Pre-print Version
DOI: http://dx.doi.org/10.1007/978-3-642-12148-7_19

Enabling Object Reuse on Genetic Programming-based Approaches to Object-Oriented Evolutionary Testing

José Carlos Bregieiro Ribeiro[1],
Mário Alberto Zenha-Rela[2], and Francisco Fernández de Vega[3]

[1] Polytechnic Institute of Leiria
Morro do Lena, Alto do Vieiro, Leiria, Portugal
jose.ribeiro@estg.ipleiria.pt
[2] University of Coimbra
CISUC, DEI, 3030-290, Coimbra, Portugal
mzrela@dei.uc.pt
[3] University of Extremadura)
C/ Sta Teresa de Jornet, 38, Mérida, Spain
fcofdez@unex.es

Abstract. Recent research on search-based test data generation for Object-Oriented software has relied heavily on typed Genetic Programming for representing and evolving test data. However, standard typed Genetic Programming approaches do not allow Object Reuse; this paper proposes a novel methodology to overcome this limitation. Object Reuse means that one instance can be passed to multiple methods as an argument, or multiple times to the same method as arguments. In the context of Object-Oriented Evolutionary Testing, it enables the generation of test programs that exercise structures of the software under test that would not be reachable otherwise. Additionally, the experimental studies performed show that the proposed methodology is able to effectively increase the performance of the test data generation process.

1 Introduction

Software testing is the process of exercising an application to detect errors and to verify that it satisfies the specified requirements. It is an expensive process, typically consuming roughly half of the total costs involved in software development; automating test data generation is thus vital to advance the state-of-the-art in software testing.

The application of Evolutionary Algorithms (EAs) to test data generation is often referred to as *Evolutionary Testing (ET)* [1]. The goal of ET is to find a set of test cases that satisfies a certain test criterion. If structural adequacy criteria are employed, 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. Object Reuse is a feature of paramount importance in this context.

Object Reuse (OR) means that one instance can be passed to multiple methods as an argument, or multiple times to the same method as arguments [2]. In the context of Object-Oriented Evolutionary Testing (OOET), it enables the generation of test cases that exercise specific structures of software that would not be reachable otherwise. The `equals` method of Java's `Object` class [3] provides a paradigmatic example. Class `Object` is the root of the Java class hierarchy, and the `equals` method is used to assess if two objects are equivalent; also, several search methods rely on it to verify if an item is present in a collection (e.g., `Vector`'s `indexOf`). However, the `equals` method implements the most discriminating possible equivalence relation on objects: for any non-`null` reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same reference. This means that, in order for the method `equals` to return `true`, the same `Object` reference must be passed as an argument twice – in the place of both the implicit parameter (i.e., the `this` parameter) and the explicit parameters. Also, every class has `Object` as a superclass; this means that every class inherits the `equals` method, and uses it internally for equivalence verification. `Object` subclasses may override `equals` in order to implement a less stringent equivalence relation. Still, it is not mandatory; what's more, recent studies have concluded that implementations of the `equals` methods are often faulty [4].

Recent research on ET has relied heavily on typed Genetic Programming (GP) for representing and evolving test data (e.g., [2, 5, 6]). However, standard GP approaches do not allow node reuse; this paper proposes a novel methodology to overcome this limitation, which involves the definition of novel type of GP nodes – the *At-Nodes* – that "point to" other nodes, thus effectively enabling the creation of edges to nodes that are already part of the tree, and allowing the reuse of sub-trees. The introduction of At-Nodes is performed by means of a custom-made evolutionary operator – the *Object Reuse operator*. This operator acts on an individual by selecting two nodes – the node to be replaced by the At-Node, and the node to be "pointed at" by the At-Node – and by inserting the newly created At-Node into the tree. At-Nodes may be removed from a tree by means of the *Reverse Object Reuse operator* which, in short, searches the tree for At-Nodes, and replaces these nodes with copies of the sub-trees pointed at by the At-Nodes. This particular operator removes the need for the reformulation of other common biology-inspired mechanisms (e.g., Mutation and Crossover [7]). In addition to allowing the search to traverse specific structures, the Object Reuse methodology is able to enhance the performance of the test case generation process: it yields solutions with smaller overall size and lower average structural complexity; and the feasibility of the generated Test Programs is increased as a result of the introduction of a specific heuristic for node selection.

This paper is organised as follows: the next Section starts by providing theoretical background on ET; Section 3 details the proposed Object Reuse methodology; in Section 4, the experimental studies developed in order to validate the approach are described and discussed; related work is contextualised in Section 5; and Section 6 summarises the methodology and emphasises the most relevant contributions.

2 Background and Terminology

ET is an emerging methodology for automatically generating high quality test data for Object-Oriented (OO) software and, in particular, for producing a set of unit-tests that meets a predefined structural adequacy criterion [6]. A unit-test case for OO software consists of an *Method Call Sequence (MCS)*, 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 [5]. Each test case focuses on the execution of one particular public method – the *Method Under Test (MUT)*. MUTs may be represented internally by Control-Flow Graphs (CFGs); the aim of the search will then be that of generating a set of Test Programs that traverse all the MUT's CFG nodes, thus achieving full structural coverage. GP is usually associated with the evolution of tree structures, and is particularly suited for representing and evolving Test Programs, which may be represented by *Method Call Trees (MCTs)*. Non-typed GP approaches are, however, unsuitable for representing OO programs, because any element can be a child node in a parse tree for any other element without having conflicting data types; conversely, with *Strongly-Typed Genetic Programming (STGP)* [8], types are defined a priori in the Function Set and define the constraints involved in MCT construction. This feature enables the initialization process and the various genetic operations to only construct syntactically correct MCTs, thus restraining the search-space to the set of compilable Test Programs.

Test case quality evaluation typically involves instrumenting the MUT, and executing it using the generated Test Programs with the intention of collecting trace information with which to derive coverage metrics [6]. Test case execution requires decoding an individual's *genotype* (i.e., the MCT) into its *phenotype* (i.e., the Test Program); Figures 1a, 1b and 1c exemplify this process; Object Reuse has not been introduced at this point. The MUT is the `indexOf` method of the `Vector` class – which corresponds to the root node of the MCT depicted in Figure 1a. Each node's parameters are provided by its children; the MCS (Figure 1b) corresponds to the linearised MCT, with tree linearisation being performed by means of a depth-first traversal algorithm [2]. Each MCS entry contains an *Method Information Object (MIO)*, which encloses: the method signature data necessary for the Test Program's source code to be assembled (e.g., the method's name and class, the parameter types and the return type); and references to other MIOs providing the parameters (if any) for that method (enumerated between square brackets). The Test Program (Figure 1c) is computed with basis on the MCS and corresponds to a syntactically correct translation of the latter.

Compilable Test Programs may still abort prematurely during execution if a runtime exception is thrown [6]. Test cases can thus be separated in two classes: *feasible* test cases are effectively executed, and terminate with a call to the MUT; *unfeasible* test cases terminate prematurely because a runtime exception is thrown before the call to the MUT is reached, and when this happens it is not possible to observe the structural entities traversed in the MUT because the final instruction is not reached. The Test Program depicted in Figure 1c, for example, is unfeasible because a runtime exception it thrown at instruction 6.

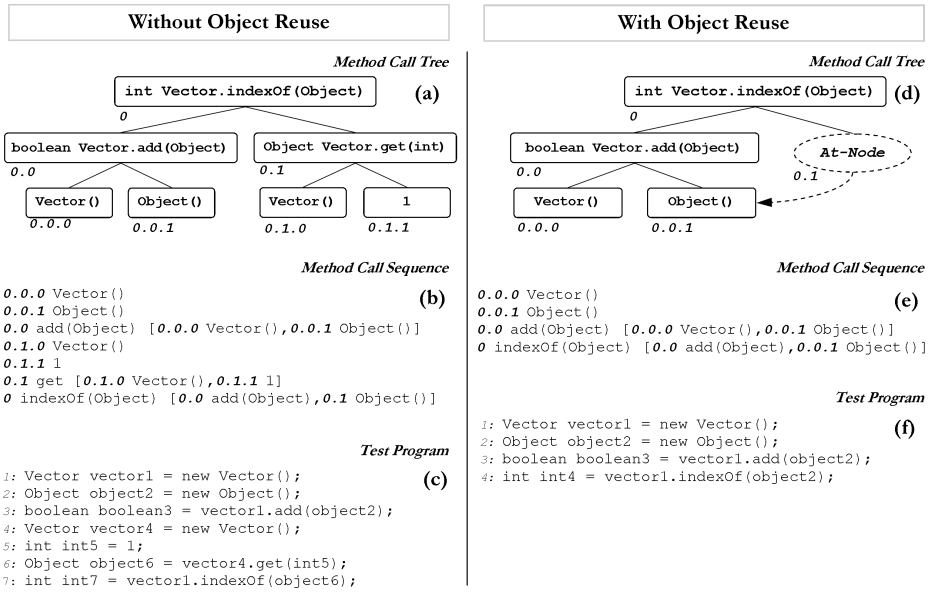

Fig. 1. Example Method Call Trees without and with Object Reuse (a and d), and corresponding Method Call Sequences (b and e) and Test Programs (c and f).

3 An Object Reuse Methodology for OOET

The proposed Object Reuse methodology is based on the introduction of two novel evolutionary operators: the Object Reuse Operator (detailed in the following Subsection), and the Reverse Object Reuse Operator (described in Subsection 3.2). Figure 2 provides an overview of these operators.

3.1 The Object Reuse Operator

The primary goal of the Object Reuse Operator is that of inserting a custom-made type of GP nodes – the At-Nodes – into valid locations of an MCT. The concept of At-Node is, thus, key to the proposed Object Reuse methodology.

At-Nodes At-Nodes are GP nodes that refer to other (standard) GP nodes, thus enabling the reuse of portions of the tree and, specifically, the reuse of the object references returned by the functions corresponding to the reused sub-trees. This is accomplished by having the node pointed at by the At-Node provide the parameter not only to its parent node, but also to the At-Node's parent node; parameter assignment is performed during the MCT's linearisation by means of the process described in Subsection 3.1.4.

Figure 1d contains an example of a possible MCT resulting from the application of the Object Reuse operator to the tree depicted in Figure 1a. The At-Node

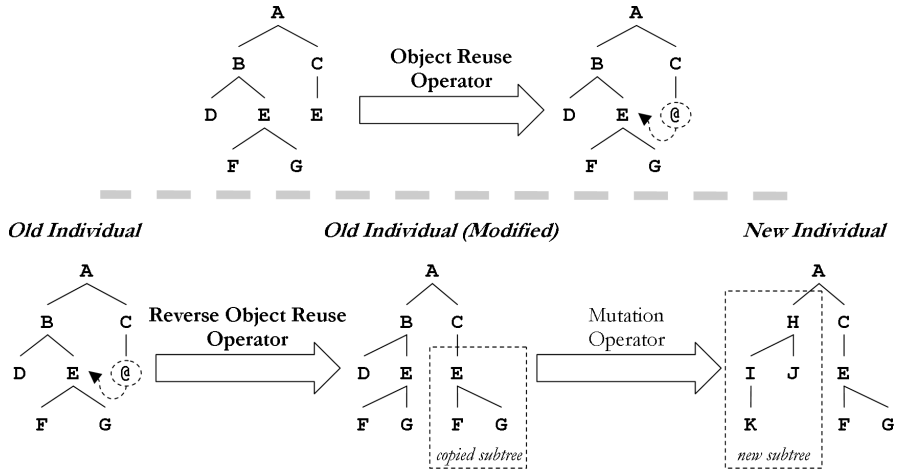

Fig. 2. Object Reuse (*top*) and Reverse Object Reuse (*bottom*) operators overview.

labeled 0.1 replaces the node with the same label existing in the original MCT, whereas node 0.0.1 was selected as the node to be reused. As such, the `Object` instance returned by node 0.0.1 will be used both by its parent (labeled 0.0) and by the At-Node's parent (labeled 0). The MCS and Test Program shown in Figure 1e and 1f mirror this alteration: in the former, the MIO 0.0.1 provides the argument for the explicit parameters of both the 0.0 and 0 MIOs; and in the latter, the reference to the `Object` instance created at instruction 2 is passed to both the `add` and `indexOf` methods (instructions 3 and 4).

The creation of an At-Node for posterior introduction into an MCT requires the Object Reuse operator to select two MCT nodes in the original tree: the *Destination Node* (i.e., the node to which At-Node points to) and the *Replaced Node* (i.e., the root node of the subtree to be truncated and substituted by the At-Node). The first task of the Object Reuse Operator is precisely that of indexing all the valid Replaced-Destination node pairs in an MCT.

Valid Replaced-Destination Node Pairs A Replaced-Destination node pair is valid if:

- both nodes are distinct non-root standard GP nodes;
- the Replaced Node possesses a type that is swap-compatible with the Destination Node (e.g., a node of type `String` is swap-compatible with a node of type `Object`, because `String` is a sub-type of `Object`);
- the sub-tree rooted at the Replaced Node does not contain a node that is pointed at by an existing At-Node. When an At-Node is inserted into the tree, the sub-tree rooted at the Replaced Node is truncated; if it contains a node that is already being reused, this operation will render the tree invalid;
- the Replaced Node is in a position reached by the linearisation algorithm prior to the Destination Node. This validation ensures that the MIOs only

contain parameter references to elements that precede them in the MCS, and that the corresponding Test Program's method calls have their parameters provided by previously created instances.

After all the valid Replaced-Destination node pairs have been indexed, the Object-Reuse Operator proceeds to select one of those pairs.

Replaced-Destination Node Pair Selection The node pair selection procedure is performed differently according to the individual's feasibility:

- if the individual is *feasible*, a Replaced-Destination node pair is chosen at random from the set of valid Replaced-Destination node pairs;
- if the individual is *unfeasible*, the Object Reuse operator attempts to select a valid pair so that the Replaced Node belongs to the non-executed portion of the tree, and the Destination Node belongs to the executed portion of the tree. If such pairs exist, one is selected at random; otherwise, a node pair is chosen at random from the set of all valid Replaced-Destination node pairs.

The heuristic described aims to promote Test Program feasibility by favouring the reuse of feasible portions of the MCT. As was mentioned in Section 2, the Test Program depicted in Figure 1c throws a runtime exception at instruction 6; the feasible portion of this program is thus the sequence of instructions 1 to 5, whereas instructions 6 and 7 form the unfeasible sequence. These sequences can be mapped directly to MCS entries which, in turn, can matched to the corresponding MCT node. The valid Replaced-Destination node pairs which fulfil the premise of the heuristic are, thus, the following: {0.1, 0.0.1}; {0.1.0, 0.0.0}; {0.1.0, 0.0}.

Method Call Tree Linearisation As was referred in Section 2, evaluating the quality of an individual involves its execution which, in turn requires decoding the MCT into the Test Program. However, if At-Nodes exist, a depth-first traversal algorithm does not suffice to linearise a tree; the linearisation algorithm must take into account the fact that certain parameters are supplied not by that node's children, but rather by the node pointed at by an At-Node. The algorithm depicted in Figure 3 describes the polymorphic recursive function utilised to obtain an MCS with basis on an MCT in the presence of At-Nodes.

3.2 The Reverse Object-Reuse Operator

If an MCT contains At-Nodes, some standard evolutionary operators, such as Mutation and Crossover, require the tree to be analysed and possibly modified prior to their application. This necessity is related with the fact that these operators replace subtrees in the original individual by newly created trees (in the case of the former) or by a copy of an another individual's subtree (in the case

```
Data: Method Call Tree
Result: Method Call Sequence
Global Variables:
Current Node ← Root Node;
isDestinationNode ← false;
Previous MIO ← null;
MCS ← empty sequence;
begin Function linearizeMCT(Current Node, isDestinationNode)
    if Current Node ≠ Root Node and isDestinationNode = false then
        Previous MIO ← get MIO from from Parent Node of Current Node;
    if Current Node is an instance of At-Node then
        Destination Node ← get Destination Node from At-Node;
        call linearizeMCT(Destination Node, true);
    else if Current Node is an instance of Standard Node then
        Current MIO ← get MIO from Current Node;
        if Previous MIO ≠ null then
            add Current MIO to Parameter Providers List of Previous MIO;
        if isDestinationNode = false then
            Child Nodes List ← get Child Nodes List from Current Node;
            foreach Child Node in Child Nodes List do
                call linearizeMCT(Child Node, false);
            add Current MIO to MCS;
end
```

Fig. 3. Algorithm for Method Call Tree linearisation in the presence of At-Nodes.

of the latter); however, if the subtrees truncated in the original individual contain Destination Nodes their elimination will render the MCT inconsistent and disable the possibility of translating it to a syntactically correct Test Program.

The Reverse Object Reuse operator's task is precisely that of pre-processing the individuals to be provided to other well-established operators, thus avoiding their reformulation. It starts by indexing all the At-Nodes in an MCT, and then proceeds to replace each At-Node with a clone copy of the sub-tree rooted at its Destination node. The resulting MCT can then be provided to another evolutionary operator. That is, the Reverse Object Reuse operator's purpose it that of being the first component of a breeding pipeline and acting as a source of individuals; it selects individuals directly from the population (e.g., using Tournament Selection [7]), and provides the (possibly) modified individual to the operator at the end of the breeding pipeline. This process is schematised in Figure 2.

4 Experimental Studies

The Object Reuse methodology described was embedded into *eCrash*, a tool for the ET of OO Java software, with the objective of assessing its impact on both the efficiency and the effectiveness of the evolutionary search; a thorough description of the *eCrash* tool can be found in [6].

Table 1. Sources of Individuals.

With Object Reuse	Without Object Reuse
Object Reuse Op. (25%)	Mutation Op. (34%)
Reverse Object Reuse Op. / Mutation Op. (25%)	Crossover Op. (33%)
Reverse Object Reuse Op. / Crossover Op. (25%)	Reproduction Op. (33%)
Reproduction Op. (25%)	

4.1 Targets and Configuration

The Java `TreeMap` (an implementation of Red-Black Tree) and `Vector` classes of JDK 1.4.2 [3] were used as test objects. Their selection is supported by the fact that they are container classes, which are a typical benchmark in software testing of OO programs; Red-Black Trees, in particular, have been empirically shown to be the most difficult to test among containers programs [9]. As MUTs, the 5 most complex public methods (in terms of their Cyclomatic Complexity Number (CCN) [10]) of each class were selected. For each MUT, 2 sets of 20 runs were executed. The Object Reuse and Reverse Object Reuse operators were included in the first, and excluded from the second; Table 1 depicts the sources of individuals selected for each set of runs. The decision of selecting equal probabilities for the Mutation, Crossover and Reproduction operators is supported by previous experiments described in [6]. The remaining evolutionary parameters were common to both sets, and were defined as follows: a single population of 25 individuals was used; the search stopped if an ideal individual was found or after 200 generations; the selection method was Tournament Selection [7] with size 2; the tree builder algorithm was PTC2 [11], with the minimum and maximum tree depths being defined as 4 and 14. The *eCrash* tool was configured in accordance to the setup proposed in [6]. An additional set of 20 runs, in which all individuals were randomly generated using the PCT2 algorithm (with minimum and maximum tree depths of 4 and 14), was performed for comparison purposes; because no evolutionary operators were used, Object Reuse was absent from the process. This random search stopped if an ideal individual was found or after the generation of 5000 individuals. The results were included in Table 2.

4.2 Results and Discussion

The results depicted in Table 2 show that, for both classes, a higher percentage of runs attaining full structural coverage was achieved when including the Object Reuse operator as a source (the only exception being the `putAll(Map)` method of the `TreeMap` class). An average success rate of 62% was achieved with Object Reuse, whereas only 42.5% of the runs were successful without it. What's more, the impossibility of attaining full structural coverage for some of the methods tested is symptomatic of the way in which the lack of the Object Reuse functionality can hinder the evolutionary search. In fact, several search methods – in particular, `Vector`'s `indexOf` and `lastIndexOf`; and `TreeMap`'s `put`, `remove` and `get` – rely on `equals` to verify if an item is contained in a

Table 2. Percentage of runs attaining full coverage (*%f*) and average number of individuals evaluated per run (*#i*); for the *With OR*, *Without OR* and *Random* runs; for the 5 public methods with the highest CCN of the `TreeMap` and `Vector` classes.

MUT	CCN	With OR %f	With OR #i	W/out OR %f	W/out OR #i	Random %f	Random #i
`TreeMap`							
put(Object,Object)	10	10%	4563	0%	5000	0%	5000
putAll(Map)	10	85%	1389	95%	1154	75%	2385
remove(Object)	3	25%	4119	0%	5000	0%	5000
containsValue(Object)	3	100%	501	100%	548	100%	628
get(Object)	2	25%	4000	0%	5000	0%	5000
`Vector`							
lastIndexOf(Object,int)	10	60%	3203	0%	5000	0%	5000
indexOf(Object,int)	8	40%	4243	0%	5000	0%	5000
removeElementAt(int)	6	85%	1829	75%	2258	70%	2948
addAll(int,Collection)	5	100%	871	95%	1130	80%	1668
remove(int)	4	90%	1904	80%	2545	80%	2815

collection. This means that if instances are not reused, the search for non-`null` arguments of type `Object` will fail. A commonly used workaround (e.g., [6]) is that of including substitute classes into the test cluster, which extend `Object` and override `equals` with a less stringent implementation; this approach, however, does not suffice for the following reasons. Firstly, certain test scenarios may specifically involve using classes that do not override `equals` or the `Object` class itself. Secondly, the decision on which additional classes to include into the test cluster is problem specific and human dependant; to the best of our knowledge, no systematic strategy has been proposed to automate this task. Thirdly, the inclusion of redundant classes into the test cluster will enlarge the search space and will thus have negative consequences on the efficiency of the search [6].

The graphs depicted in Figure 4 provide an overview of the way in which the runs evolved, and on how the Object Reuse methodology affects the test case generation process in terms of coverage, Test Program size and feasibility. The runs in which Object Reuse was employed yield solutions with shorter MCS length (a difference of 20.3%, on average, for `TreeMap`, and 12% for `Vector`). Also, feasibility is significantly promoted, with an average increase of 4% for both the `TreeMap` and `Vector` classes. These observations show that the proposed methodology is not only able to enhance the effectiveness of the test case generation process, but also its efficiency. Firstly, it yields solutions with smaller overall size and lower average structural complexity, thus contributing positively to the area of MCS minimisation. Simpler and shorter test programs do not only reduce the computational effort involved in compilation and execution; they also ease the (mostly human-dependant) task of defining a mechanism for checking that the output of a program is correct given some input (i.e., an oracle). Secondly, the application of the Replaced-Destination Node Pair Selection heuristic is able to increase the average feasibility of the generated Test Programs. Because only feasible Test Programs are concluded with a call to the MUT, a higher level of feasibility will increase the performance of the test case generation process [6].

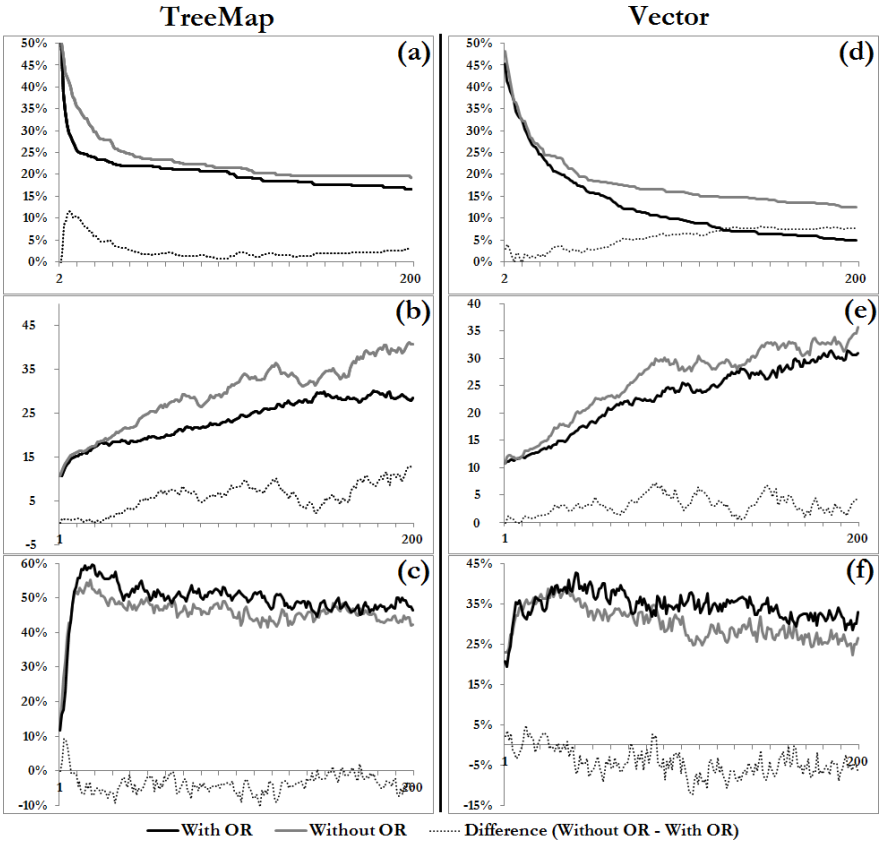

Fig. 4. Average percentage of CFG nodes left to be covered per generation *(a and d)*, average MCS length per generation *(b and e)*, and average percentage of feasible individuals per generation *(c and f)*; for the *With OR* and *Without OR* runs; for the 5 public methods with the highest CCN of the `TreeMap` and `Vector` classes.

5 Related Work

The proposed approach to Object Reuse has some similarities with Koza's work on Automatically Defined Functions (ADFs) [12]. ADFs enable GP to solve problems by decomposing them into subproblems, solving the subproblems, and assembling the partial solutions into a solution to the overall problem; an individual's genotype usually consists of a forest of trees (or functions), which are then called repeatedly from the main tree. Therefore, ADFs do enable function reuse, as the possibility of selecting and calling the same function multiple times exists. However, functions in OO languages typically return object references, and each individual function call – even to the same function – returns a distinct reference. As such, ADFs do not enable Object Reuse, as the possibility of utilising the object reference returned by a single function call more than

once is not possible. The Object Reuse methodology described also shares some characteristics with graph-based approaches to GP, such as Parallel Distributed GP [13] and Cartesian GP [14], as it also involves loosening the interpretation of the edges of an MCT thus effectively transforming it into a graph. However, to the best of the authors' knowledge, there has been no research on applying any of the above approaches to the generation of OO software and, in particular, to OOET; conversely, STGP has been extended to support type inheritance and polymorphism [15, 16], and extensive work has been performed on applying it to OOET (e.g., [2, 5, 6]). As such, we believe that the proposed methodology constitutes a significant contribution to the OOET area. The only previous approach to Object Reuse known to the authors does not involve a loosening of the interpretation of the edges of an MCT, but rather a loosening of the parameter object assignments during tree linearisation. In [2], Wappler proposes employing an Object Pool that stores references to all the objects created during a Test Program execution; this pool is consulted if a parameter object is required for a method call, and a parameter object selector component selects the instance to be used among all available instances of the required type. There are, however, some drawbacks to the Object Pool approach to Object Reuse. Firstly, all the objects, even those that are not used, must be created and stored in the Object Pool, which will obviously increase the length and complexity of Test Programs. Also, and perhaps most importantly, changing parameter object assignments during tree linearisation will result in a discrepancy between the individual's hereditary information (i.e., its genotype) and its actual observed properties (i.e., its phenotype); in other words, the Test Program might not directly correspond to the MCT. Considering that an individual's evaluation is performed at the phenotype level, the Test Program must be an exact translation of the MCT in order for the fitness to be accurately assessed and reflect an individual's quality.

6 Conclusions

The goal of OOET is to find a set of Test Programs that satisfies a predefined test criterion. Object Reuse means that a single object instance can be passed as an argument multiples times to one or more methods; if structural adequacy criteria are employed it is a feature of the utmost importance, as it enables the generation of test cases that exercise structures of software that would not be reachable otherwise. The main contribution of this work is that of proposing a methodology for enabling Object Reuse on typed GP-based approaches to OOET, which involves the definition of novel type of GP nodes (the At-Nodes) that "point to" other nodes, thus permitting the reuse of portions of the tree and, specifically, the reuse of the object references returned by the functions corresponding to the reused sub-trees. Additionally, At-Nodes may be removed from a tree; this functionality allows avoiding the reformulation of other well-established evolutionary operators, such as Mutation and Crossover. Besides enhancing the effectiveness of the search, the experimental studies performed show that the proposed methodology improves the performance of the test case generation process:

it yields solutions with smaller overall size and lower structural complexity, and it is able to increase the feasibility of Test Programs.

Acknowledgements The third author acknowledges the support of the following projects: TIN2007-68083-C02-01 (Nohnes project, Spanish Ministry of Science and Education); PDT-08A09 and GRU-09105 (Junta de Extremadura).

References

1. Tonella, P.: Evolutionary testing of classes. In: ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, New York, NY, USA, ACM Press (2004) 119–128
2. Wappler, S.: Automatic Generation Of Object-Oriented Unit Tests Using Genetic Programming. PhD thesis, Technischen Universitat Berlin (12 2007)
3. Sun Microsystems: JavaTM 2 Platform, Standard Edition, v 1.4.2, API Specification. (2003) http://java.sun.com/j2se/1.4.2/docs/api/.
4. Vaziri, M., Tip, F., Fink, S., Dolby, J.: Declarative object identity using relation types. In Ernst, E., ed.: ECOOP. Volume 4609 of Lecture Notes in Computer Science., Springer (2007) 54–78
5. Wappler, S., Wegener, J.: 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, New York, NY, USA, ACM Press (2006) 1925–1932
6. Ribeiro, J.C.B., Zenha-Rela, M.A., Fernández de Vega, F.: Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software. Inf. Softw. Technol. **51**(11) (2009) 1534–1548
7. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). The MIT Press (12 1992)
8. Montana, D.J.: Strongly typed genetic programming. Evolutionary Computation **3**(2) (1995) 199–230
9. Arcuri, A.: Insight knowledge in search based software testing. In: GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, New York, NY, USA, ACM (2009) 1649–1656
10. McCabe, T.J.: A complexity measure. IEEE Trans. Software Eng. **2**(4) (1976) 308–320
11. Luke, S.: Two fast tree-creation algorithms for genetic programming. IEEE Transactions on Evolutionary Computation **4**(3) (September 2000) 274–283
12. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. The MIT Press, Cambridge, Massachusetts (1994)
13. Poli, R.: Evolution of graph-like programs with parallel distributed genetic programming. In Bäck, T., ed.: ICGA, Morgan Kaufmann (1997) 346–353
14. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Proceedings of the European Conference on Genetic Programming, London, UK, Springer-Verlag (2000) 121–132
15. Haynes, T.D., Schoenefeld, D.A., Wainwright, R.L.: Type inheritance in strongly typed genetic programming. In Angeline, P.J., Kinnear, Jr., K.E., eds.: Advances in Genetic Programming 2. MIT Press, Cambridge, MA, USA (1996) 359–376
16. Yu, T.: Polymorphism and genetic programming. In: EuroGP '01: Proceedings of the 4th European Conference on Genetic Programming, London, UK, Springer-Verlag (2001) 218–233