Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1053–1062
DOI: 10.15439/2018F51
ISSN 2300-5963 ACSIS, Vol. 15

Deep Object Comparison for Interface-based Regression Testing of Software Components

Tomas Potuzak
Department of Computer Science and Engineering/ NTIS – New Technologies for the Information Society, European Center of Excellence, Faculty of Applied Sciences, University of West Bohemia
Univerzitni 8, 30614 Plzen, Czech Republic
Email: tpotuzak@kiv.zcu.cz

Richard Lipka
NTIS – New Technologies for the Information Society/Department of Computer Science and Engineering, European Center of Excellence, Faculty of Applied Sciences, University of West Bohemia
Univerzitni 8, 30614 Plzen, Czech Republic
Email: lipka@kiv.zcu.cz

***Abstract*—In this paper, we describe the deep object comparison (DOC) algorithm, which is used for comparison of general objects in Java programming language based on their internal structures and values of primitive attributes. The DOC algorithm was designed to be utilized in our interface-based regression testing of software components, which enables to uncover subtle changes of the behavior of a component-based application under test with a newly installed version of a software component in comparison to its behavior with an old version of this component.**

I. Introduction

THE component-based software development is a part of software engineering for nearly two decades. It utilizes isolated reusable software parts called *software components*, which provide and/or require functionalities called *services*. The services are accessible using public interfaces of the components and the components are expected to interact solely using these interfaces. Specific details depend on the utilized *component model*, which defines the behavior, features, and interactions of software components and is implemented by a *component framework* [1].

Regardless the utilized component model, a common situation using the component-based software development is that a component can be used in different applications and an application consists of multiple components, which can originate by different manufacturers [1]. This underlines the necessity for the testing, not only of the individual components, but of the entire component-based application as well.

Additionally, many components exist in several versions, which can mutually differ by the internal behavior (i.e., there are different computations), by the external behavior (i.e., different interactions with other components), or by the public interface (i.e., different required and/or provided services). In theory, the change of the component's internal behavior should not affect its external behavior and therefore should not affect the behavior of the entire component-based application. Nevertheless, in reality, an unwanted error can be introduced into the new version of the component, a side effect of a method invocation can be added or removed, a computation can be prolonged leading to a time-out to expire, and so on. So, when installing a new version of a component to a functional component-based application, adequate regression testing is desirable even when there are no apparent external changes of the new version of the component in comparison to the old version [2].

During our previous research, we developed an approach for interface-based regression testing of software components, whose source code is not available (e.g., third-party software components). The approach is tailored for the situation when there is a new version of a component installed in a component-based application and we want to check if it exhibits the same behavior within the application as its old version [2].

The experimental implementation of the approach, which was described in [2] in detail, was designed for the OSGi [3] component model for Java programming language, but the ideas behind it can be used for other component models and programming languages as well. The overall process starts with the analysis of the services and their methods of the software components of the entire component-based applica-tion under test. For each method of each service of each component, a set of invocations is generated. Then, the invocations are performed in an iterative phase. In each iteration, all invocations are performed and their conse-quences are being observed and stored. New consequences of the same invocations can emerge, because the inner states of the components can change between iterations due to the invocation of the methods. Besides the consequences, new invocations can emerge during this phase as consequences of different invocations. Both the new consequences and new invocations are stored only if they are not already stored. This requires comparison of the already stored items with the newly created items. The process stops when no new consequences are created. The result is a testing scenario

This work was supported by Ministry of Education, Youth and Sports of the Czech Republic, project PUNTIS (LO1506) under the program NPU I and by European structural and investment funds (ESIF), project CZ.02.1.01/0.0/0.0/17_048/0007267.

with actions (i.e., invocations) and their consequences, which can be saved to a file. If this process is performed prior and after the installation of a new version of a component to the component-based application under test, the detailed comparison of these two saved scenarios can uncover changes in the behavior of the application caused by the new version of the component [2].

In our experimental implementation, we used the standard `equals()` method for the comparison of the objects associated with the consequences and the invocations (e.g., return values, values of the parameters). Although this can work in many cases, it cannot be used universally. Some objects do not implement the `equals()` method, which leaves them with the default implementation corresponding to the identity (only the same objects are considered equal). Even if the `equals()` method is implemented in the object, it can be implemented incorrectly, as pointed out in [4], [5], or [6]. And even if it is implemented correctly, it does not mean that it considers all primitive values of the object recursively [7]. So, it is possible that, although the `equals()` method returns `true` for a pair of objects, their internal structures can be different and/or contain some different primitive values. These subtle differences can be important for our approach, since they could mean different behavior, which we want to detect.

In order to mitigate this problem, in this paper, we describe the deep object comparison, which will replace the utilization of the `equals()` method. The deep object comparison enables to compare two objects based on the "shape" of their internal structures and all the corresponding primitive values. So, no changes in the objects are missed. This deep object comparison is suited to be used both during the generation of the scenario and during the comparison of two scenarios of our interface-based approach for regression testing of software components. The algorithm was implemented within our Interface Analysis Tool (InAnT). The deep object comparison was first tested as a stand-alone algorithm, before it will be incorporated into our approach. The description of the deep object comparison algorithm along with the description of the performed tests is the main contribution of this paper.

The paper is structured as follows. The interface-based regression testing of software components is briefly described in Section II. Related work is discussed in Section III. In Section IV, the deep object comparison is described in detail. The performed tests and results are described in Section V and the paper is concluded and the future work is discussed in Section VI.

II. Interface-based Regression Testing of Components

As it was mentioned in Section I, the interface-based regression testing of software components is designed to uncover any changes in a component-based application's behavior after the installation of a new version of a component [2]. The changes are detected during comparison of the testing scenario generated and stored from the application with the old version of the component and the scenario generated and stored from the application with the new version of the component [2].

A. Generation of the Testing Scenario

Our approach assumes that the entire component-based application is under test, because the components within it interact with each other. Their interactions are observed during the generation of the scenario in order to uncover the behavior of the particular components [2].

First step in the generation of the testing scenario is the determination of all methods of all services of the components of the application under test. This can be done by any method capable to retrieve complete method signature. We use standard OSGi methods and Java reflection [8] for this purpose in our experimental implementation [2]. The components, their services, and their method are inserted into a tree data structure, which forms the basis of the testing scenario (see Fig. 1a).

For each method of this structure, an initial set of invocations is generated and added into the structure. Each invocation contains a unique combination of values for all the parameters of the method.

The invocations are then successively performed (i.e., the methods are invoked with the parameters stored in the invocations in the tree data structure) in the iterative phase, one at a time, and the consequences of each invocation are observed (i.e., what happened when the method was invoked). The possible consequences are a thrown exception, a return value, a value change in "out" parameters of the method, a subsequent invocation of a service method of ano-

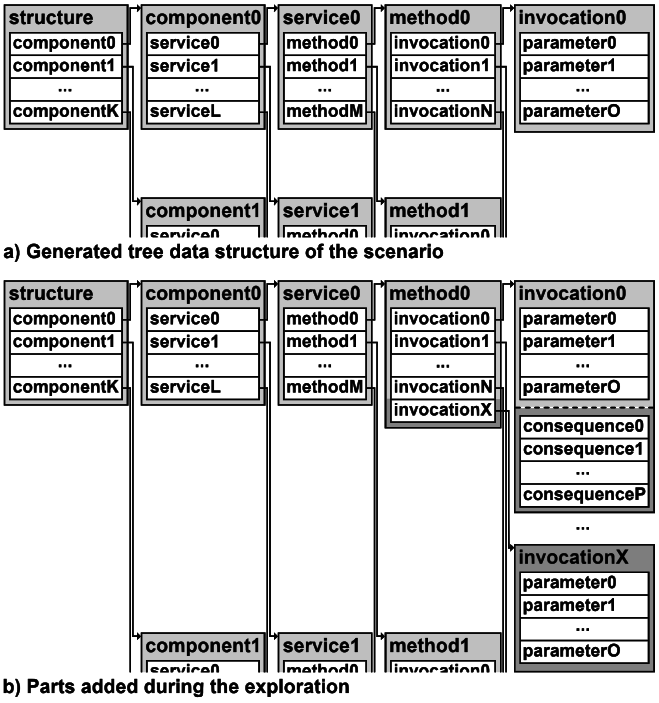

Fig. 1 The tree structure of the scenario

ther component, and a change of the inner state of the component. The last consequence differs from the others, since it is not easily observable from outside. So, it is not considered by our approach. There can be several consequences per method invocation. All the observed consequences are added to the tree data structure to the invocation, which caused them (see Fig. 1b), but only if they are not already present. Each type of consequence contains its type and type-dependent data (e.g., the return value, the instance of an exception, the changed value of an "out" parameter, etc.) [2]. Based on the type and the data, the consequences can be mutually compared, which is necessary for the determination, whether a new consequence is already present or not.

The most important consequences are the subsequent invocations. Each subsequent invocation is defined by the method, which it is invoking, and by the unique combination of its parameter values. When this consequence is observed, it is added to the tree data structure (if not already present) similarly to other types of consequences. Moreover, the invocation that the consequence represents is added to the invocations of the corresponding method into the tree data structure (again, only if not already present). These invocations are valuable, since their parameter values are genuine, originating in the internal logic of the component, which invoked the method [2].

The invocations contained in the tree data structure are performed several times in the iterative phase in order to exploit the subsequent invocations. The subsequent invocations generated in nth iteration can be performed in $(n + 1)$th iteration and their consequences can be thus observed. The iterative phase is stopped when no new consequences are generated in the current iteration. At this point, the testing scenario represented by the tree data structure is complete (see Fig. 1b). All invocations and consequences contain a number representing the iteration, in which they were added to the structure (starting with 1). The initial invocations created prior the iterative phase have this number set to 0. The generated scenario is saved to a XML file [2].

B. Comparison of the Testing Scenario

When a new version of a component is installed into the component-based application under test, the process described in Section II.A is repeated and a new scenario is created. The saved scenario is then loaded from the XML file and both tree data structures are compared. The comparison is performed on each level of the structures, starting from the component level [2].

On each level, it is checked, whether there are corresponding items (i.e., components, services, methods, invocations, consequences) in both tree data structures. If so, their subtree is expanded and the comparison continues on the lower level. If not so, the difference (item is missing in one or second tree data structure) is reported, this item is not expanded and its lower levels are not considered further [2]. The example of the result of the comparison is depicted in Fig. 2.

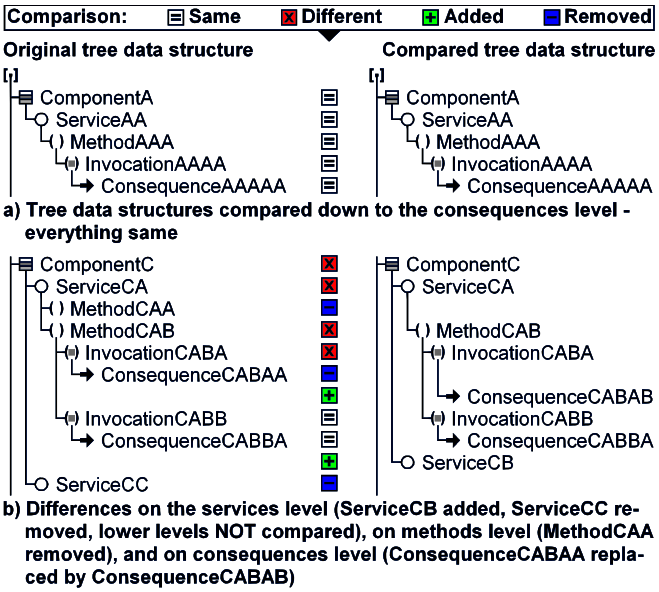

Fig. 2 Result of the comparison of two scenarios (tree data structures)

The most important differences are on the invocations and invocation consequences levels. These differences mean different behavior of the application under test with the old and the new version of the component. Differences on the methods or services levels imply that there are changes in the public interface of the component. Our approach of course detects these changes, but, unlike the changes in the behavior, these changes can be detected by other means as well, such as advanced static analysis methods (e.g., see [9]) [2].

C. Object Comparison Issues

Both during the generation of the scenario and during the comparison of two scenarios, we need to compare general objects, which is problematic.

During the generation of the scenario, the comparison of general objects is necessary in the iterative phase when new consequences and invocations are generated. They are added to the tree data structure only if they are not already contained, requiring their comparison to other consequences and invocations. It should be noted that due to the tree nature of the data structure, a newly generated consequence is compared only to the consequences of the corresponding invocation. Similarly, a newly generated invocation is compared only to the invocations of the corresponding method. So, the number of comparison is limited, it is not necessary to compare the consequence or invocation to all consequences or invocations.

Nevertheless, the comparison of two consequences lies in the comparison of their type and, if the type is the same, in the comparison of the type-related data. If the type of both compared consequences is the return value, then the associated return values are compared. The comparison is

trivial if the return values are of primitive types, but ambiguous if they are objects. In the experimental implementation of our approach, we use the standard `equals()` method for objects, which are not `null`. This can work in many cases, but cannot be used universally.

For example, some objects do not implement the `equals()` method, which leaves them with the default implementation corresponding to the identity (only the same objects are considered equal). An example of such object is in Fig. 3a. The `Point3D` class represents a point in space, but does not override the `equals()` method, leaving it with its default implementation (from the `Object` class). When a method returns a new instance of the `Point3D` in every invocation (see Fig. 3b), the comparison of this instance to another instance using `equals()` will always return `false`, even with the same values of their corresponding coordinates (see Fig. 3c). If an invocation of the method depicted in Fig. 3b were performed repeatedly during the iterative phase, its return value consequence would always seem different, because the return values would not be identical (based on the `equals()` method), although they would contain the same values of their corresponding coordinates. So, each newly generated consequence would be added to the tree data structure in each iteration. The iterative phase would not stop until an out-of-memory exception would occur. This problem can be mitigated (not solved) by introduction of the maximal number of iterations, but it is clear that this is not the intended behavior.

Moreover, there are further issues. Even if the `equals()` method is implemented in the object, it can be implemented

```
class Point3D {
  public int x;
  public int y;
  public int z;

  public Point3D(int x, int y, int z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }
}
```

a) A class representing a point in space without overridden `equals()` method

```
...
public Point3D asPoint(int x, int y, int z) {
  return new Point3D(x, y, z);
}
...
```

b) A method returning a new instance of the `Point3D` class in every invocation

```
...
Point3D p1 = asPoint(1, 2, 3);
Point3D p2 = asPoint(1, 2, 3);
boolean comparison = p1 == p2; //comparison false
...
```

c) Two results of the method with the same primitive values compared

Fig. 3 Example of a class without overridden `equals()` method and of following problems

incorrectly, as pointed out in [4], [5], or [6]. And even if it is implemented correctly, it does not mean that it considers all primitive values of the object recursively [7]. So, it is possible that, although the `equals()` method returns `true` for a pair of objects, their internal structures can be different and/or contain some different primitive values. These subtle differences would not be detected using the `equals()` method.

The described problem is not limited to the return value consequences. The same problem is with the comparison of consequences representing a change in the "out" parameters of a method and with the comparison of invocations. Each invocation contains the combination of parameter values of a method and these values, which can be general objects, are compared during the comparison of the invocations.

During the comparison of two scenarios, the comparison of general objects is needed for the comparison of invocations and consequences, similarly to the generation of the scenario. Comparison of methods and higher levels of the tree data structure are based on data types and names, not general objects. The problem with the comparison of general objects is more pronounced here, though. The reason is that at least one of the compared scenarios is loaded from a XML file. In order to utilize the `equals()` method for the comparison, it would be necessary to recreate all the objects during the loading of the scenario. This would necessitate full-scale serialization of general objects during the saving of the scenario to the file. Hence, in the experimental implementation of our approach, the general objects contained in the invocations and consequences were compared only based on their classes and `null` values.

More specifically, the information stored to the XML file for an object was its real class or `null`. Hence, during the comparison of two scenarios, it was only checked, whether both compared objects are `null` or whether both compared objects are of the same class. In these two cases, the objects were considered equal. The exception was the instances of the `String` class, which were compared using their content. The reason is that the `String` instances can be easily saved and loaded to/from a file. It is clear that this significantly reduces abilities of our interface-based regression testing of software components. The entire approach works correctly and is able to detect changes in behavior of the application under test (see [2]). However, many subtle differences in the compared scenarios can remain hidden, because the information is lost during the saving of the scenario to a file. So, some changes in behavior of the application under test could remain undetected.

In order to solve all the described problems, we designed the deep object comparison (DOC) described in Section IV in detail. The DOC will be incorporated into our interface-based regression testing of software components where it will replace the `equals()` method during the generation of the scenario and the class-based comparison during the comparison of two scenarios.

III. Related Work

The issue of object equality in object-oriented languages is discussed in scientific literature mainly in relation to memory optimization and to object equality implementation. Both branches are discussed in following subsections. Although none of these branches is related to software testing, the algorithms described in the discussed papers solve problems similar to our deep object comparison.

We focused mainly on the papers regarding the Java programming language, since our current implementation is written in this language. However, the principles can be used in similar languages (e.g., C#) as well.

A. Memory Optimization

There are several papers focused on the optimization of memory management in languages, which utilize garbage collection (i.e., automatic disposal of objects, which are no longer in use by the program). The main idea behind these works is that there are a number of equivalent objects in the memory during the execution of a program, which can be replaced by a single instance while preserving the same behavior of the entire program (see [7], [10], [11], [12], etc.).

In [7], a tool enabling detection of equivalent objects in a Java application, which can be replaced by a single instance, is described. The investigated application is instrumented and, during its execution, all relevant heap activity is recorded. After the execution, the post-mortem analysis is performed. The objects of the application are separated into equivalence classes. Each equivalence class can be replaced by a single instance. No automatic optimization is performed. The tool only uncovers and reports the sites (i.e., positions in source code) of the program with the potential for an optimization by replacing more equivalent objects with a single instance [7]. In order to determine, whether an object can be replaced by another object without affecting the behavior of the application, it is necessary to compare these objects thoroughly. It is pointed out that a full comparison requires checking of two potentially cyclic labeled graphs for isomorphism. As a large number of comparisons is required in this approach, a hash value is calculated for each object, which is then used for faster comparison of the objects [7].

Similar approach is described in [10], although it is intended for a different programming language (Pharo). Again, the relevant heap activity is recorded during a run of the application under test and the post-mortem analysis is performed after the run. The main difference is that, in [10], the objects are divided into several types and only certain types of objects, which are susceptible to redundancy (e.g., instances of the `String` or `Point` classes), are considered as optimization opportunities. Directed vertex and edge labeled graph is used for the representation of the internal structure of objects for the comparison purposes. Based on it, a hash value is calculated for each considered object to speed up its comparison to other objects [10].

In [11], the conditions, which an object must satisfy to be considered for caching, are discussed. The paper is focused on immutability of objects, which is in some form often required by caching and similar techniques (including techniques described in [7] and [10]). It is pointed out that the border between construction and mutation of an object is not always clear [11].

In [13], an advanced method for the reduction of the duplication of strings is described. The comparison of objects is straightforward in this case, as the compared objects are instances of the `String` class.

In [12], a similar yet different technique to [7] and [10] is described. Similarly to the [7] and [10], the technique searches the sites (i.e., positions in source code) in an application, which have the potential to be optimized by reusing existing objects or data structures [12]. Unlike [7] and [10], the technique is focused not only on finding equivalent objects, but on finding reusable instances as well. The idea is that it is not necessary to create a new (possibly internally complex) instance when there is another instance of the same class available, which is no longer in use. In that case, it is only necessary to change settings of its primitive values, but it is not necessary to create new object and its entire internal structure. Moreover, since the unused object is reused, its garbage collection is saved. Thus, the techniques tend to utilize objects, which are relatively short-lived. So, unlike [7], [10], which are focused mainly on the reduction of the memory consumption, [12] is focused also on the reduction of computation time. For the representation of the internal structure of the objects, modified balanced parenthesis algorithm is used [12].

In contrast to [12], the technique described in [14] is focused on the long-lived objects, which are candidates for caching. For the speedup of the object comparison, a form of hash value called “fingerprint” of the object is used. In standard classes from the `java.lang` package, the `equals()` method is used [14].

In [15], the performance of the hash-consing (i.e., utilization of a global cache of objects) is discussed. The comparison of objects is based on the `equals()` method even when this method does not consider all attributes of the object. The authors also define weak immutability of an object based only on the values of the attributes, which are considered by the `equals()` method. Again, a hash value is used for the description of the objects [15].

B. Object Equality Implementation

The other group of scientific papers regarding the object equality is focused on the implementation of the `equals()` method. In majority of this papers, the required features of the `equals()` method are cited – the *reflexivity*, *symmetricity*, and *transitivity* [4]. In some works, it is pointed out that many textbooks contain flawed implementations of the `equals()` method (e.g., [4], [6]).

In [4], a checker of the `equals()` methods is described. The checker is focused on the features of the `equals()` methods under test and reports any violation of these features. However, it is not focused on comparison of the entire internal structure of complex objects [4]. In [6], the right design of the `equals()` method using design patterns is discussed.

A generator of the `equals()` methods for complex objects is described in [16]. In this work, the objects are compared on per-field basis. There are several "depths" of equality defined. Depth-0 corresponds to the referential equality, meaning that the objects are equal if both are the same object (corresponding to the comparison operator "=="). Depth-1 (shallow equality) means that, for all corresponding fields of two objects, the referential equality holds. The deep equality then means that, for all corresponding fields of two objects, the deep equality holds [16].

IV. Deep Object Comparison

As it was mentioned in Section II.C, the deep object comparison (DOC) is designed for our interface-based regression testing of software components. It will replace the comparison of general objects using the `equals()` method during the generation of the testing scenario and the comparison of general objects using their classes and `null` values during the comparison of two scenarios. Hence, the DOC algorithm has two phases – the forming of the graph representation of the object and the comparison of two graphs of the compared objects. The main advantage of this approach is that the graph representation of the object can be easily saved to and loaded from a file. So, no information will be lost during the saving of the scenario.

A. Object Equality

As arises from Section III, there are various views on the equality of two objects. Nevertheless, since our goal is to uncover any difference of the compared objects, however subtle, we have to adopt the view of the deep equality described in [16]. Similar definitions are described also in [7], [10], or [11]. Nevertheless, all the definitions of the equality described in these works are recursive. Since the DOC algorithm creates a graph representing the internal structure of the object, our definition is based on this graph. It is not recursive, but expresses similar conditions as the definition of the deep equality described in [16].

Suppose there are two compared objects `a` and `b` and the graph representations of their internal structures were created (see Section IV.B). Objects `a` and `b` are considered equal if and only if they are of the same class, their graph representations are isomorphic (i.e., have the same "shape"), and all values of the corresponding primitive attributes in the corresponding vertices of the graphs are of the same type and equal. Fig. 4 shows several examples of pairs of objects, which are considered equal or different.

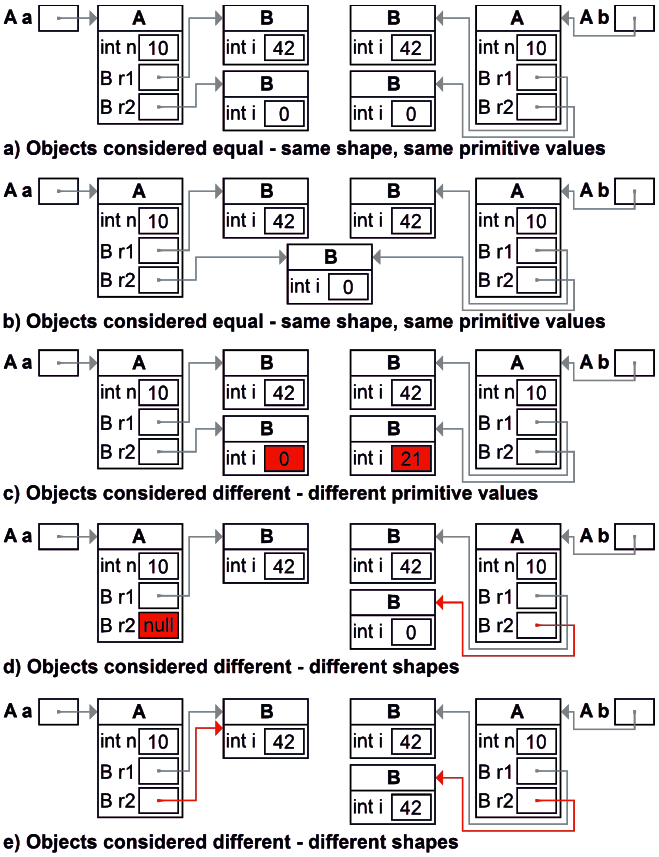

Fig. 4 Examples of objects considered equal and different

B. Forming of the Graph of an Object

When we compare two objects, it is checked whether both objects are not `null` and whether they are of the same class. If not so, the objects are considered different. If so, the DOC algorithm is used. Its first step is the creation of the graph representations of both compared objects. The representation is a directed vertex- and edge-labeled graph.

Each vertex of the graph corresponds to a single object of the internal structure of the object that we want to compare. The starting vertex of the graph corresponds to the object that we want to compare. Each vertex incorporates the ID, the state that is used during the comparison of two graphs (see Section IV.C) and the reference to the object that this vertex represents. It also incorporates a list containing types, names, and values of all primitive attributes of the object that this vertex represents.

Each directed edge of the graph represents a reference attribute of the object and points to the vertex, representing the object, to which the reference attribute is pointing. Each edge incorporates the type (i.e., the class) and the name of the reference attribute it represents.

The reference attributes that are arrays are treated specifically. If the attribute is an array of primitive values, each value is considered a primitive attribute with the name created from the name of the array attribute and the index of the value. If the attribute is an array of references, each value is considered a reference attribute, which means that it is

represented as an edge in the graph. The name is again constructed from the name of the array attribute and the index of the value. Similar approach is used for multi-dimensional arrays. As these arrays are represented as an array of arrays in Java, each inner dimension of the array is treated as an object (i.e., it is represented by a vertex in the graph) with attributes corresponding to the values on particular indices. Since these attributes do not have names, their indices are used instead. The reference attributes, which are `null`, and array attributes pointing to empty arrays are treated as primitive attributes.

The forming of the graph is based on the breadth-first search (BFS) algorithm starting in the object that we want to compare. There are a queue and a list of all vertices of the graph, which are empty at the start. First vertex is created from the object that we want to compare and is inserted to the list and to the queue. The forming of the graph then continues while the queue is not empty. First vertex is removed from the queue (current vertex) and all attributes of the object that is represented by this vertex are determined using reflection [8]. All primitive attributes are added to the list in this vertex. For each reference attribute, a new vertex is created. If this vertex is not in the list of all vertices, this new vertex is added to the queue and an edge is formed from the current vertex to the newly created vertex. If it is already present in the list, it is not added to the queue and a new edge is formed from the current vertex to the vertex from the list.

The presence of the vertex in the list is determined using the sequence searching and the comparison of objects that the vertices represent using the comparison operator "==". This way, it is ensured that already visited objects (e.g., due to cyclic references) are not visited again.

```
allVertices = List();
queue = Queue();
vertex = Vertex(comparedObject);
allVertices.add(vertex);
queue.add(vertex);
while (!queue.isEmpty()) {
  vertex = queue.remove();
  vertex.primitives=getPrimitives(vertex.object);
  references = getReferences(vertex.object);
  for (r: references) {
    neighbor = Vertex(r.object);
    index = allVertices.indexOf(neighbor);
    edge = Edge(r);
    if (index >= 0) {
      edge.vertex = allVertices[index];
    }
    else {
      edge.vertex = neighbor;
      queue.add(neighbor);
      allVertices.add(neighbor);
    }
    vertex.addEdge(edge);
  }
}
for (i = 0; i < allVertices.length; i++) {
  allVertices[i].ID = i;
}
```

Fig. 5 Pseudocode for the forming of the graph representation of an object

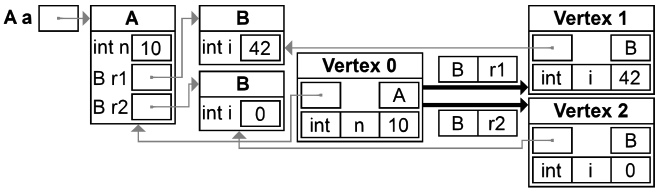

Fig. 6 The resulting graph (on the right) with references to the object (on the left)

When the queue is empty, the graph is fully formed. The last step is the assignment of the IDs to all vertices based on their indices in the list. These IDs are used during the saving of the graph to a file. The algorithm is described in Fig. 5 in pseudocode. The resulting graph for the object `a` from Fig. 4a is depicted in Fig. 6.

C. Comparison of Graphs

Two graphs created as described in Section IV.B can be easily compared without the references to the original objects, from which they were formed. The structure of each graph is represented by its vertices and edges, which also incorporate all necessary information – the values of primitive attributes and the names and types of all attributes. This is important, because the graph can be saved to a file and then loaded from this file and it is not necessary to recreate the original object, from which the graph was formed.

The graphs are compared using their parallel BFS exploration. Basically, in one loop, both graphs are explored.

```
v1 = graph1.rootVertex; v1.state = GRAY;
v2 = graph2.rootVertex; v2.state = GRAY;
queue1 = Queue();
queue2 = Queue();
queue1.add(v1);
queue2.add(v2);
equal = true;
while (!queue1.isEmpty()) {
  v1 = queue1.remove();
  v2 = queue2.remove();
  if (v2 == null) {
    equal = false;
    break;
  }
  if (!compare(v1.primitives, v2.primitives)) {
    equal = false;
    break;
  }
  for (e: v1.edges) {
    if (e.vertex.state == WHITE) {
      queue1.add(e.vertex);
      e.vertex.state = GRAY;
    }
  }
  for (e: v2.edges) {
    if (e.vertex.state == WHITE) {
      queue2.add(e.vertex);
      e.vertex.state = GRAY;
    }
  }
  v1.state = BLACK;
  v2.state = BLACK;
}
if (!queue2.isEmtpy())
  equal = false;
```

Fig. 7 Pseudocode for the comparison of two graphs

For each node, it is checked, whether they have the same count of primitive attributes with the same values. If a difference is found, the loop is ended prematurely and the objects are considered different. The objects are also considered different, when the graph of one of the objects is fully explored and the other is not. The algorithm is described in Fig. 7 in pseudocode.

V. Validation and Results

The described DOC algorithm was thoroughly tested using two sets of tests. In the first set, we focused on the correct functionality of the algorithm. The second set of tests was focused on the performance of the algorithm. All tests were performed on a standard notebook computer with dual-core Intel i5-6200U at 2.30 GHz with HyperThreading, 8 GB of RAM and 250GB SSD/500GB HDD. The software environment consisted of the Windows 7 SP1 (64 bit), Java 1.6 (32 bit), and Equinox OSGi framework.

A. Correct Functionality of the DOC Algorithm

The correct functionality of the DOC algorithm was tested by comparison of pairs of similar or equal objects. There were 5 pairs with variously complicated internal structures. The structures of all objects were created manually using the `A` class depicted in Fig. 8. The class and the objects were designed to test various situations, which can occur in internal structures of general objects – primitive attributes, reference attributes, arrays, and lists. Similarly, the changes introduced into one object of each pair of equal objects in order to create a similar but slightly different object represent various differences, which can occur in general objects – different lengths of an array, different values of an primitive attribute, different references, different elements of an array and/or a list. The internal structures of the objects were reasonably small (see Fig. 9) in order to enable controlled manual introduction of the changes and checking whether the results of the DOC algorithm are correct.

For each pair of objects, four tests were performed. In two tests, both objects of the pair were identical. In the remaining

```
import java.util.List;

public class A {
  private A parent;
  private int number;
  private String string;
  private List<A> list;
  private A[] array;

  public A(A parent, int number, String string,
      List<A> list, A[] array) {
    this.parent = parent;
    this.number = number;
    this.string = string;
    this.list = list;
    this.array = array;
  }
}
```

Fig. 8 The class `A` used for the testing of the functionality of the DOC algorithm

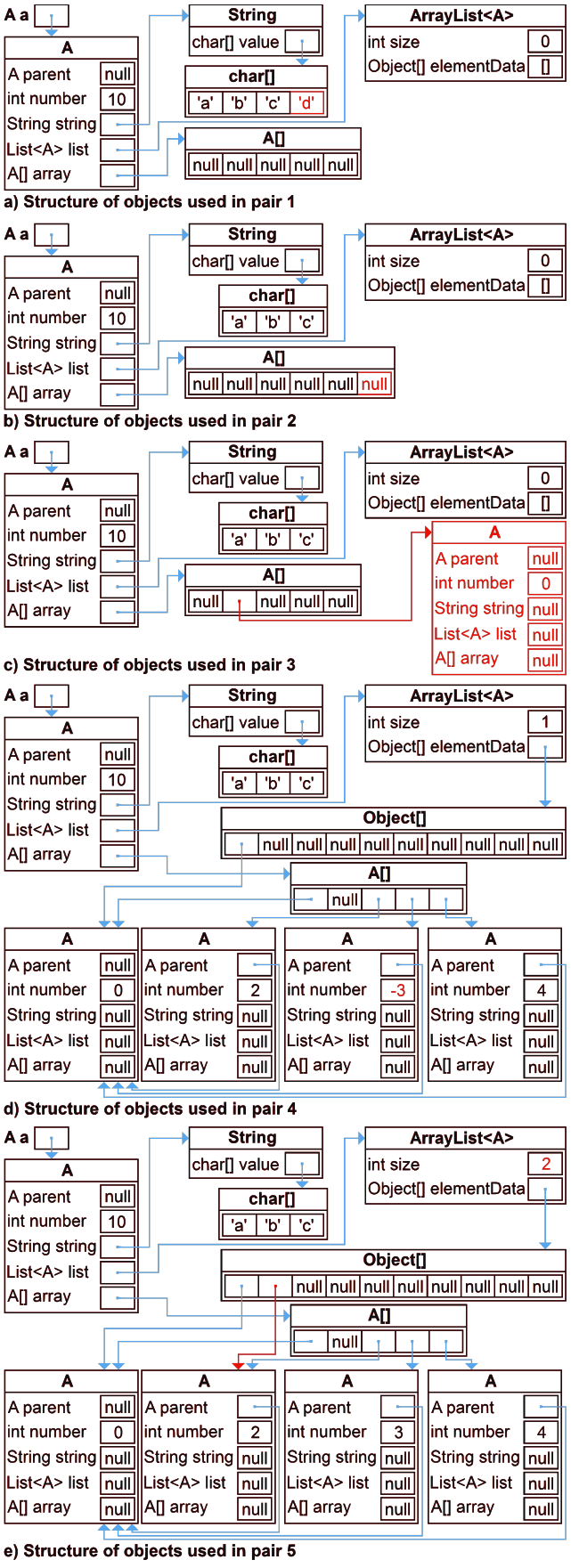

Fig. 9 Structures of objects used for the testing of the functionality of the DOC algorithm with highlighted differences

TABLE I THE RESULT OF THE COMPARISON OF PAIRS OF OBJECTS USING THE DOC ALGORITHM

Pair of objects	Both objects in memory		One object saved & loaded	
	Equal	Different	Equal	Different
1	`true`	`false`	`true`	`false`
2	`true`	`false`	`true`	`false`
3	`true`	`false`	`true`	`false`
4	`true`	`false`	`true`	`false`
5	`true`	`false`	`true`	`false`

two tests, one object was similar but slightly different. The differences are highlighted in Fig. 9. This way, both required results of the comparison (i.e., `true` and `false`) were tested. The graphs were created for both objects of the pair and then compared (once for identical objects and once for similar objects). Moreover, one of the graphs was saved to and loaded from a file and the comparison was performed again (once for identical objects and once for similar objects). This way, it was tested that the saving of the graph to a file will not affect the comparison.

For pair 1, the objects were instances with empty lists, arrays with 5 `null` elements, `null` parents, but with set values of numbers and strings. The difference introduced into one object was one character longer string (see Fig. 9a). For pair 2, the objects were the same, only the difference was that one array was one element longer (see Fig. 9b). For pair 3, the objects were again the same and the difference was that one array has one element (with index equal to 1) not `null`. Instead, it pointed to another instance of the `A` class with all attributes set to `null` or 0 (see Fig. 9c).

For pairs 4 and 5, the arrays had 5 elements, one element (with index equal to 1) was `null` and the remaining elements were instances of the `A` class with `null` strings, arrays and lists, but with parents set to the zeroth element of the array (with the exception of the zeroth element, which had the parent set to `null`) and the number set to the index in the array. The list contained one element corresponding to the zeroth element of the array. For pair 4, the difference was a different number value in the fourth element of the array (see Fig. 9d). For pair 5, the difference was one added element to the list (second element of the array – see Fig. 9e).

The results of the testing are summed in Table I. The algorithm returned the expected value (`true` or `false`) in every instance.

B. Performance of the DOC Algorithm

The performance of the DOC algorithm is important, because there is a significant number of object comparisons during both the generation of the testing scenarios and the comparison of two scenarios. Nevertheless, the number of comparisons is still far lower than it is necessary for the caching and similar techniques described for example in [7], [10], or [12]. For the testing, increasingly complex objects created using the `A` class were generated and compared. The compared objects were always equal. Different objects would cause premature ending of the comparison, because it is stopped on first uncovered difference (see Fig. 7). The results are summed in Table II.

TABLE II THE PERFORMANCE OF THE DOC ALGORITHM

Vertices count	Edges count	Graphs construction time [ms]	Graph comparison time [ms]
33	42	1.9	0.1
333	442	7.6	0.7
3 333	4 442	110.7	4.6
33 333	44 442	10 131.2	28.8

As can be seen in Table II, the graph construction phase is far more time-demanding than the graph comparison phase. There are several reasons for this. In the graph construction phase, the graphs of both the compared objects are created sequentially, which means that roughly half of the time is necessary to create the graph of a single object. More importantly, the sequential searching of the list of already visited objects is employed during this phase (see Fig. 5). Lastly, Java reflection is used during this phase, which is inherently slow [8].

The graph construction is not needed during the comparison of two testing scenarios, where the graphs are already constructed. However, it is necessary during the generation of the testing scenario for the comparison of invocations and consequences. Nevertheless, although these comparisons are numerous, the graph must be constructed only once per object. Multiple comparisons can then be performed using the graph representations of the objects only. So, the construction of the graph representations of the objects have a significant performance benefit in addition to the ability to easily save the graph representation to a file.

VI. CONCLUSION AND FUTURE WORK

In this work, we described the deep object comparison (DOC) algorithm. The algorithm was designed for our interface-based regression testing of software components for the comparison of general objects based on their internal structures and values of primitive attributes. Based on the performed tests, the algorithm works correctly for arbitrary objects, which can incorporate primitive and reference attributes, lists, and/or (single- or multi-dimensional) arrays. The performance of the algorithm seems to be reasonable, since the slower graph construction phase is expected to be performed by an order of magnitude less often than the faster graph comparison phase.

In our future work, we will incorporate the DOC algorithm to our interface-based regression testing of software components. We will then perform thorough testing focused on the resulting performance of the entire approach as well as on its increased ability to detect subtle changes in behavior of the component-based applications with new versions of components.

We will also consider the use of the DOC algorithm in another project focused on the generation of complex testing data (i.e., objects with complex internal structures). We plan

to employ particle swarm optimization [17] and/or combinatorial testing [18] to minimize the amount of generated data. The DOC algorithm can be utilized for the comparison of generated objects and/or their parts to remove any duplicity.

REFERENCES

[1] C. Szyperski, D. Gruntz, and S. Murer, Component Software – Beyond Object-Oriented Programming, ACM Press, New York, 2000.

[2] T. Potuzak, R. Lipka, and P. Brada, "Interface-based Semi-automated Testing of Software Components," Proceedings of the 2017 Federated Conference on Computer Science and Information Systems, Prague, September 2017, pp. 1335-1344, http://dx.doi.org/10.15439/2017F139

[3] The OSGi Alliance, OSGi Service Platform Core Specification, release 4, version 4.2, 2009.

[4] C. R. Rupakheti and D. Hou, "An Abstraction-Oriented, Path-Based Approach for Analyzing Object Equality in Java," 2010 17th Working Conference on Reverse Engineering, Beverly, October 2010, pp. 205-214, http://dx.doi.org/10.1109/WCRE.2010.30

[5] C. R. Rupakheti and D. Hou, "EQ: Checking the Implementation of Equality in Java," 2011 27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, September 2011, pp. 590-593, http://dx.doi.org/10.1109/ICSM.2011.6080837

[6] D. E. Stevenson and A. T. Phillips, "Implementing Object Equivalence in Java Using the Template Method Design Pattern," Proceedings of the 34th SIGCSE technical symposium on Computer science education, Reno, January 2003, pp. 278-282, http://dx.doi.org/10.1145/611892.611987

[7] D. Marinov and R. O'Callahan, "Object Equality Profiling," Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, Anaheim, October 2003, pp. 313-325, http://dx.doi.org/10.1145/949305.949333

[8] I. R. Forman, N. Forman, Java Reflection in Action, Manning Publications, 2004.

[9] K. Jezek, L. Holy, A. Slezacek, and P. Brada, "Software Components Compatibility Verification Based on Static Byte-Code Analysis," 39th Euromicro Conference Series on Software Engineering and Advanced Applications, Santander, September 2013, pp. 145-152, http://dx.doi.org/10.1109/SEAA.2013.58

[10] A. Infante, A. Bergel, "Object Equivalence: Revisiting Object Equality Profiling (An Experience Report)," Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages, Vancouver, October 2017, pp. 27-38, http://dx.doi.org/10.1145/3170472.3133844

[11] A. Infante, "Identifying Caching Opportunities, Effortlessly," Companion Proceedings of the 36th International Conference on Software Engineering, Hyderabad, May 2014, pp. 730-732, http://dx.doi.org/10.1145/2591062.2591198

[12] G. Xu, "Finding Reusable Data Structures," Proceedings of the ACM international conference on Object oriented programming systems languages and applications, Tuscon, October 2012, http://dx.doi.org/10.1145/2398857.2384690

[13] K. Nasartschuk, M. Dombrowski, K. B. Kent, A. Micic, D. Henshall, and C. Gracie, "String Deduplication During Garbage Collection in Virtual Machines," Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, October 2016, pp. 250-256

[14] G. M. Rama and R. Komondoor, "A Dynamic Analysis to Support Object-Sharing Code Refactorings,", Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, Vasteras, September 2014, pp. 713-723, http://dx.doi.org/10.1145/2642937.2642992

[15] M. J. Steindorfer and J. J. Vinju, "Performance Modeling of Maximal Sharing," Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, Delft, March 2016, http://dx.doi.org/10.1145/2851553.2851566

[16] N. Grech, J. Rathke, and B. Fischer, "JEqualityGen: Generating Equality and Hashing Methods," Proceedings of the ninth international conference on Generative programming and component engineering, Eindhoven, October 2010, pp. 177-186, http://dx.doi.org/10.1145/1942788.1868320

[17] B. S. Ahmed, L. M. Gambardella, W. Afzal, and K. Z. Zamli, "Handling constraints in combinatorial interaction testing in the presence of multi objective particle swarm and multithreading," Information and Software Technology, vol. 86, pp. 20–36, 2017 http://dx.doi.org/10.1016/j.infsof.2017.02.004

[18] M. Bures and B. S. Ahmed, "On the effectiveness of combinatorial interaction testing: A case study," 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), July 2017, pp. 69–76, http://dx.doi.org/10.1109/QRS-C.2017.20