Information and Software Technology 51 (2009) 1534–1548

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Test Case Evaluation and Input Domain Reduction strategies for the Evolutionary Testing of Object-Oriented software

José Carlos Bregieiro Ribeiro [a,*], Mário Alberto Zenha-Rela [b], Francisco Fernández de Vega [c]

[a] Polytechnic Institute of Leiria, Morro do Lena, Alto do Vieiro, Leiria, Portugal
[b] University of Coimbra, CISUC, DEI, 3030-290 Coimbra, Portugal
[c] University of Extremadura, C/Sta Teresa de Jornet, 38 Mérida, Spain

ARTICLE INFO

Article history:
Available online 5 July 2009

Keywords:
Evolutionary Testing
Search-Based Software Engineering
Test Case Evaluation
Input Domain Reduction

ABSTRACT

In Evolutionary Testing, meta-heuristic search techniques are used for generating test data. The focus of our research is on employing evolutionary algorithms for the structural unit-testing of Object-Oriented programs. Relevant contributions include the introduction of novel methodologies for automation, search guidance and Input Domain Reduction; the strategies proposed were empirically evaluated with encouraging results.

Test cases are evolved using the Strongly-Typed Genetic Programming technique. Test data quality evaluation includes instrumenting the test object, executing it with the generated test cases, and tracing the structures traversed in order to derive coverage metrics. The methodology for efficiently guiding the search process towards achieving full structural coverage involves favouring test cases that exercise problematic structures. Purity Analysis is employed as a systematic strategy for reducing the search space.

1. Introduction

Software testing is expensive, typically consuming roughly half of the total costs involved in software development while adding nothing to the raw functionality of the final product. Yet, it remains the primary method through which confidence in software is achieved [6]. A large amount of the resources spent on testing are applied on the difficult and time consuming task of locating quality test data; automating this process is vital to advance the state-of-the-art in software testing. However, automation in this area has been quite limited, mainly because the exhaustive enumeration of a program's input is unfeasible for any reasonably-sized program, and random methods are unlikely to exercise "deeper" features of software [25].

Meta-heuristic search techniques, like Evolutionary Algorithms – high-level frameworks which utilise heuristics, inspired by genetics and natural selection, in order to find solutions to combinatorial problems at a reasonable computational cost [4] – are natural candidates to address this problem, since the input space is typically large but well defined, and test goal can usually be expressed as a fitness function [10].

The application of Evolutionary Algorithms to test data generation is often referred to as *Evolutionary Testing* [39] or *Search-Based Testing* [25]. Approaches have been proposed that focus on the usage of Genetic Algorithms [16,17,39,48], Ant Colony Optimization [22], Genetic Programming [38], Strongly-Typed Genetic Programming [43,45], and Memetic Algorithms [1].

Evolutionary Testing is an emerging methodology for automatically generating high quality test data. It is, however, a difficult subject, especially if the aim is to implement an automated solution, viable with a reasonable amount of computational effort, which is adaptable to a wide range of test objects. Significant success has been achieved by applying this technique to the automatic generation of unit-test cases for procedural software [24,25]. The application of search-based strategies for Object-Oriented unit-testing is, however, fairly recent [39] and is yet to be investigated comprehensively [11].

The focus of our research is precisely on developing a solution for employing Evolutionary Algorithms for generating test sets for the structural unit-testing of Object-Oriented programs. Our approach involves representing and evolving test cases using the Strongly-Typed Genetic Programming technique [28]. The methodology for evaluating the quality of test cases includes instrumenting the program under test, and executing it using the generated test cases as inputs with the intention of collecting trace information with which to derive coverage metrics. The aim is that of efficiently guiding the search process towards achieving full structural

* Corresponding author. Tel.: +351 965522037.
E-mail addresses: jose.ribeiro@estg.ipleiria.pt, jcbribeiro@gmail.com (J.C.B Ribeiro), mzrela@dei.uc.pt (M.A. Zenha-Rela), fcofdez@unex.es (F. Fernández de Vega).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.06.009

coverage of the program under test. These concepts have been implemented into the *eCrash* automated test case generation tool – which will be described below.

Our main goals are those of defining strategies for addressing the challenges posed by the Object-Oriented paradigm and of proposing methodologies for enhancing the efficiency of search-based testing approaches. The primary contributions of this work are the following:

- Presenting a strategy for Test Case Evaluation and search guidance, which involves allowing unfeasible test cases (i.e., those that terminate prematurely due to a runtime exception) to be considered at certain stages of the evolutionary search – namely, once the feasible test cases that are being bred cease to be interesting.
- Introducing a novel Input Domain Reduction methodology, based on the concept of Purity Analysis, which allows the identification and removal of entries that are irrelevant to the search problem because they do not contribute to the definition of test scenarios.

Additionally, our methodology for automated test case generation is thoroughly described and validated through a series of empirical studies performed on standard Java classes.

This article is organized as follows. In the next Section, we start by introducing the concepts underlying our research. Next, related work is reviewed and contextualized. In Section 4, our test case generation methodology and the *eCrash* tool are described. The experiments conducted in order to validate and observe the impact of our proposals are discussed in Section 5, with special emphasis being put on studying the novel Test Case Evaluation and Input Domain Reduction strategies. The concluding Section presents some final considerations, the most relevant contributions, and topics for future work.

2. Background and terminology

In Evolutionary Testing, meta-heuristic search techniques are employed to select or generate test data; this section presents the most important Software Testing and Evolutionary Algorithms aspects related with this interdisciplinary area. Special attention is paid to the concepts of particular interest to our technical approach.

2.1. Software testing

Software testing is the process of exercising an application to detect errors and to verify that it satisfies the specified requirements [21]. When performing *unit-testing*, the goal is to warrant the robustness of the smallest units – the *test objects* – by testing them in an isolated environment. Unit-testing is performed by executing the test objects in different scenarios using relevant and interesting *test cases*. A *test set* is said to be adequate with respect to a given criterion if the entirety of test cases in this set satisfies this criterion.

Distinct levels of testing include functional (black-box) and structural (white-box) testing [6]. Traditional *structural adequacy criteria* include branch, data-flow and statement coverage; the basic idea is to ensure that all the control elements in a program are executed by a given test set, providing evidence of its quality. The metrics for measuring the thoroughness of a test set can be extracted from the structure of the target object's source code, or even from compiled code (e.g., Java bytecode).

The evaluation of the quality of a given test set and the guidance to the test case selection using structural criteria generally requires the definition of an underlying model for program representation – usually a *Control-Flow Graph* (e.g., Fig. 4). The Control-Flow Graph is an abstract representation of a given method in a class; control-flow testing criteria can be derived based on such a program representation to provide a theoretical and systematic mechanism to assess the quality of the test set [29]. Two well known *control-flow testing standards* to derive testing requirements from the Control-Flow Graph are the all-nodes and all-edges criteria [42].

The observations needed to assemble the metrics required for the evaluation of test data suitability can be collected by abstracting and modelling the behaviours programs exhibit during execution, either by static or dynamic analysis techniques [40]. Static analysis involves the construction and analysis of an abstract mathematical model of the system (e.g., symbolic execution); testing is performed without executing the method being tested, but rather this abstract model. This type of analysis is complex, and often incomplete due to the simplifications in the model. In contrast, *dynamic analysis* involves executing the actual test object and monitoring its behaviour; while it may not be possible to draw general conclusions from dynamic analysis, it provides evidence of the successful operation of the software.

Dynamic monitoring of structural entities can be achieved by *instrumenting* the test object, and *tracing* the execution of the structural entities traversed during test case execution. Instrumentation is performed by inserting probes in the test object.

2.1.1. Object-Oriented Software Testing

Most work in testing has been done with "procedure-oriented" software in mind; nevertheless, traditional methods – despite their efficiency – cannot be applied without adaptation to Object-Oriented systems.

For Object-Oriented programs, classes and objects are typically considered to be the smallest units that can be tested in isolation. An *object* stores its state in fields and exposes its behaviour through methods. Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation* – a fundamental principle of Object-Oriented programming [5].

A unit-test case for Object-Oriented software consists of a *Method Call Sequence*, which defines the test scenario. During test case execution, all participating objects are created and put into particular states through a series of method calls. Each test case focuses on the execution of one particular public method – the *Method Under Test*. It is not possible to test the operations of a class in isolation; testing a single class involves other classes, i.e., classes that appear as parameter types in the method signatures of the *Class Under Test*. The transitive set of classes which are relevant for testing a particular class is called the *test cluster* [43].

In summary, the process of performing unit-testing on Object-Oriented programs usually requires [44]:

- at least, an instance of the Class Under Test;
- additional objects, which are required (as parameters) for the instantiation of the Class Under Test and for the invocation of the Method Under Test – and for the creation of these additional objects, more objects may be required;
- putting the participating objects into particular states, in order for the test scenario to be processed in the desired way – and, consequently, method calls must be issued for these objects.

2.2. Evolutionary Algorithms

Evolutionary Algorithms use simulated evolution as a search strategy to evolve candidate solutions for a given problem, using operators inspired by genetics and natural selection. The best

known algorithms in this class include Evolution Strategies, Evolutionary Programming, Genetic Algorithms and Genetic Programming. These methodologies are usually employed to solve problems for which no reasonable fast algorithms have been developed, and they are especially fit for optimization problems [8].

Independently of its class, any evolutionary program should possess the following attributes [27]:

- a *genetic representation* for potential solutions to the problem;
- a way to create an *initial population* of potential solutions;
- an *evaluation function* that plays the role of the environment, rating solutions in terms of their "fitness";
- *genetic operators* that alter the composition of children;
- *values for various parameters* that the Genetic Algorithm uses (population, size, probabilities of applying genetic operators, etc.).

Genetic Algorithms [15] are probably the most well known form of Evolutionary Algorithms. The term "Genetic Algorithm" comes from the analogy between the encoding of candidate solutions as a sequence of simple components and the genetic structure of a chromosome; continuing with this analogy, solutions are often referred to as *individuals* or *chromosomes*. The components of the solution are referred to as *genes*, with the possible values for each component being called *alleles* and their position in the sequence being the *locus*. The encoded structure of the solution for manipulation by the Genetic Algorithm is called the *genotype*, with the decoded structure being known as the *phenotype*.

Genetic algorithms maintain a *population* of candidate solutions rather than just one current solution; in consequence, the search is afforded many starting points, and the chance to sample more of the search space than local searches. The population is iteratively *recombined* and *mutated* to evolve successive populations, known as *generations*. Various selection mechanisms can be used to decide which individuals should be used to create offspring for the next generation; key to this is the concept of the *fitness* of individuals. A fitness function quantifies the optimality of a solution; the idea of selection is to favour the fitter individuals, in the hope of breeding better offspring.

However, too strong a bias towards the best individuals will result in their dominance of future generations, thus reducing diversity and increasing the chance of premature convergence on one area of the search space. Conversely, too weak a strategy will result in too much exploration, and not enough evolution for the search to make substantial progress.

Traditional Genetic Algorithm *operators* include selection, crossover, and mutation [8,15]:

- *Selection* (or Reproduction) is the process of copying individuals. They are chosen according to their fitness value; selection methodologies include Fitness-Proportionate Selection, Linear Ranking or Tournament Selection.
- *Crossover* is the procedure of mating the members of the new population, in order to create a new set of individuals. As genetic material is being combined, new genotypes will be produced.
- *Mutation* modifies the values of one or several genes of an individual.

Genetic Programming [19] is a specialization of Genetic Algorithms usually associated with the evolution of tree structures; it focuses on automatically creating computer programs by means of evolution, and is thus especially suited for representing and evolving test cases. In most Genetic Programming approaches, the programs are represented using *tree genomes* (e.g., Fig. 8). The leaf nodes are called *terminals*, whereas the non-leaf nodes are called *non-terminals*. Terminals can be inputs to the program, constants or functions with no arguments; non-terminals are functions taking at least one argument. The *Function Set* is the set of functions from which the Genetic Programming system can choose when constructing trees. A set of programs is manipulated by applying reproduction, crossover and mutation until the optimum program is found or other termination criteria are met. Fig. 1 is a flowchart for the Genetic Programming paradigm.

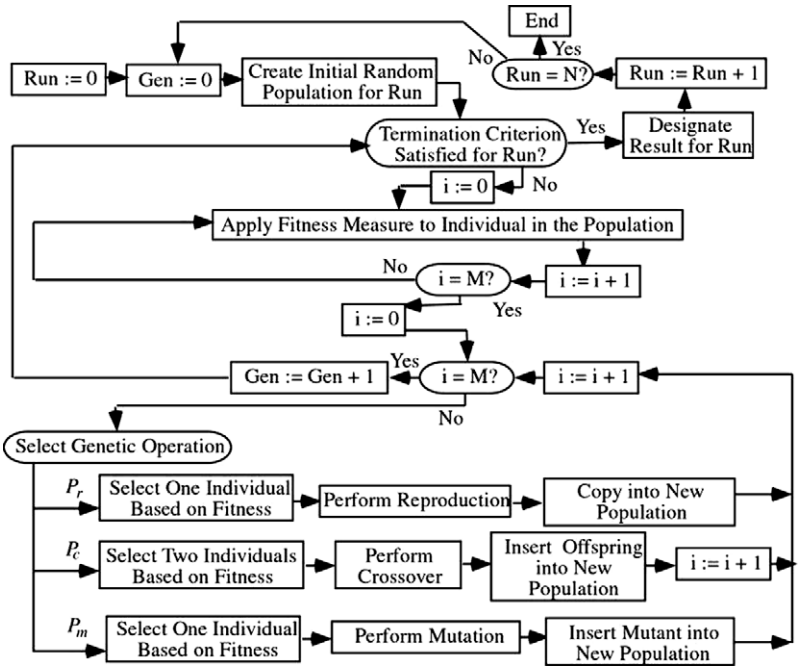


Fig. 1. Flowchart for the Genetic Programming paradigm [19]. The index *i* refers to an individual in the population of size *M*. The variable *Gen* is the number of the current generation. The variable *Run* is the number of the current run, and *N* is the predefined number of runs.

The nodes of a Genetic Programming tree are usually not typed – i.e., all the functions are able to accept every conceivable argument. Non-typed Genetic Programming approaches are, however, unsuitable for representing test programs for Object-Oriented software [13], because any element can be a child node in a parse tree for any other element without having conflicting data types, which can lead to the generation of syntactically illegal trees. In [28], Montana suggested the *Strongly-Typed Genetic Programming* mechanism for defining typed Genetic Programming nodes. Strongly-Typed Genetic Programming allows the definition of types for the variables, constants, arguments and returned values; the only restriction is that the data type for each element must be specified beforehand in the Function Set (e.g., Table 3). This causes the initialization process and the various genetic operations to only construct syntactically correct trees.

The Strongly-Typed Genetic Programming search space is the set of all legal parse trees – i.e., all of the functions have the correct number of parameters of the correct type. Strongly-Typed Genetic Programming is thus particularly suited for representing the Method Call Sequences of Object-Oriented test cases, as it enables the reduction of the search space to the set of *compilable* sequences, by allowing the definition of constraints that eliminate invalid combinations of operations. The usage of Strongly-Typed Genetic Programming in this context was first proposed in [45].

2.3. Evolutionary Testing

The application of Evolutionary Algorithms to test data generation is often referred to in the literature as Evolutionary Testing [25]. The goal of Evolutionary Testing problems is to find a set of test cases that satisfies a certain test criterion – such as full structural coverage of the test object. The test objective must be defined numerically and suitable fitness functions, that provide guidance to the search by telling how good each candidate solution is, must be defined [11].

The search space is the set of possible inputs to the test object [12]. In the particular case of Object-Oriented programs, the input domain encompasses the parameters of the test object's public methods, including the *implicit parameter* (i.e., the `this` parameter) and all the *explicit parameters*. As such, the goal of the evolutionary search is to find Method Call Sequences that define interesting state scenarios for the variables which will be passed, as arguments, in the call to the Method Under Test.

The method call dependences involved in the Method Call Sequences' construction can be represented by means of an *Extended Method Call Dependence Graph (EMCDG)* [45]. This graph is a bipartite, directed graph with two types of nodes: nodes of type 1 represent members (i.e., methods or constructors), and the nodes of type 2 represent data types. A link between a member node and a data type node means that the method can only be called if an instance of the linked data type is created in advance; a link between a data type node and a member node means that an instance of the class is created or delivered by the linked member (e.g., Fig. 9).

One of the most pressing challenges faced by researchers in the Evolutionary Testing area is the *state problem* [26], which occurs with objects that exhibit state-like qualities by storing information in fields that are protected from external manipulation – and that can only be accessed through the public methods that expose the classes' internals and grant the access to the objects' state. The encapsulation principle, in particular, constitutes a serious hindrance to testing [5], because the only way to observe the state of an object is through its operations, and the only way to change the state of an object is through the execution of a series of method calls.

Defining a test set that achieves full structural coverage may, in fact, involve the generation of complex and intricate test cases in order to define elaborate state scenarios, and requires the definition of carefully fine-tuned methodologies that promote the transversal of problematic structures and difficult control-flow paths. Our technical approach (Section 4) includes the presentation of novel techniques for Test Case Evaluation (Section 4.2) and Input Domain Reduction (Section 4.4); the following subsections establish the rationale for pursuing new advances on these topics and present the most relevant concepts. In Section 3, a review of the work developed in the area of Object-Oriented Evolutionary Testing is presented.

2.3.1. Feasible and unfeasible test cases

Syntactically correct and compilable Method Call Sequences may still abort prematurely, if a runtime exception is thrown during execution [43]. Test cases can thus be separated in two classes:

- *feasible* test cases are effectively executed, and terminate with a call to the Method Under Test;
- *unfeasible* test cases terminate prematurely because a runtime exception is thrown by an instruction of the Method Call Sequence.

Fig. 2 depicts an example of a Method Call Sequence; in this program, instructions 1, 3 and 4 instantiate new objects, whereas instructions 2 and 5 aim to change the state of the `stack0` and `object3` instance variables that will be used in the call to the Method Under Test (`push`) at instruction 6.

However, instructions 2 and 5 render the test case unfeasible by throwing runtime exceptions (`EmptyStackException`s, to be precise). When this happens, it is not possible to observe the structural entities traversed in the Method Under Test because the final instruction of the Method Call Sequence is not reached.

2.3.2. Input Domain Reduction

The massive number of distinct Method Call Sequences that can possibly be created while searching for a particular test scenario constitutes a notorious hindrance to the test case generation process. *Input Domain Reduction* deals with the removal of irrelevant variables from a given test data generation problem, thereby reducing the size of the search space [12].

The input domain can effectively be reduced by acknowledging that some methods in Object-Oriented languages have no externally visible side effects when executed or, at least the extent of these side effects is limited in some way; these are called *pure methods* [37].

According to the definition provided by the Java Modelling Language (JML), a pure method is one which does not: perform Input/Output operations; write to any pre-existing objects; or invoke any impure methods [20]. This definition allows a method to change the state of newly allocated objects and/or construct objects and return them as a result.

JML is annotation based, requiring purity information to be provided manually by users. Salcianu and Rinard have, however, presented a systematic Purity Analysis methodology [37] based on a previous points-to and escape analysis [46]. Their purity definition is similar to the one specified by JML: a pure method can read from

```
1: Stack stack0 = new Stack();
2: Object object1 = stack0.peek();
3: Stack stack2 = new Stack();
4: Object object3 = new Object();
5: int int4 = stack2.search(object3);
6: Object object5 = stack0.push(object3);
```

Fig. 2. Example Method Call Sequence.

or write to local objects, and can also create, modify and return new objects not present in the input state.

More interestingly, Purity Analysis is able to identify important purity properties even when a method is not pure, such as safe and read-only parameters:

- a parameter is *read-only* if the method does not write the parameter or any objects reachable from the parameter;
- a parameter is *safe* if it is read-only, and the method does not create any new externally visible paths in the heap to objects reachable from the parameter.

Parameter Purity Analysis is especially useful in the context of search-based test case generation, as it provides a means to automatically identify and remove entries that are irrelevant to the search problem, reducing the size of the set of method calls from which the algorithm can choose when constructing the Method Call Sequences that compose test cases.

In the example Method Call Sequence depicted in Fig. 2, instructions 2 and 5 do not actually change the state of the `stack0` and `object3` reference variables like they were supposed to: the `peek` method simply looks at the object at the top of the stack without removing it, and the `search` method returns the 1-based position where an object is on the stack without changing the state of the object or the stack.

In fact, performing Parameter Purity Analysis on the implicit parameters of the `peek` and `search` methods and on the explicit parameter of the `search` method would allow marking them as being *safe*. Therefore, these particular methods could safely be discarded of being `Stack` and `Object` data type providers, and instructions 2 and 5 could be excluded from the set of instructions selectable by the test case generation algorithm.

Incidentally, the exclusion of these instructions from this particular Method Call Sequence would transform the corresponding test case into a feasible one; Parameter Purity Analysis may thus indirectly enhance the search process by preventing the creation of test cases that are rendered unfeasible by the inclusion of instructions that do not contribute to the definition of state scenarios.

3. Related work

Evolutionary Algorithms have already been applied with significant success to the search for test data; Xanthakis et al. [47] presented the first application of heuristic optimization techniques for test data generation in 1992 [25].

However, research has been mainly geared towards generating test data for procedural software. The first approach to the field of Object-Oriented Evolutionary Testing, based on the concept of Genetic Algorithms, was presented by Tonella [39] in 2004. In this work, the *eToc* tool for the Evolutionary Testing of Object-Oriented software was described. The approach presented involved generating input sequences for the white-box testing of classes by means of Genetic Algorithms, with possible solutions being represented as chromosomes. A source-code representation was used, and an original evolutionary algorithm, with special evolutionary operators for recombination and mutation on a statement level – i.e., mutation operators inserted or removed methods from a test program – was defined. A population of individuals, representing the test cases, was evolved in order to increase a measure of fitness, accounting for the ability of the test cases to satisfy a coverage criterion of choice. New test cases were generated as long as there were targets to be covered or a maximum execution time was reached. However, the encapsulation problem was not addressed, and this proposal only dealt with a simple state problem.

An approach which employed an Ant Colony Optimization algorithm was presented in [22]. The focus was on the generation of the shortest Method Call Sequence for a given test goal, under the constraint of state dependent behaviour and without violating encapsulation. Ant PathFinder, hybridizing Ant Colony Optimization and Multiagent Genetic Algorithms were employed. To cover branches enclosed in private/protected methods without violating encapsulation, call chain analysis on class call graphs was introduced.

In [44] the focus was put on the usage of Universal Evolutionary Algorithms – i.e., evolutionary algorithms, provided by popular toolboxes, which are independent from the application domain and offer a variety of predefined, probabilistically well-proven evolutionary operators. An encoding was proposed that represented Object-Oriented test cases as basic type value structures, allowing for the application of various search-based optimization techniques – such as Hill Climbing or Simulated Annealing. The test cases generated could be transformed into test classes according to popular testing frameworks (e.g., JUnit). Still, the suggested encoding did not prevent the generation of individuals which could not be decoded into test programs without errors; the fitness function used different penalty mechanisms in order to penalize invalid sequences and guide the search towards regions that contained valid sequences. Due to the generation of invalid sequences, the approach lacked efficiency for more complicated cases.

A methodology for creating test software for Object-Oriented systems using a Genetic Programming approach was proposed in [38]; this methodology was advantageous over the more established search-based test-case generation approaches because the test software is represented and altered as a fully functional computer program. However, it was pointed out that the number of different operation types is quite limited, and that large classes which contain many methods lead to huge hierarchical trees.

A Strongly-Typed Genetic Programming based approach was presented in [45]. Potential solutions were encoded using the Strongly-Typed Genetic Programming technique, with Method Call Sequences being represented by method call trees; these trees are able to express the call dependences of the methods that are relevant for a given test object. To account for polymorphic relationships which exist due to inheritance relations, the Strongly-Typed Genetic Programming types used by the Function Set were specified in correspondence to the type hierarchy of the test cluster classes. The emphasis of this work was on sequence validity; the usage of Strongly-Typed Genetic Programming preserves validity throughout the entire search process, with only compilable test cases being generated.

Wappler et al. were also the first to take runtime exceptions into account and address the topic of unfeasible test cases [43]. They proposed a minimizing distance-based fitness function in order to assess and differentiate the test programs that are generated during the evolutionary search, which rated the test programs according to their distance to the given test goal (the program element to be covered). The aim of each individual search was to generate a test program that covers a particular branch of the class under test. This fitness function makes use of a distance metric that is based on the number of unexecuted methods of a Method Call Sequence if a runtime exception occurs. Unlike previous approaches in this area, the search is guided in case of uncaught runtime exceptions.

Our Test Case Evaluation approach (Section 4.4) also deals with the issue of runtime exceptions and unfeasible test cases. We propose tackling this challenge by defining weighted Control-Flow Graph nodes. The direction of the search is under constant adaptation, as the weight of Control-Flow Graph nodes is dynamically re-evaluated every generation. This causes the fitness of feasible test cases to fluctuate throughout the search process, allowing unfeasi-

ble test cases to be considered at certain points of the evolutionary search.

In [1,3,35], Arcuri et al. have focused on the testing of Container Classes (e.g., `Vector`, `Stack`, `BitSet`). Besides analysing how to apply different search algorithms (Random Search, Hill Climbing, Simulated Annealing, Genetic Algorithms, Memetic Algorithms and Estimation of Distribution Algorithms) to the problem and exploiting the characteristics of this type of software to help the search, more general techniques that can be applied to Object-Oriented software were studied.

In [16,17,48], the authors proposed augmenting Tonella's Genetic Algorithm-based approach [39] by integrating Symbolic Execution into the process. Symbolic Execution was used primarily for primitive-type method argument selection, whereas Evolutionary Testing is employed to generate Method Call Sequences; the objective was that of mitigating the weaknesses of both methodologies. The test case generation framework described (*Evacon*) is reported to outperform *eToc* and also other symbolic execution and random testing approaches.

There has been little investigation of the relationship between the size of the input domain (the search space) and performance of search-based algorithms; Harman et al. [12] were the first to characterise and empirically explore the search space/search algorithm relationship for search-based test data generation. In this work, static analysis was used to remove irrelevant variables for a given test data generation problem (i.e., from the set of possible input vector parameter-value combinations) thereby reducing the search space size. However, this study focused on procedural software and primitive parameter values. To the best of our knowledge, only two works addressed the issue of reducing the input domain of Object-Oriented test data generation problems.

In [1], Arcuri and Yao presented a way to reduce the search space for Object-Oriented software by dynamically eliminating the functions that cannot give any further help to the search, so as to avoid inserting method calls that do not change the state of the object in the Method Call Sequence. For determining if a function was read-only, a syntactic analysis of the source code was performed. Additionally, and because only container classes were used in the experiments, a database of common read-only function names (e.g., `insert`, `add`, `push`) was built and used to eliminate such functions using string matching algorithms. For the container classes employed in the experiments an improvement of 65.5% (on average) was reported in terms of efficiency.

In Arjan Seesing's Master Thesis report [38], Purity Analysis was proposed as a means to improve the performance of a search-based approach to test case generation for Object-Oriented software. A Genetic Programming approach was employed for creating test software for Object-Oriented systems, and Purity Analysis was integrated into the test tool described (*EvoTest*). Its usage is reported to almost double the coverage/time performance of the tool. However, the methodology lacked complete automation; it was stated that the analysis performed by the *EvoTest* tool still made many mistakes, and manual annotations were allowed and used to complement the information generated automatically. Additionally, the usage of Parameter Purity Analysis is not reported. Also, because Input Domain Reduction was not the primary focus of this work, the procedure is not thoroughly explored and described.

Our Input Domain Reduction strategy builds on the concept of Purity Analysis; however, our methodology, described with detail in Section 4.2, is systematic and fully automated. What's more, we introduce the usage of Parameter Purity Analysis, which allows the automatic identification and removal of entries even if the corresponding methods are not entirely pure.

Interesting review articles in on the topic of Evolutionary Testing include [2,7,24,25]. In [2], several issues in the current state-of-art of test data generation for Object-Oriented software are pinpointed, namely the unexistence of a common benchmark cluster which can be used to test and compare different techniques and the lack of theoretical work on the subject. McMinn [25] points out that extensions to search-based structural test data generation for Object-Oriented systems are complicated by problems of internal states, since objects are inherently state-based.

4. Technical approach

In this section, our evolutionary approach for automatic test case generation is described. The concepts presented were implemented into the *eCrash* automated test case generation tool for Object-Oriented Java software [30]. The process is summarized in Fig. 3.

The *eCrash* tool was employed to empirically assess the impact of our Test Case Evaluation and Input Domain Reduction strategies; the experimental studies are detailed in Section 5.

4.1. Test object analysis and instrumentation

The test object instrumentation and Control-Flow Graph generation tasks must precede the test case generation and evaluation phases, so as to allow tracing the structural entities traversed during test case execution. Probe insertion and Control-Flow Graph computation are performed at the Java bytecode level. Java bytecode is an assembly-like language that retains much of the high-level information about the original source program [42]. Class files (i.e., compiled Java programs containing bytecode information) are a portable binary representation that contains class related data, such as the class's name, information about the variables and constants, and the bytecode instructions of each method. Given that the target object's source code is often unavailable, working at the bytecode level allows broadening the scope of applicability of software testing tools; they can be used, for instance, to perform structural testing on third-party components.

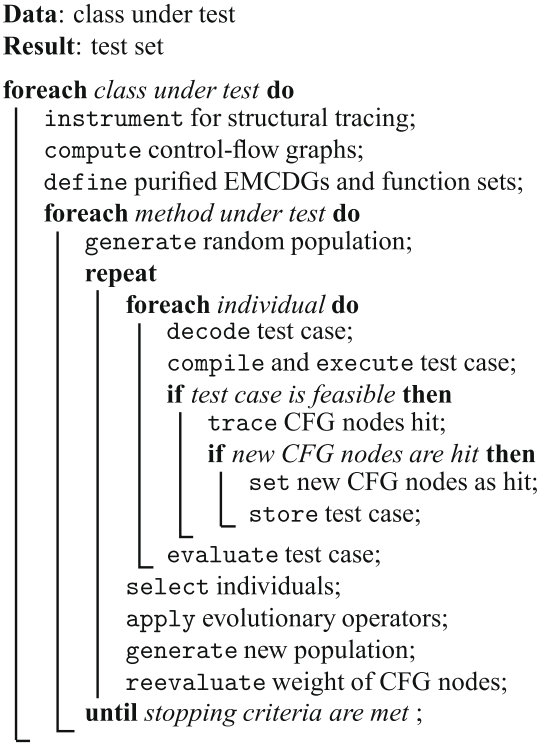


Fig. 3. Methodology overview.

The Control-Flow Graph building procedure involves grouping bytecode instructions into a smaller set of *Basic Instruction* and *Call blocks*, with the intention of easing the representation of the test object's control flow [31]. For the example Control-Flow Graph depicted in Fig. 4, attaining full structural coverage involves traversing nodes 4, 5, 8, 11, 12, 15 (Basic Instruction blocks) and 2, 6, 9, 13 (Call blocks). Additionally, other types of blocks, which represent virtual operations are defined: Entry blocks (e.g., block 1 in Fig. 4), Exit blocks (e.g., blocks 18–23 in Fig. 4), and Return blocks (e.g., blocks 3, 7, 10, 14 in Fig. 4). These *Virtual blocks* encompass no bytecode instructions, and are used to represent certain control flow hypothesis.

Test Object Analysis and Instrumentation is performed statically with the aid of the Sofya framework [18], a dynamic Java bytecode analysis framework. The Sofya package provides implementations and tools for the construction of various kinds of graphs – most notably Control-Flow Graphs – and native capabilities for dispatching event streams of specified program observations, which include instrumentators, event dispatchers, and event selection filters for semantic and structural event streams.

4.2. Function set definition

With our approach, Method Call Sequences are encoded as Strongly-Typed Genetic Programming trees; each tree subscribes to a Function Set, which defines the Strongly-Typed Genetic Programming nodes legally permitted in the tree, and establishes the constraints involved in Method Call Sequence construction. For modelling call dependences and defining the Function Set, an Extended Method Call Dependence Graph (EMCDG) [43,45] is employed. Our Input Domain Reduction strategy involves the removal of irrelevant edges from the Extended Method Call Dependence Graph, in order to build the *purified Extended Method Call Dependence Graph*. The *purified Function Set* is computed with basis on the purified Extended Method Call Dependence Graph, so as to include only those entries that are relevant to the search [34]. The algorithm depicted in Fig. 5 summarizes this process, which is detailed in the remaining of this section; the empirical studies performed in order to assess the validity of this approach are described and discussed in Section 5.2.

The first task of the purified Function Set generation phase is that of defining the test cluster. In order to do so, the Class Under Test provided by the user is firstly loaded. Next, the Class Under Test's public members – i.e., its public methods and constructors – are identified by means of the Java Reflection API in order to define the *public members list*.

The *Method Under Tests list* includes only the public methods – which are to be the subjects of the unit-test case generation process. The test cluster for the Class Under Test is computed by analysing the signatures of these methods, and includes all the distinct parameter data types.

Our current methodology for extending the *public members list* involves complementing it with the public constructors of the data types included in the test cluster; for primitive data types, a set of constants defining acceptable and boundary values [33] is used to sample the search space.

At this point, the *Parameter Purity Analysis* is performed on the parameters (implicit and explicit) of the members contained in the Method Under Tests list with the aid of the Soot Java Optimization Framework [41]. Purity Analysis was implemented into the

```
public void reconfigure(Config cfg)
0:    aload_1
1:    invokevirtual cfg.Config.getSignalCount ()I (6)
4:    iconst_5
5:    if_icmple #18
8:    new <java.lang.Exception> (7)
11:   dup
12:   ldc "Too many signals." (8)
14:   invokespecial java.lang.Exception (java.lang.String)
17:   athrow
18:   aload_1
19:   invokevirtual cfg.Config.getPort ()I (10)
22:   sipush 8000
25:   if_icmplt #38
28:   aload_1
29:   invokevirtual cfg.Config.getPort ()I (10)
32:   sipush 8005
35:   if_icmple #48
38:   new <java.lang.Exception> (7)
41:   dup
42:   ldc "Invalid port." (11)
44:   invokespecial java.lang.Exception (java.lang.String)
47:   athrow
48:   aload_0
49:   aload_1
50:   putfield cfg.Controller.cfg Lcfg/Config; (2)
53:   aload_0
54:   aload_1
55:   invokevirtual cfg.Config.getSignalCount ()I (6)
58:   newarray <int>
60:   putfield cfg.Controller.signals [I (3)
63:   return
```

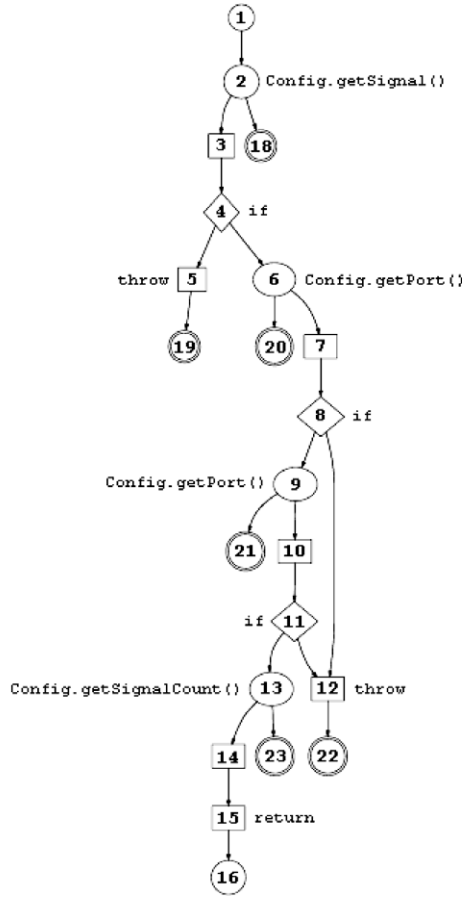


Fig. 4. Example Test Object's bytecode (left) and corresponding Control-Flow Graph (right).

```
Data: class under test
Result: purified function set
compute public members list;
compute methods under test list;
compute test cluster;
extend public members list;
foreach public member do
    foreach parameter do
        annotate parameter purity;
    compute types required table;
    compute types provided table;
initialize EMCDG with nodes;
connect EMCDG nodes with edges;
remove irrelevant edges;
create purified EMCDG;
create purified function set;
```

Fig. 5. Input Domain Reduction overview.

Soot framework by Antoine Mine, in conformity to the methodology proposed by Salcianu and Rinard [36,37]. Parameters are annotated as being *safe*, *read-only* or *read/write* (Section 2.3.2).

Next, two tables are computed: a *types required table*, which identifies the data types required to build a method call for each member (i.e., the parameter data types); and a *provider members table*, which identifies the data types potentially supplied by each member (i.e., the parameter reference data types and the return types). These tables are built with basis on the methods' signatures and return type information, and the entries are labelled with information on the *associated item*. The *associated item* label links data types to the implicit parameter, to an explicit parameter, or to the return value; it allows the unambiguous definition of the data type's provider/consumer. Without this information, it would not be possible to construct the instructions in the posterior test case generation phase (Section 4.3).

The list of possible labels for the *associated item* is the following:

- *implicit parameter (IP)* – the data type is associated to the implicit parameter;
- *explicit parameter* ($P\rho$) – the data type is associated to the explicit parameter number $\rho, \rho \in \aleph^*$;
- *return value (RE)* – the data type is associated to the return value.

At this point, all the data required for building the Extended Method Call Dependence Graph and modelling call dependences has been assembled. The Extended Method Call Dependence Graph is initialized, in accordance to the information contained in the test cluster and public members list tables, with two types of nodes: *member nodes* and *data type nodes*.

The Extended Method Call Dependence Graph nodes are then connected, in accordance to the information contained in the *data types required table* and *provider members table*: a directed edge between a data type (origin) and a member (destination) means that the data type at the origin is provided by the member at the destination; a direct edge between a member (origin) and a data type (destination) means that the member at the origin requires the data type at the destination. Edge information is complemented with a label containing information on the *associated item*, in order to complete the Extended Method Call Dependence Graph definition.

The *purified Extended Method Call Dependence Graph* is computed with basis on the Extended Method Call Dependence Graph and on the parameters' purity information, in accordance to the algorithm depicted in Fig. 6. The purified Extended Method Call Dependence Graph is obtained by removing the edges representing *safe* and *read-only* parameters from the Extended Method Call Dependence Graph (e.g., Fig. 9). Finally, the *purified Function Set* is defined with basis on the *purified Extended Method Call Dependence Graph*, in accordance to the algorithm shown in Fig. 7. Each entry in the Function Set table contains information on the types required (*child types* column) and types provided (*return types* column) by the corresponding member (e.g., Table 3).

4.3. Test case generation

As was abovementioned, potential solutions (i.e., test cases) are encoded as Strongly-Typed Genetic Programming trees. The process of decoding the genotype (i.e., the Strongly-Typed Genetic Programming trees) to the phenotype (i.e., the Method Call Sequences) involves the linearisation of the Strongly-Typed Genetic Programming tree using a depth-first transversal algorithm; test case source-code generation is performed by translating the linearised Strongly-Typed Genetic Programming tree to a Method Call Sequence using the method signature information embedded into each Strongly-Typed Genetic Programming node (e.g., Fig. 8).

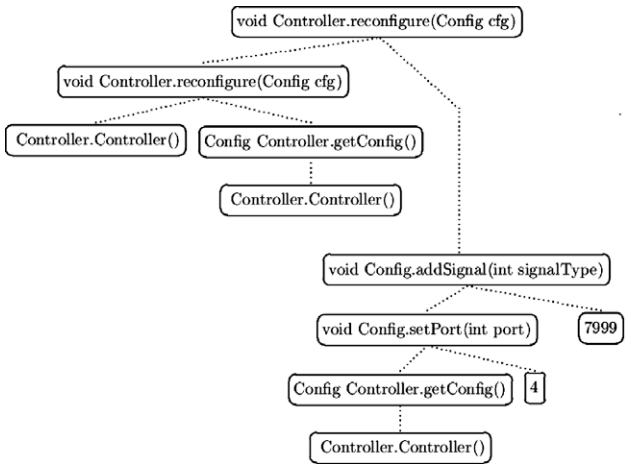
For evolving test cases, the Evolutionary Computation in Java (ECJ) package [23] is used. ECJ is a research package that incorporates several Universal Evolutionary Algorithms, and includes built-in support for Strongly-Typed Genetic Programming. It is highly flexible, having nearly all classes and their settings being

```
Data: EMCDG, parameter purity information
Result: purified EMCDG
foreach EMCDG member node do
    foreach incoming edge do
        if associated item is not RETURN then
            parameterPurity ← get parameter purity;
            if parameterPurity is SAFE or READ-ONLY
            then
                remove incoming edge;
            end
        end
    end
end
```

Fig. 6. Purified EMCDG generation algorithm.

```
Data: purified EMCDG
Result: purified function set
foreach EMCDG member node do
    create new function set entry for member;
    foreach outgoing edge do
        dataType ← get destination node;
        add dataType to member child types;
    end
    foreach incoming edge do
        dataType ← get origin node;
        add dataType to member return types;
    end
end
```

Fig. 7. Purified Function Set generation algorithm.




```
Controller controller0 = new Controller();
Controller controller1 = new Controller();
Config config2 = controller1.getConfig();
controller0.reconfigure(config2);
Controller controller3 = new Controller();
Config config4 = controller3.getConfig();
int int5 = 4;
config4.setPort(int5);
int int6 = 7999;
config4.addSignal(int6);
controller.reconfigure(config4);
```

Fig. 8. Example Strongly-Typed Genetic Programming Tree (left) and corresponding Method Call Sequence (right).

dynamically determined at runtime by user provided Parameter Files and Function Files (e.g., Strongly-Typed Genetic Programming node and Strongly-Typed Genetic Programming tree constraints, maximum number of generations, probabilities for the evolutionary operators).

Test cases are evolved while there are Control-Flow Graph nodes left to be covered (i.e., an all-nodes criterion is used), or until a predefined number of generations is reached. Whenever a test case "hits" an unexercised Control-Flow Graph node, that node is removed from the *remaining nodes list*, and the test case is added to the test set.

After all the individuals of a given generation have been evaluated, and if the termination criteria have not been met, some of them are selected for being reproduced to the next generation (possibly after being mutated) or for participating in a crossover phase. The probability of selection is related to the fitness of the individuals, which is set in accordance to the outcome of the Test Case Evaluation process (detailed in the following subsection).

4.4. Test Case Evaluation

With our approach, the quality of feasible test cases is related to the structural entities of the Method Under Test which are the current targets of the evolutionary search [32]. Test cases that exercise less explored (or unexplored) Control-Flow Graph nodes and paths must be favoured. However, unfeasible test cases cannot be blindly penalized because, as a general rule, longer and more intricate test cases are more prone to throw runtime exceptions; still, they are often needed for defining elaborate state scenarios and traversing certain problem nodes.

The issue of steering the search towards the traversal of interesting control-flow paths was address by assigning weights to the Control-Flow Graph nodes; the higher the weight of a given node the higher the cost of exercising it, and hence the higher the cost of traversing the corresponding control-flow path. Also, the weights of Control-Flow Graph nodes are dynamically reevaluated every generation; the direction of the search is thus under constant adaptation. The remaining of this subsection details the Test Case Evaluation strategy proposed.

Let each Control-Flow Graph node $n \in N$ represent a linear sequence of computations (i.e., bytecode instructions) of the Method Under Test; each Control-Flow Graph edge e_{ij} represents the transfer of the execution control of the program from node n_i to the node n_j. Conversely, n_j is a successor node of n_i if an edge e_{ij} between the nodes n_i and n_j exists. The set of successor nodes of n_i is defined as $N_s^{ni}, N_s^{ni} \subset N$.

4.4.1. Weight reevaluation

The weight of traversing node n_i is identified as W_{ni}. At the beginning of the evolutionary search the weights of nodes are initialized with a predefined value W_{init}.

The Control-Flow Graph nodes' weights are reevaluated every generation according to Eq. (1).

$$W_{ni} = (\alpha W_{ni})\left(\frac{hitC_{ni}}{|T|} + 1\right)\left(\frac{\sum_{x \in N_s^{ni}} W_x}{\left|N_s^{ni}\right| \times \frac{W_{init}}{2}}\right) \tag{1}$$

The $hitC_{ni}$ parameter is the "Hit Count", and contains the number of times a particular Control-Flow Graph node was exercised by the test cases of the previous generation. T represents the set of test cases produced in the previous generation, with $|T|$ being its cardinality.

The constant value $\alpha, \alpha \in]0, 1]$ is the *weight decrease constant*.

In summary, each generation the weight of a given node is multiplied by:

- the *weight decrease constant* value α, so as to decrease the weight of all Control-Flow Graph nodes indiscriminately;
- the *hit count factor*, which worsens the weight of recurrently hit Control-Flow Graph nodes;
- the *path factor*, which improves the weight of nodes that lead to interesting nodes and belong to interesting paths.

After being reevaluated, the weights of all the nodes are normalized to the nodes' initial weight W_{ni} in accordance to Eq. (2).

$$W_{ni} = \frac{W_{ni} \times W_{init}}{W_{max}} \tag{2}$$

W_{max} corresponds to the maximum value for the weight existing in N.

4.4.2. Evaluation of feasible test cases

For feasible test cases, the fitness is computed with basis on their trace information; relevant trace information includes the "Hit List" – i.e., the set $H_t, H_t \subseteq N$ of traversed Control-Flow Graph nodes. The fitness of feasible test cases is, thus, evaluated as follows:

$$Fitness_{feasible}(t) = \frac{\sum_{h \in H_t} W_h}{|H_t|} \tag{3}$$

This strategy causes the fitness of feasible test cases that exercise recurrently traversed structures to fluctuate throughout the search

process; frequently hit nodes will have their weight increased, thus worsening the fitness of the test cases that exercise them.

4.4.3. Evaluation of unfeasible test cases

For unfeasible test cases, the fitness of the individual is calculated in terms of the distance between the runtime exception index $exInd_t$ (i.e., the position of the method call that threw the exception) and the Method Call Sequence length $seqLen_t$.

Also, an *unfeasible penalty constant* value β is added to the final fitness value, so as to penalise unfeasibility.

$$Fitness_{unfeasible}(t) = \beta + \frac{(seqLen_t - exInd_t) \times 100}{seqLen_t} \quad (4)$$

With this methodology, and depending on the value of β and on the fitness of feasible test cases, unfeasible test cases may be selected for breeding at certain points of the evolutionary search, thus favouring the diversity and complexity of Method Call Sequences.

5. Experimental studies

In this section, the empirical studies implemented with the objectives of validating and observing the impact of our Test Case Evaluation (Section 5.1) and Input Domain Reduction (Section 5.2) strategies are described and discussed.

The Java `Stack` and `BitSet` classes (JDK 1.4.2) were used as test objects. The rationale for employing these classes is related with the fact that they represent "real-world" problems and, being container classes, possess the interesting property of containing explicit state, which is only controlled through a series of method calls [1]. Additionally, they have been used in several other case studies described in literature (e.g., [1,17,39,45]), providing an adequate testbed in the lack of common benchmark cluster that can be used to test and compare different techniques [2].

5.1. Test Case Evaluation experiments

The main objective of this case study was that of experimenting with different configurations for the *probabilities of evolutionary operators* – mutation, reproduction and crossover (Section 2.2) – and for the values of the *test case evaluation parameters* – the *weight decrease constant* α (Eq. (1)) and the *unfeasible penalty constant* β (Eq. (4)).

For evolving test cases, ECJ was configured using a single population of five individuals. The Method Under Tests' Control-Flow Graph nodes were initialized with a weight W_{ni} of 200. The search stopped if an ideal individual was found or after 200 generations.

For the generation of individuals a multi-breeding pipeline was used, which stored three child sources; each time an individual had to be produced, one of those sources was selected with a predefined probability. The available breeding pipelines were the following:

- a *Reproduction pipeline*, which simply makes a copy of the individuals it receives from its source;
- a *Crossover pipeline*, which performs a strongly-typed version of Subtree Crossover [19] – two individuals are selected, a single tree is chosen in each such that the two trees have the same constraints, a random node is chosen in each tree such that the two nodes have the same return type, and finally the swap is performed;
- and a *Mutation pipeline*, which implements a strongly-typed version of Point Mutation [19] – an individual is selected, a random node is selected, and the subtree rooted at that node is replaced by a new valid tree.

The selection method employed was *Tournament Selection* with a size of 2, which means that first 2 individuals are chosen at random from the population, and then the one with the best fitness is selected.

5.1.1. Probabilities of operators case study

This particular experiment was performed with the intention of assessing the impact of the evolutionary operators' probabilities on the test case generation process. In order to do so, four distinct parametrizations of the multi-breeding pipeline were defined, having:

1. a high probability of selecting the mutation pipeline;
2. a high probability of selecting the crossover pipeline;
3. a high probability of selecting the reproduction pipeline;
4. equal probabilities of selecting either pipeline.

For each of the above multi-breeding pipeline parametrizations, 20 runs were executed for the `Stack`'s methods, and 10 runs were executed for the `BitSet`'s methods.

The *weight decrease constant* α was set to 0.9, and the unfeasible penalty constant β was defined as 150. It should be noted that the definition of these values was heuristic, as no experiments had been performed that allowed a fundamented choice; these were conducted later, and are described in the following subsection.

Table 1 summarizes the results obtained. The statistics show that the strategy of assigning balanced probabilities (r:0.33 c:0.33 m:0.34) to the all of the breeding pipelines yields better results. For the `Stack` class, this configuration was the only one in which full coverage was achieved in all of the runs (in, at most, 200 generations), and it was also the best in terms of the average number of generations required to attain it; for the `BitSet` class, it was the only configuration in which full coverage was attained at least once for all the Method Under Tests. The worst results were obtained for the parametrization in which the reproduction breeding pipeline was given a high probability of selection.

5.1.2. Evaluation parameters case study

In this experiment, different combinations of values for the α and β parameters were studied, with the intention of analysing the impact of the Test Case Evaluation parameters on the evolutionary search.

The following values were used:

- α – 0.1, 0.5, and 0.9;
- β – 0, 150, and 300.

The probabilities of choosing the three breeding pipelines were chosen in accordance to the results yielded by the experiment described in Section 5.1.1 – i.e., the probabilities for reproduction, crossover and mutation were set to 0.33, 0.33 and 0.34, respectively. The other configurations remained unaltered. All the 9 combinations of the α and β values were employed; 20 runs were executed for each of the `Stack`'s Method Under Tests, and 5 runs where executed for the `BitSet`'s methods.

The results obtained are summarized in Table 2. The statistics clearly show that the best configuration for the test case evaluation parameters is that of assigning a value of 150 to β and a value of 0.5 to α.

5.1.3. Discussion

Automatic test case generation using search-based techniques is a difficult subject, especially if the aim is to implement a systematic solution that is adaptable to a wide range of test objects. Key to the definition of a good strategy is the configuration of parameters

Table 1
Statistics for the probabilities of operators case study. Relevant data includes the probabilities of choosing the reproduction (r), crossover (c) and mutation (m) pipelines and, for each configuration, the percentage of runs in which full coverage was achieved (*%f*) and the average number of generations required attain full coverage (*#g*).

	r:0.1 c:0.1 m:0.8		r:0.8 c:0.1 m:0.1		r:0.1 c:0.8 m:0.1		r:0.3 c:0.3 m:0.3	
	#g	*%f*	*#g*	*%f*	*#g*	*%f*	*#g*	*%f*
Stack								
empty	10.2	100	11.2	100	17.5	100	4.5	100
peek	6.6	100	10.7	100	9.4	100	2.8	100
pop	6.5	100	8.9	100	8.6	100	2.8	100
push	20.6	100	16.4	57	37.2	95	2.5	100
search	48.9	95	48.2	57	98.8	82	18.7	100
BitSet								
hashCode	1.4	100	2.0	100	1.3	100	1.4	100
clear(int)	65.3	100	154.8	40	140.8	50	92.1	90
clear( )	7.2	100	31.2	100	28.6	100	12.6	100
clear(int,int)	181.5	20	–	0	–	0	175.4	30
toString	7.6	100	29.6	100	32.0	100	8.6	100
isEmpty	7.6	100	52.4	100	32.0	100	8.6	100
length	41.4	100	125.4	60	126.2	70	49.2	100
get(int,int)	–	0	–	0	–	0	186.2	30
get(int)	57.8	100	184.8	30	137.4	80	75.8	100
size	1.2	100	2.0	100	1.2	100	1.2	100
set(int,boolean)	24.8	100	29.0	100	37.8	100	23.6	100
set(int,int)	42.8	100	122.2	100	109.0	100	44.8	100
set(int,int,boolean)	32.2	100	68.2	100	97.4	80	25.0	100
set(int)	37.2	100	127.0	80	86.8	100	51.8	100
flip(int,int)	80.4	80	184.8	60	165.0	40	89.8	100
flip(int)	73.4	100	174.0	40	148.6	60	77.8	100
andNot	38.0	100	–	0	104.8	100	20.8	100
cardinality	7.6	100	52.4	100	32.0	100	8.6	100
intersects	127.8	60	–	0	150.6	50	107.4	60
nextSetBit	105.8	100	192.2	20	192.8	20	114.6	60
xor	80.6	80	123.6	50	133.4	40	70.3	90
Averages								
Stack	18.6	99	19.1	83	34.3	95	6.3	100
BitSet	51.1	80	97.4	56	92.5	65	59.3	81

Table 2
Statistics for the Evaluation Parameters case study. Relevant data includes the percentage of runs in which full coverage was achieved (*%f*) and the average number of generations required attain full coverage (*#g*). Using different combinations for the alpha (α) and beta (β) parameters.

β	0						150						300					
α	0.1		0.5		0.9		0.1		0.5		0.9		0.1		0.5		0.9	
	%f	*#g*	*%f*	*#g*	*%f*	*#g*	*%f*	*#g*	*%f*	*#g*	*%f*	*#g*	*%f*	*#g*	*%f*	*#g*	*%f*	*#g*
Stack																		
hline empty	100	5	100	6	100	5	100	5	100	5	100	5	100	5	100	5	100	5
peek	100	3	100	4	100	3	100	3	100	3	100	3	100	3	100	3	100	3
pop	100	3	100	3	100	3	100	2	100	2	100	3	100	3	100	3	100	3
push	100	5	100	5	100	5	100	5	100	5	100	3	100	5	100	5	100	5
search	100	18	100	18	100	22	100	16	100	16	100	19	100	16	100	21	100	22
BitSet																		
hashCode	100	2	100	2	100	2	100	2	100	2	100	2	100	2	100	2	100	2
clear(int)	20	163	80	133	80	107	40	126	100	123	100	105	40	126	100	122	80	135
clear( )	100	15	100	8	100	8	100	13	100	8	100	8	100	7	100	8	100	8
clear(int,int)	0	–	0	–	0	–	0	–	40	177	0	–	0	–	40	180	20	181
toString	100	14	100	8	100	8	100	13	100	8	100	8	100	8	100	8	100	8
isEmpty	100	10	100	8	100	8	100	10	100	8	100	8	100	10	100	8	100	8
length	80	88	100	59	100	49	100	106	100	67	100	49	100	109	100	44	100	49
get(int,int)	0	–	0	–	0	–	0	–	20	188	0	–	0	–	0	–	0	–
get(int)	60	136	100	97	100	87	100	96	100	83	100	96	60	146	100	86	100	70
size	100	1	100	1	100	1	100	1	100	1	100	1	100	1	100	1	100	1
set(int,boolean)	100	73	100	15	100	17	100	73	100	15	100	17	100	73	100	15	100	17
set(int,int)	60	130	100	40	100	66	60	122	100	45	100	60	60	129	100	45	100	54
set(int,int,boolean)	100	63	100	25	100	24	100	47	100	25	100	27	100	44	100	25	100	27
set(int)	80	106	100	68	100	42	60	108	100	62	100	60	60	113	100	62	80	63
flip(int,int)	60	145	100	66	80	111	60	122	100	94	100	94	80	136	100	70	100	95
flip(int)	20	178	60	144	80	126	40	170	100	102	100	79	60	139	100	97	100	82
andNot	40	137	100	21	100	47	60	147	100	21	100	47	80	125	100	21	100	47
cardinality	100	15	100	8	100	8	100	9	100	8	100	8	100	16	100	8	100	8
intersects	0	–	40	149	60	171	20	173	60	149	60	149	0	–	20	190	60	180
nextSetBit	20	166	80	118	60	140	40	158	60	132	60	132	40	172	60	145	60	139
xor	0,8	124	100	34	100	51	80	92	100	26	100	37	60	89	100	33	100	37
Averages																		
Stack	100	7	100	7	100	8	100	6	100	6	100	6	100	6	100	7	100	8
BitSet	63	87	84	53	84	57	70	84	90	64	87	52	69	80	87	59	86	60

so as to find a good balance between the *intensification* and the *diversification* of the search. With our approach, the test case evaluation parameters α and β and the evolutionary operators' selection probabilities play a central role in the test case generation process.

The main task of the mutation and crossover operators is that of diversifying the search, allowing it to browse through a wider area of the search landscape and to escape local maximums; the task of intensifying the search and guiding it towards the transversal of unexercised structures is performed as a result of the strategy of assigning weights to Control-Flow Graph nodes.

Nevertheless, to strong a bias towards the breeding of feasible test cases will hinder the generation of more complex test cases, which are sometimes needed to exercise problem structures in the test object; on the other hand, if feasible test cases are not clearly encouraged, the search process will wander.

This issue was addressed by allowing the fitness of feasible test cases to fluctuate throughout the search process as a result of the impact of the α and β parameters, in order to allow unfeasible test cases to be selected at certain points of the evolutionary search.

The experiments performed allow drawing a preliminary conclusion: the assumption made on the Probabilities of Operators case study (Section 5.1.1), in which $\alpha = 0.9$ was employed as being an adequate value, was incorrect. Using a value of 0.5 for this evaluation parameter yielded better results.

On the other hand, it is possible to affirm that the strategy of assigning the value 150 to the *unfeasible penalty constant* β yields good results. An explanation for this behaviour follows.

The worst value a Control-Flow Graph node can have is 200 – since the weights of Control-Flow Graph nodes are normalized each generation (Eq. (2)). If all the nodes exercised by a feasible test case have the worst possible value – because they are being recurrently exercised by test cases, i.e., because the search is stuck in a local maximum – the fitness of the corresponding test case will also be 200 (Eq. (3)).

However, for a given unfeasible test case t, if $exInd_t \leqslant \frac{seqLen_t}{2}$ and $\beta = 150$, then $Fitness_{unfeasible}(t) \in [150, 200]$, i.e., if the exception index of a given unfeasible test case is lower or equal to half of its Method Call Sequence length, and if the value 150 is used for β, then the fitness of that test case will belong to the interval 150 to 200.

This means that, with $\beta = 150$, *some* good unfeasible test cases may be selected for breeding; conversely, if $\beta = 0$, *all* unfeasible test cases will be evaluated with relatively good fitness values, and if $\beta = 300$, *none* of the unfeasible test cases will be evaluated as being interesting. The concept of good unfeasible test cases, in this context, can thus be verbalized as being a test case in which at least half of the Method Call Sequence is executed without an exception being thrown.

Assigning the value $\beta = W_{init} - 50$ is, thus, a good compromise between the need to penalize unfeasible test cases and the need to consider them at some points of the evolutionary search.

5.2. Input Domain Reduction experiments

This subsection describes the case studies implemented with the objectives of observing the impact of our Input Domain Reduction proposal – both in terms of the size of the input domain (Section 5.2.1) and of the results yielded by the *eCrash* tool (Section 5.2.2).

5.2.1. Input Domain Size case study

With our approach, the Extended Method Call Dependence Graph models call dependences and the Function Set encompasses the entries from which the test case generation algorithm can choose when evolving test programs; as such, the impact of our Input Domain Reduction strategy on the size of the search space is best assessed by comparing the purified Function Sets and Extended Method Call Dependence Graphs with those obtained when no Parameter Purity Analysis is employed.

Fig. 9 and Table 3 illustrate, respectively, the purified Extended Method Call Dependence Graph and the purified Function Set generated for the `Stack` class; additionally, the original Extended Method Call Dependence Graph and the entries excluded from the Function Set as a result of the Parameter Purity Analysis procedure are shown for comparison purposes.

Table 3
Purified Function Set for the `Stack` class (top) and entries excluded from the Purified Function Set (bottom).

Member	Return type	Child types
Stack( )	Stack [IP]	
Object pop( )	Object [RE]	Stack [IP]
Object pop( )	Stack [IP]	Stack [IP]
Object push(Object)	Object [RE]	Stack [IP]
		Object [P0]
Object push(Object)	Stack [IP]	Stack [IP]
		Object [P0]
Object peek( )	Object [RE]	Stack [IP]
Object( )	Object [RE]	
Entries excluded		
Object push(Object)	Object [P0]	Stack [IP]
		Object [P0]
Object peek( )	Object [IP]	Stack [IP]
boolean empty( )	Stack [IP]	Stack [IP]
int search(Object)	Stack [IP]	Stack [IP]
		Object [P0]
int search(Object)	Stack [P0]	Stack [IP]
		Object [P0]

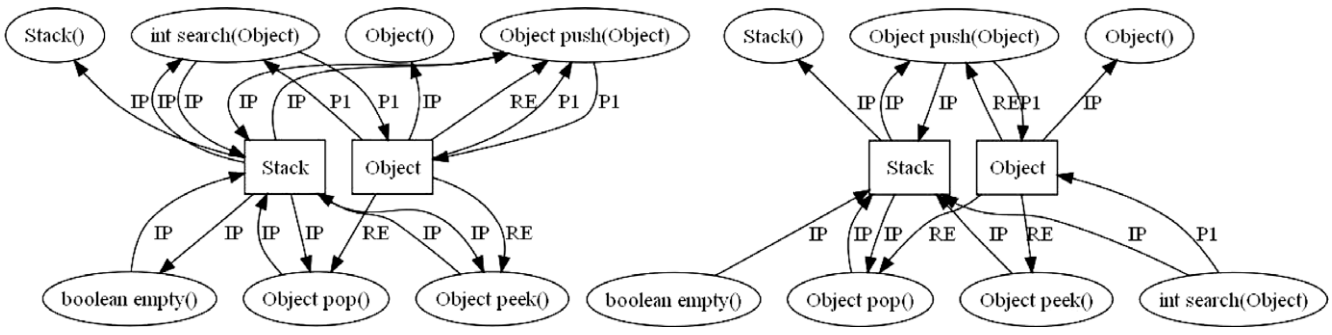


Fig. 9. EMCDG (left) and purified EMCDG (right) for the `Stack` class.

The statistics depicted in Table 4 for the `Stack` and `BitSet` classes show a clear reduction in the size of the input domain; the number of Function Set entries in the Purity column is only 65.2% of those obtained when no Parameter Purity Analysis is used.

5.2.2. Test case generation results case study

This subsection compares the results yielded by the *eCrash* tool when the Parameter Purity Analysis phase is included and excluded from the process. A single population of 10 individuals was used. Twenty runs were executed for each of the Method Under Tests. The Method Under Tests' Control-Flow Graph nodes were initialized with a weight of 200, with the α and β (Subsection 4.3) parameters being set to 0.5 and 150, respectively. The search stopped if an ideal individual was found or after 100 generations. For breeding individuals, three pipelines were used: a Reproduction pipeline, a Crossover pipeline, and a Mutation pipeline; the probabilities of choosing these pipelines were set equal values. The selection method employed was Tournament Selection with a size of 2. Table 5 presents the results obtained for the `Stack` and `BitSet` classes.

Table 4
Statistics for the Input Domain Size case study. Relevant data includes the number of EMCDG edges and Function Set entries obtained for the `Stack` and `BitSet` classes with and without Parameter Purity Analysis.

	No purity		Purity	
	EMCDG edges	FS entries	EMCDG edges	FS entries
`Stack`	19	12	14	7
`BitSet`	106	54	88	36

Table 5
Statistics for the Test Case Generation Results case study. Relevant data includes the average number of generations required to attain full structural coverage (*gens*) and the percentage of runs attaining full structural coverage (*full*) for the public members of the `Stack` and `BitSet` classes with and without Parameter Purity Analysis.

	No purity		Purity	
	Gens	Full (%)	Gens	Full (%)
`Stack`				
pop	4.5	100	1.3	100
push	1.9	100	1.9	100
empty	7.1	100	1.4	100
peek	4.2	100	1.3	100
search	9.4	100	6.9	100
`BitSet`				
hashCode	1.6	100	1.3	100
clear(int)	43.3	55	37.7	60
clear( )	15.8	100	17.7	100
clear(int,int)	–	0	63.0	10
toString	21.6	100	18.4	100
isEmpty	13.4	90	4.2	90
length	48.0	50	47.4	95
get(int,int)	75.5	15	–	0
get(int)	13.2	50	41.3	60
size	1.6	100	1.3	100
set(int,boolean)	13.5	100	9.2	90
set(int,int)	42.2	90	36.6	90
set(int,int,boolean)	42.0	90	39.4	100
set(int)	26.0	70	25.4	90
flip(int,int)	48.8	40	57.7	35
flip(int)	41.0	60	33.5	60
andNot	38.2	45	26.0	70
cardinality	10.9	100	9.7	100
intersects	58.5	40	61.8	40
nextSetBit	68.0	10	56.0	45
xor	34.9	70	29.8	90
Averages				
`Stack`	*5.4*	*100.0*	*2.6*	*100.0*
`BitSet`	*32.9*	*65.5*	*30.9*	*72.6*

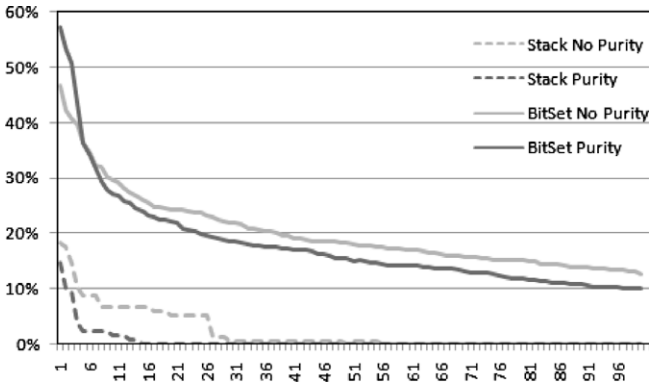


Fig. 10. Average percentage of nodes remaining per generation for the `Stack` and `BitSet` classes with and without Parameter Purity Analysis.

For the `Stack` class, the number of generations required to attain full coverage using Parameter Purity Analysis was, on average, 2.6 – less than half of those required when no Parameter Purity Analysis was employed. All the runs yielded full coverage in both cases. For the `BitSet` class – and although 33.3% of the Function Set entries were eliminated when Parameter Purity Analysis was used – the improvement was not as clear. Still, the average percentage of test cases that accomplished full coverage within a maximum of 100 generations increased approximately 7%.

The graphs shown in Fig. 10 represent the average number of Control-Flow Graph nodes left to be covered per generation. Again, the results obtained for the `Stack` yield a significant improvement, whereas those presented for the `BitSet` test object show a slight (but clear) improvement.

5.2.3. Discussion

The results observed in the Input Domain Size experiment indicate that the search space of Evolutionary Testing problems can be dramatically reduced by embedding Parameter Purity Analysis into the test case generation process. For the test objects used, approximately a third of the set of entries that could potentially be selected for integrating the generated test cases were discarded; these instructions would have no (positive) impact on the definition of test scenarios.

In terms of the results yielded by the *eCrash* automated test case generation tool when Parameter Purity Analysis is used, a significant improvement is clearly observable in terms of the efficiency of the search – i.e., fewer generations (and, consequently, less computational time) are required to find an adequate test set if the conditions are, otherwise, similar.

Finally, it should be mentioned that the Input Domain Reduction strategy proposed also enhances the test case generation process indirectly, by preventing irrelevant instructions from obstructing the search by throwing runtime exceptions and rendering test cases unfeasible.

This fact is especially pertinent given that our Test Case Evaluation methodology does consider unfeasible test cases for breeding at certain points of the evolutionary search (Section 4.3). The inclusion of a Parameter Purity Analysis phase into the process thus strengthens our Test Case Evaluation proposal, by ensuring that unfeasible Method Call Sequences are composed solely by instructions that are relevant in terms of state scenario definition.

6. Conclusions

Evolutionary Testing is an emerging methodology for automatically generating high quality test data. It is, however, a difficult

subject, especially if the aim is to implement an automated solution, viable with a reasonable amount of computational effort, which is adaptable to a wide range of test objects.

The state problem of Object-Oriented programs requires the definition of carefully fine-tuned methodologies that promote the transversal of problematic structures and difficult control-flow paths. We proposed tackling this particular challenge by defining weighted Control-Flow Graph nodes. The direction of the search is under constant adaptation, as the weight of Control-Flow Graph nodes is dynamically reevaluated every generation. Also, the fitness of feasible test cases fluctuates throughout the search process; this strategy allows unfeasible test cases to be considered at certain points of the evolutionary search – once the feasible test cases that are being bred cease to be interesting because they exercise recurrently traversed structures. In conjunction with the impact of the evolutionary operators, a good compromise between the intensification and diversification of the search can be achieved.

Additionally, an Input Domain Reduction methodology, based on the concept of Parameter Purity Analysis, for eliminating irrelevant variables from Object-Oriented test case generation search problems was proposed. With our approach, test cases are evolved using the Strongly-Typed Genetic Programming paradigm; Purity Analysis is particularly useful in this context, as it provides a means to automatically identify and remove Function Set entries that do not contribute to the definition of interesting test scenarios. Nevertheless, the concepts presented are generic and may be employed to enhance other search-based test case generation methodologies in a systematic and straight-forward manner. The observations made indicate that the Input Domain Reduction strategy presented has a highly positive effect on the efficiency of the test case generation algorithm; less computational time is spent to achieve results.

The process of "trimming" the input domain in order to eliminate irrelevant entries also ensures that test cases are not rendered unfeasible by the inclusion of unsuitable instructions; this strategy is thus of special importance, given that our Test Case Evaluation strategy does consider unfeasible test cases at certain stages of the search.

6.1. Future work

Several open problems persist in the area of search-based test case generation. Future work will be mainly focused on addressing the challenges posed by the three cornerstones of Object-Oriented programming: *encapsulation*, *inheritance*, and *polymorphism*. The importance of the inheritance and polymorphism properties, in particular, is yet to be fully studied by researchers in this area. Inheritance allows the treatment of an object as its own type or its base type; polymorphism means "different forms", and allows one type to express its distinction from another similar type through differences in behaviour of the methods that can be called through the base class [9]. *Search space sampling* deals with the inclusion of *all* the relevant variables to a given test object into the test data generation problem, so as to enable the coverage of the entire search space whenever possible and improve the effectiveness the approach. Because the test cluster cannot possibly include all the subclasses that may override the behaviours of the classes which are relevant for the test object, adequate strategies for search space sampling – which take the commonality among classes and their relationships with each other into account – are of paramount importance.

In the near future, we will also be proposing an adaptive strategy [14] for enhancing Genetic Programming-based approaches to automatic test case generation. This novel Adaptive Evolutionary Testing methodology aims to promote the introduction of relevant instructions into the generated test cases by means of mutation; the instructions from which the algorithm can choose are ranked, with their rankings being updated every generation in accordance to the feedback obtained from the individuals evaluated in the preceding generation. We are also planning to experiment with parallel systems in order to enhance our methodology's performance; the evaluation procedure, in particular, is inherently parallelizable, as all the test cases generated in a given generation can be executed simultaneously. Also, we intend to treat the test case selection process as a multi-objective optimization problem [49], so as to take into account several goals simultaneously – such as structural coverage and execution cost.

Acknowledgements

This paper has been partially funded by Projects TIN2007-68083-C02 (Spanish Ministry of Education and Culture, NoHNES – Non-Hierarchical Network Evolutionary System Project), and Project GRU09105, Junta de Extremadura, Consejeria de Economia-Comercio e Innovación and FEDER.

References

[1] Andrea Arcuri, Xin Yao, A memetic algorithm for test data generation of object-oriented software, in: Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC), IEEE, 2007, pp. 2048–2055.
[2] Andrea Arcuri, Xin Yao, On test data generation of object-oriented software, in: TAICPART-MUTATION'07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques – MUTATION, IEEE Computer Society, Washington, DC, USA, 2007, pp. 72–76.
[3] Andrea Arcuri, Xin Yao, Search Based Testing of Containers for Object-Oriented Software, Technical Report CSR-07-3, University of Birmingham, School of Computer Science, April 2007.
[4] Thomas Back, David B. Fogel, Zbigniew Michalewicz (Eds.), Handbook of Evolutionary Computation, IOP Publishing Ltd., Bristol, UK, 1997.
[5] Stéphane Barbey, Alfred Strohmeier, The problematics of testing object-oriented software, in: M. Ross, C.A. Brebbia, G. Staples, J. Stapleton (Eds.), SQM'94 Second Conference on Software Quality Management, Edinburgh, Scotland, UK, July 26–28, vol. 2, 1994, pp. 411–426.
[6] Boris Beizer, Software Testing Techniques, John Wiley & Sons, Inc., New York, NY, USA, 1990.
[7] Antonia Bertolino, Software testing research: achievements, challenges, dreams, in: FOSE'07: 2007 Future of Software Engineering, IEEE Computer Society, Washington, DC, USA, 2007, pp. 85–103.
[8] Kenneth A. De Jong, Evolutionary Computation, The MIT Press, 2002.
[9] Bruce Eckel, Thinking in Java, Prentice Hall Professional Technical Reference, 2002.
[10] Mark Harman, Automated test data generation using search based software engineering, in: AST'07: Proceedings of the Second International Workshop on Automation of Software Test, IEEE Computer Society, Washington, DC, USA, 2007, p. 2.
[11] Mark Harman, The current state and future of search based software engineering, in: FOSE'07: 2007 Future of Software Engineering, IEEE Computer Society, Washington, DC, USA, 2007, pp. 342–357.
[12] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Joachim Wegener, The impact of input domain reduction on search-based test data generation, in: ESEC-FSE'07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ACM Press, New York, NY, USA, 2007, pp. 155–164.
[13] Thomas D. Haynes, Dale A. Schoenefeld, Roger L. Wainwright, Type inheritance in strongly typed genetic programming, in: Peter J. Angeline, K.E. Kinnear Jr. (Eds.), Advances in Genetic Programming, vol. 2, MIT Press, Cambridge, MA, USA, 1996, pp. 359–376.
[14] Robert Hinterding, Zbigniew Michalewicz, A.E. Eiben, Adaptation in evolutionary computation: a survey, in: Proceedings of the Fourth International Conference on Evolutionary Computation (ICEC 97), IEEE Press, 1997, pp. 65–69.
[15] John H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, The MIT Press, 1992.
[16] Kobi Inkumsah, Tao Xie, Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs, in: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 2007, pp. 425–428.
[17] Kobi Inkumsah, Tao Xie, Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution, in: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), September 2008.

[18] Alex Kinneer, Matthew B. Dwyer, Gregg Rothermel, Sofya: supporting rapid development of dynamic program analyses for Java, in: ICSE COMPANION'07: Companion to the Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2007, pp. 51–52.
[19] John R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems), The MIT Press, 1992.
[20] Gary T. Leavens, Albert L. Baker, Clyde Ruby, Preliminary design of jml: a behavioral interface specification language for java, SIGSOFT Software. Eng. Notes 31 (3) (2006) 1–38.
[21] Kanglin Li, Mengqi Wu, Effective Software Test Automation: Developing an Automated Software Testing Tool, SYBEX Inc., Alameda, CA, USA, 2004.
[22] Xiyang Liu, Bin Wang, Hehui Liu, Evolutionary search in the context of object-oriented programs, in: MIC'05: Proceedings of the Sixth Metaheuristics International Conference, 2005.
[23] Sean Luke, ECJ 16: A Java Evolutionary Computation Library, 2007. <http://cs.gmu.edu/~eclab/projects/ecj/>.
[24] Timo Mantere, Jarmo T. Alander, Evolutionary software engineering, a review, Appl. Software Comput. 5 (3) (2005) 315–331.
[25] P. McMinn, Search-based software test data generation: a survey, Software Testing Verif. Reliab. 14 (2) (2004) 105–156.
[26] P. McMinn, M. Holcombe, The State Problem for Evolutionary Testing, 2003.
[27] Zbigniew Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, second extended ed., Springer-Verlag, Inc., New York, NY, USA, 1994.
[28] David J. Montana, Strongly typed genetic programming, Evolut. Comput. 3 (2) (1995) 199–230.
[29] Glenford J. Myers, Corey Sandler, The Art of Software Testing, John Wiley & Sons, 2004.
[30] José Carlos Bregieiro Ribeiro, Search-based test case generation for object-oriented java software using strongly-typed genetic programming, GECCO'08: Proceedings of the 2008 GECCO Conference Companion on Genetic and Evolutionary Computation, vol. 7, ACM, New York, NY, USA, 2008, pp. 1819–1822.
[31] José Carlos Bregieiro Ribeiro, Francisco Fernández de Vega, Mário Zenha Rela, Using dynamic analysis of java bytecode for evolutionary object-oriented unit testing, in: SBRC WTF 2007: Proceedings of the 8th Workshop on Testing and Fault Tolerance at the 25th Brazilian Symposium on Computer Networks and Distributed Systems, Brazilian Computer Society (SBC), 2007, pp. 143–156.
[32] José Carlos Bregieiro Ribeiro, Mário Zenha Rela, Francisco Fernández de Vega, A strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of object-oriented software, in: AST'08: Proceedings of the 3rd International Workshop on Automation of Software Test, ACM, New York, NY, USA, 2008, pp. 85–92.
[33] José Carlos Bregieiro Ribeiro, Mário Zenha-Rela, Francisco Fernández de Vega, An evolutionary approach for performing structural unit-testing on third-party object-oriented java software, in: Nature Inspired Cooperative Strategies for Optimization (NICSO 2007), Studies in Computational Intelligence, vols. 129, Springer, Berlin/Heidelberg, 2008/2007, pp. 379–388.
[34] José Carlos Bregieiro Ribeiro, Mário Alberto Zenha-Rela, Francisco Fernández de Vega, Strongly-typed genetic programming and purity analysis: input domain reduction for evolutionary testing problems, GECCO'08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, vol. 7, ACM, New York, NY, USA, 2008, pp. 1783–1784.
[35] Ramon Sagarna, Andrea Arcuri, Xin Yao, Estimation of distribution algorithms for testing object oriented software, in: Dipti Srinivasan, Lipo Wang (Eds.), IEEE Congress on Evolutionary Computation, 25–28 September 2007, IEEE Computational Intelligence Society, IEEE Press, Singapore pp. 438–444.
[36] A. Salcianu, M. Rinard, A Combined Pointer and Purity Analysis for Java Programs, Technical Report MIT-CSAILTR-949, MIT, May 2004.
[37] Alexandru Salcianu, Martin C. Rinard, Purity and side effect analysis for java programs, in: VMCAI'05: Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science, vol. 3385. Springer, 2005, pp. 199–215.
[38] Arjan Seesing, Hans-Gerhard Gro, A genetic programming approach to automated test generation for object-oriented software, ITSSA 1 (2) (2006) 127–134.
[39] Paolo Tonella, Evolutionary testing of classes, in: ISSTA'04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM Press, New York, NY, USA, 2004, pp. 119–128.
[40] Nigel Tracey, John Clark, John McDermid, Keith Mander, A search-based automated test-data generation framework for safety-critical systems, in: Systems Engineering for Business Process Change: New Directions, 2002, pp. 174–213.
[41] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, Vijay Sundaresan, Soot – a java bytecode optimization framework, in: CASCON'99: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, IBM Press, 1999, p. 13.
[42] A.M.R. Vincenzi, M.E. Delamaro, J.C. Maldonado, W.E. Wong, Establishing structural testing criteria for java bytecode, Software Pract. Exp. 36 (14) (2006) 1513–1541.
[43] S. Wappler, J. Wegener, Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm, in: CEC'06: Proceedings of the 2006 IEEE Congress on Evolutionary Computation, IEEE, 2006, pp. 851–858.
[44] Stefan Wappler, Frank 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, ACM Press, New York, NY, USA, 2005, pp. 1053–1060.
[45] Stefan Wappler, Joachim Wegener, Evolutionary unit testing of object-oriented software using strongly-typed genetic programming, in: GECCO'06: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, ACM Press, New York, NY, USA, 2006, pp. 1925–1932.
[46] John Whaley, Martin Rinard, Compositional pointer and escape analysis for java programs, in: OOPSLA'99: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, New York, NY, USA, 1999, pp. 187–206.
[47] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, K. Karapoulios, Application of genetic algorithms to software testing [application des algorithmes génétiques au test des logiciels], in: Proceedings of the 5th International Conference on Software Engineering, 1992, pp. 625–636.
[48] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, Wolfram Schulte, Method-sequence exploration for automated unit testing of object-oriented programs, in: Proceedings of the Workshop on State-Space Exploration for Automated Testing (SSEAT 2008), July 2008.
[49] Shin Yoo, Mark Harman, Pareto efficient multi-objective test case selection, in: ISSTA'07: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, 2007, pp. 140–150.