorithm
```

---