Proceedings of the 16th Conference on Computer Science and Intelligence Systems pp. 501–510
DOI: 10.15439/2021F58
ISSN 2300-5963 ACSIS, Vol. 25

Semi-automated Algorithm for Complex Test Data Generation 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, 306 14 Plzen, Czech Republic
Email: tpotuzak@kiv.zcu.cz

Richard Lipka
NTIS – New Technologies for the Information Society, European Center of Excellence/Department of Computer Science and Engineering, Faculty of Applied Sciences, University of West Bohemia
Univerzitni 8, 306 14 Plzen, Czech Republic
Email: lipka@kiv.zcu.cz

***Abstract*—This paper describes in detail the Complex Object Generation (COG) algorithm, which is a semi-automated algorithm for the generation of instances of classes (i.e., objects) with a complex inner structure for Java and similar languages designed for black-box testing (i.e., without available source code). The algorithm was developed and tested as a stand-alone algorithm and can be used as such (e.g., during unit testing). However, we plan to use it to generate the parameter values of generated method invocations, which is a vital part of our interface-based regression testing of software components.**

I. Introduction

SOFTWARE development utilizing components has been around about two decades. Its main idea is to construct software from isolated parts (called software components), which can interact solely using well-defined interfaces. One of the purposes is to enhance the reusability of the software parts meaning to use a software component in different applications. On the other hand, a single application often consists of components possibly from different authors [1]. Besides the benefits of software parts reuse, there are also some setbacks, especially regarding testing. Since the components in a single application can originate from different authors, testing of their correct cooperation within this application is vital [2], because such testing cannot be performed by the authors of the individual components.

The necessity for testing is even more stressed by the common situation that the individual components exist in several versions. These versions can have only subtle changes, but can also differ significantly [3]. The individual versions of the component can have different internal calculations, interact differently with other components, or can have different public interfaces. Theoretically, the changes in the internal behavior of the component should not propagate past its interface. So, there should be no influence on the working of the entire component-based application. However, the reality is different from this theory. During the development of the new version of the component, new errors can be introduced, side effects of method invocations can change, computations may become more complex (e.g., because of an improved fidelity of the results) leading to a longer computation time and a time-out expiration. Further examples could continue. For the reasons described above, a thorough regression testing should be performed whenever a new version of a component is installed to a component-based application. From this point of view, it is not important whether there are changes to the public interface of the new version of the component or not [2], [4].

In order to support this regression testing, we developed a testing approach for components without available source code (i.e., black-box testing). This is, for example, the case of third-party components, which are not open source. It should be noted that some form of source code can be obtained using the reverse engineering even when it is not at our disposal directly. However, this requires additional effort and the results may not be ideal. Even the languages such as Java, whose byte code can be transformed to source code readily, can use obfuscation techniques [5] to hamper the reverse engineering. There are also legal aspects – the reverse engineering might violate the software license.

Our approach is designed for black-box testing for the situation when an old version of a component is replaced by its new version. The aim is to determine, whether both versions exhibit the same external behavior within the component-based application of our interest [2], [3], [4]. A prototype implementation of this approach was described in [2] in detail. It is implemented in Java and designed for the OSGi [6] component model, but its core ideas can be used also for other similar component models and languages [3]. It is implemented in our Interface Analysis Tool (InAnT). The approach is based on the analysis of public interfaces of the individual software components in the component-based application. The analysis discovers all services provided by

This work was supported by University specific research project SGS-2019-018 Processing of heterogeneous data and its specialized applications.

all the components together with the methods of these services. For each method, a set of method invocations (i.e., unique combinations of parameter values) is generated. These invocations are then performed and their consequences are observed in an iterative phase. This way, the behavior of the entire application is recorded. The process is performed in the application with the old and then with a new version of the component. The comparison of the two recordings can then show any changes in the behavior of the entire component-based application (presumably caused by the installation of the new version of the component) [2], [3], [4].

For the generation of the parameter values for the method invocations, various automated approaches can be used, for example the combinatorial testing [7] or the particle swarm optimization [8]. The approach used in our prototype implementation is rather primitive [3]. For parameters of primitive data types, several common and border values are used. For general objects, only `null` value is used [2]. Even then, our approach was able to uncover changes in our two test cases [2]. Nevertheless, usage of more realistic values would significantly improve the performance of our approach [3].

It should be noted that our interface-based approach for regression testing of software components is a black-box testing approach, which means that we do not expect the access to the source codes of the software components under tests [2]. At the same time, the most problematic part is the generation of complex objects. Since there are very few existing approaches for this task, we decided to develop a semi-automated approach for it. The approach described in this paper, the Complex Object Generation (COG), explores the structure of a selected class and enables to create its new instance (object) and fill its structure using existing constructors and manual changing of the attributes via a generated graphical user interface (GUI). The generated objects can be then used as parameter values for the method invocations. The COG was implemented in InAnT. First, it was tested as a stand-alone algorithm and can be used as such (e.g., during unit testing). However, it will be incorporated into our interface-based regression testing of software components. The main idea of the COG algorithm and its initial testing were briefly described in [3]. The detailed description of the COG algorithm and all its aspects including its further testing description is the main contribution of this paper.

II. InAnT Description

As mentioned above, the interface-based regression testing of software components is designed for component-based applications and, as such, its prototype implementation is a software component itself. It was described in [2] and one of its important parts – the Deep Object Comparison (DOC) – was described in [4]. The Complex Object Generation (COG) is another important part. Nevertheless, the basic notions of the component-based software development and the basic features of the InAnT are briefly discussed in following subsections in order to make this paper self-contained.

A. Component-based Software Development

A *software component* is a black-box software entity with a well-defined public interface consisting of provided *services*. The component can but does not have to require services of other components for its functioning. The components are expected to interact using their public interfaces only. These general features are common among various types of software components. However, the specific details of the aspects, behavior, and interactions of the software components depend on the used *component model*. A specific implementation of a component model is called a *component framework*. There can be (and often are) multiple component frameworks of a single component model [1], [4].

The prototype implementation of InAnT was implemented for the Java and the OSGi component model [2], [4]. Currently, the OSGi is quite widespread in both academic and industrial fields. There are several OSGi frameworks (i.e., implementations of the OSGi model), such as Felix or Equinox [2]. The OSGi components are called *bundles*. Each bundle is a single standard `.jar` file with additional meta-information related to the functioning of the OSGi (e.g., name and version of the bundle, required and exported packages, etc.) [2], [6]. Every bundle can provide several services in the form of Java interfaces. These services with the exported packages and their content form the public interface of the bundle [2]. The OSGi is a dynamic component model – the components can be installed and uninstalled on the fly (i.e., without the necessity to restart the OSGi framework) [9]. To enable this, the OSGi framework runtime provides means [6] for the control of the lifecycle of the bundle and also for the exploration of its environment [2].

The testing of the component-based application is similar to the testing of monolithic applications with additional issues caused by the composition of the components from different authors [2]. The testing methods can be divided based on the availability of the source code for the testers [10]. Source code can be used for the preparation of the tests leading to *white-box testing*. If it is not available or not used for the test preparation, the testing is called the *black-box testing* [11]. We consider this type, since the source code can be often not available for third-party components [2].

Regardless the type of testing, its main principle is to subject the tested software component to a set of inputs, to observe the outputs, and to compare them to the expected outputs [12]. A test is described in a so-called *scenario*. The content of the scenario forms the inputs and (optionally) the expected outputs. For the black-box testing of software components, each input can correspond to an invocation of a method of a service of the tested component [2].

B. Generation of Testing Scenarios

Our interface-based regression testing is used to find any changes of the behavior of a component-based application after a new version of a software component is installed instead of its old version. The application is tested with the old

version of the component and then with its new version. For both test runs, the invocations of the methods of the services of all components are generated, performed, and their consequences are recorded. The invocations and their consequences are stored to a recording – a scenario [2].

The InAnT prototype implementation has the form of a single OSGi bundle installed in the same framework as the component-based application under test. The invocation generation starts with the identification of all methods of all services of all components of the component-based application. This is achieved using the OSGi methods for the exploration of the bundles' environment and the Java reflection [2]. The found components, their services, and their methods are added into a data structure with the form of a tree [4].

The structure is then explored and, for every method, an initial set of invocations is generated. Each invocation is represented by the values of the method parameters. In our prototype implementation, the generation of these values is very simple – several common and border values are used for the parameters of primitive data types and only the `null` value is used for the objects [2]. Each invocation is a unique combination of parameter values of a method. The generated invocations are added to the data structure, which forms the basis of the scenario [4]. The purpose of the COG algorithm described in this paper is to provide additional object values.

Once the initial invocations are inserted into the data structure, the iterative phase begins. The data structure is explored and the invocations are consecutively performed (i.e., the corresponding method is invoked with the parameter values from the invocation). For each performed invocation, its consequences are observed. There can be multiple consequences of a single method invocation. There are four observed types – a thrown exception, a return value, a subsequent invocation of another service method, and a value change in output parameters of the method [2], [4]. All the consequences, which are not yet present in the data structure, are added to the invocation, which caused them.

The subsequent invocations are also added as invocations for the corresponding method (if they are not yet present). Their parameter values come from the internal logic of the components, making them more useful than the generated parameter values. The subsequent invocation generated in the $(n-1)$th iteration is performed in the nth iteration, where it can bring new consequences, which might remain hidden if only the generated initial invocations were used [2], [4].

Nevertheless, the recording of subsequent invocations has a setback. Since the components are black boxes, the causality of the performed invocation and its subsequent invocations is not certain. For example, if there are active threads in some components, they can perform invocations of methods of other components independently on our testing, yet these invocations will be recorded. This can cause false alarms during the testing scenarios comparison (see Section II.C) [2].

The iterative phase ends when there are no new consequences added in the last completed iteration [2], [4] or the preset maximal number of iterations was performed. The filled data structure is stored as a scenario with the old version of the component to an XML file [2], [4]. The tree-like nature of the data structure can be observed in Fig. 1.

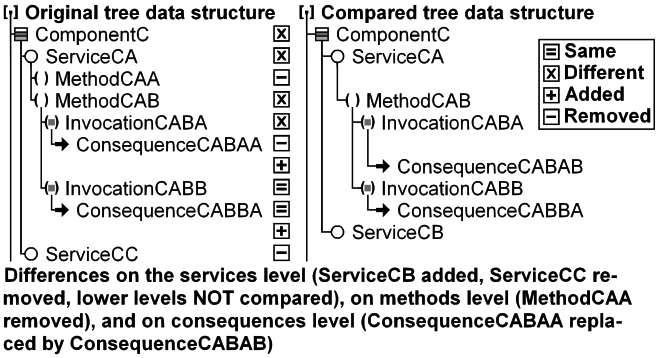

Fig. 1 An example two scenarios (data structures) comparison result

C. Comparison of Testing Scenarios

After the installation of a new version of a component into the application under test, the process described in Section II.B is performed again and a new scenario with the new version of the component is obtained. The scenario with the old version of the component is then loaded from the XML file and the data structures of both scenarios are compared on every level (i.e., the components level, the services level, the methods level, etc.) [2], [4].

On every level, the presence of an item in both structures is checked. If the item is present in both structures, the subtree of the item is expanded in both structures and the comparison continues in lower levels. If the item is missing in one of the data structures, this difference is reported and the lower levels are not explored further, since there is nothing to compare. The items are considered equal if and only if their subitems are equal on every sublevel [2], [4]. An example comparison result is depicted in Fig. 1.

The comparison result is also the result of the entire interface-based regression testing of software components. The differences found in the data structures indicate a change of the behavior of the component-based application under test after the installation of a new version of a component. The most important differences are on the invocations level and on the consequences level, since these differences cannot be easily detected by other means. The changes on the higher levels mean changes in the public interface of the components, which are detectable for example by an advanced static analysis (described for example in [13]) [2], [4].

III. Related Work

As it was mentioned in Section I, the most problematic part of the generation of the parameter values for the method invocations is the generation of complex objects. As far as we know, the research literature on this subject is limited, especially if it comes to the black-box testing. However, there are several works (partially) related to this subject.

A. Generation of Complex Data

There exist some tools for the complex testing input data generation. Nevertheless, they are in most cases designed for the web-based applications [14], [15], [16] and deal with data formats such as XML or JSON instead of instances. Additionally, they are not designed for black-box testing [3].

The PODAM tool [17] partially resembles the COG as it deals with standard Java objects. It enables to investigate their attributes and to fill them with random values based on their classes. The user can set the parameters of the random data generation or provide its own data where the random generation is insufficient or not desirable. However, this can be done only in the form of the user's own factory classes [17]. No GUI for direct input is provided. The tool also does not support the usage of objects created in past as attributes of the currently created objects. The source code is not required, but as the intended usage of the tool is the unit testing [17], which can be classified as white-box testing [3].

The JOP tool, developed during our past research [18], [19], enables to generate pools of complex Java objects. Its functionality is similar to the PODAM tool, but the objects created in past can be used as attributes of currently created objects [19]. The details of the object generation are described in annotations written into the source code [18], [19]. The object generation itself is possible without these annotations, but with a limited functionality. So, the knowledge of the source code is again presumed [3].

B. Exploration of Object Internal Structure

The knowledge of the objects' internal structure is necessary for their generation. This information is also vital in other fields such as memory optimization or object equality [3].

There exist papers focused on the memory optimization in programming languages with automatic memory management (e.g., Java). Their common idea is that some instances in the memory of a running program are equivalent and all equivalent instances can be replaced by a single instance without affecting the execution of the program [3], [4]. Examples can be found in [20], [21], [22], or [23]. In [20], it is pointed out that a thorough comparison of two objects (in order to determine their equality) requires checking of the graphs of the internal structures of the compared objects for isomorphism. Since this is a relatively time-consuming task and a large number of comparisons is expected for the memory optimization, many tools employ a sort of hash values or "fingerprints" of the objects to reduce the amount of necessary computations. Examples are in [20] and [22].

There are also several papers focused on object equality. In [24], the `equals()` method generator for complex objects is discussed. Deep equality is described recursively. The objects are deep equal if and only if, for all the corresponding fields of two compared objects, the deep equality holds [24]. We also employ the Deep Object Comparison (DOC) [4] for the comparison of objects in the InAnT tool. In the DOC, two objects are considered equal if and only if they have the same class, the graphs of their internal structures are isomorphic, and all values of the corresponding attributes of primitive data types in the corresponding vertices of the graphs have the same type and are equal [3], [4]. The DOC was used for the inspection of the internal structures of the instances and for their comparison during the testing described in this paper (see Section V).

C. Utilization of GUI for Setting Attribute Values

The employing a generated GUI for the data objects attributes values is discussed in several papers as well.

In [25], focused on a black-box testing of software components in .NET, the generation of the GUI for each component is discussed. This GUI enables an easier setting of input values and overall testing of the components. Reflection is used for the finding of the classes, methods and input/output parameters of the components [25].

With the FXForm2 tool, it is possible to automatically generate JavaFX forms from Java beans and to link the created GUI form fields with the Java bean properties [26].

IV. Description of Complex Objects Generation

The complex object generation (COG) is a semi-automated algorithm for generation of a new instance of a selected class, which source code is not known. For this purpose, an existing constructor is invoked with the parameters defined by the user with a generated GUI form. Then, the attributes of the created instance can be changed by the user with another generated GUI form. The algorithm is recursive, so the user can set both primitive and reference (object) values [3].

In comparison to more automatic tools described in Section III.A, the significant involvement of the user in the COG algorithm may seem like a setback. However, during the black-box testing, for which the COG algorithm is intended, the expertise of the user can be vital for the creation of instances with (at least partially) realistic attribute values. The user can utilize information, such as specification, vague textual description, or documentation, which may be at his or her disposal, but is not extractable automatically. He or she can also better interpret hints such as names of the attributes. Although the user can (unwittingly) introduce errors into generated objects, we consider the involvement of the user in the very generation of the objects an advantage.

The COG algorithm is designed for the generation of complex object parameter values for method invocations in our interface-based regression testing of software components. Nevertheless, it was first implemented and tested as a stand-alone algorithm usable wherever complex objects as input data are needed (e.g., during unit testing) [3].

A. Description of Data Structure

For the generation of a new instance of the input class, the COG algorithm requires a data structure ensuring the functioning of the algorithm and also enabling the storing of the instance generation process to disk. From the stored info-

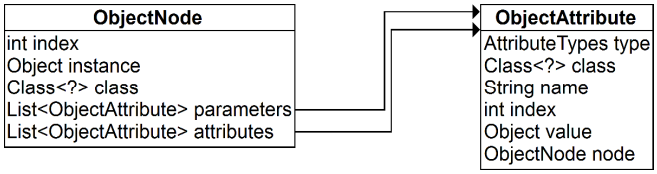

Fig. 2 The data structure for the storing of the generated instance

rmation, it is possible to reconstruct the generated instance in memory (see Section IV.G). The data structure consists of instances of two classes – one for the storing of the generated instance (called `ObjectNode`) and the second for the storing of the instance attribute values (called `ObjectAttribute`) [3]. Their attributes are depicted in Fig. 2.

The `ObjectNode` stores the information about the class of the generated instance, the list of the parameters of the constructor, which were used for the creation of the generated instance, the list of the attributes of the generated instance (non-static only), and the generated instance itself. The last attribute is the index in the *all nodes list*. In this list, all the generated `ObjectNode` instances are stored. The index also plays a role during the storing and loading to/from the disk (see Section IV.G for details). So, an `ObjectNode` instance contains all the information necessary for the creation of the generated instance [3].

The elements of the list of constructor parameters and of the list of attributes are the instances of the `ObjectAttribute` class. Each instance contains the information about the data type (i.e., the class) of the field (i.e., an instance attribute or a constructor parameter), the type of the field value (set internally or set manually), and the name and the index of the field. The name is used as a unique identification of the instance attributes and the index is used as a unique identification of the array cells (see Section IV.C) and the constructor parameters. Each `ObjectAttribute` instance also contains the value of the field, which can be a primitive value or a reference to an object. The last attribute is the reference to an instance of the `ObjectNode` class. This reference is set to `null` if the attribute is of primitive data type or its value was not specified by the user (i.e., it was set internally by the constructor). Otherwise, this last attribute points to the instance of the `ObjectNode` class describing the creation of the corresponding objects [3].

B. Description of Algorithm

The current version of the COG algorithm is implemented in Java and uses Java reflection [27] for the exploration of the classes' contents [3]. The reflection extracts the information from the bytecode, no source code is necessary. Hence, it is perfectly suited for our situation when the source code is not available. A consequence is that the COG algorithm is described with the limitations of the Java reflection in mind. For similar languages with available reflection (such as .NET platform), small changes in the implementation of the algorithm would be probably necessary, but its general idea should be utilizable in these languages as well [3].

```
createInstance(class, genericTypes, allNodes) {
  if (class.isInterface()) {
    classes = findClassesFor(class)
    if (classes.isEmpty()) {
      break; //Cannot continue without class
    }
    selected = readFromInput() //User input
    class = classes[selected]
  }
  node = createNode(class, genericTypes)
  if (!class.isArray()) {
    node.setArray(false)
    constructors = node.listConstructors()
    selected = readFromInput()
    constructor = constructors[selected]
    parameters = constructor.listParameters()
    processFields(parameters, allNodes)
    node.setConstructorParameters(parameters)
    node.createInstance(constructor, parameters)
    attributes = node.listAttributes()
    processFields(attributes, allNodes)
    node.setAttributes(attributes)
  }
  else {
    node.setArray(true)
    arrayLength = readFromInput()
    node.createArray(arrayLength)
    indices = node.listIndices()
    processFields(indices, allNodes)
    node.setAttributes(indices)
  }
  allNodes.add(node)
  return node
}
```

Fig. 3 Pseudocode for the main COG algorithm

Assume now that we want to generate parameter values for a method invocation and one of the parameter types is a class. The COG algorithm enables to create an instance of this class and to store it in the form of an instance of the `ObjectNode` to the all nodes list. The elements of this list (more specifically, the contained generated instances) can be used as constructor parameter values or attribute values of other generated instances. Before the generation of the first instance, the all nodes list is empty. The input of the COG algorithm is the class, for which the instance shall be generated, and the information about the generic type(s) if the class is parameterized (see Section IV.D) [3]. The main algorithm is depicted in Fig. 3. There are two main steps.

In the first step, the `ObjectNode` instance is created and all constructors available for the class being generated are found using the reflection. The constructors vary in the count and/or types of their parameters [3]. The names of the parameters are not stored in the bytecode, so they cannot be determined using the reflection. The list of the constructors (each constructor represented by its parameter types) is displayed to the user and he or she selects one of them.

The user then sets the values of the parameters of the selected constructor in a GUI form. For the parameters of primitive data types, he or she inputs the values directly or can use the default value (zero). For the object parameters, the user can select an instance of the `ObjectNode` from the all nodes list if there are any applicable instances. The applicable instances of the `ObjectNode` class must contain

a generated instance compatible with the data type of the constructor parameter. Only the applicable instances are displayed to the user. If there are no applicable instances in the all nodes list (besides the default `null` value) or there are some, but the user does not wish to use any of them, he or she can create a new instance for the constructor parameter recursively using the COG algorithm.

Once all the constructor parameters are inputted, they are stored in the list of the constructor parameters in the `ObjectNode` instance, each parameter as an `ObjectAttribute` instance. Since the names of the parameters are not known (see above), the parameters are identified by their order (stored as an index). The constructor is then invoked using the reflection and the new instance is created. This instance is then stored to the `ObjectNode` instance.

In the second step, all the non-static attributes of the instance being generated are found using the reflection. The information about them including their names and values are stored as the instances of the `ObjectAttribute` class to the list of attributes of the `ObjectNode` instance. The list of the attributes is displayed to the user as a GUI form. The user can change the values of the attributes similarly to the values of the constructor parameters (see above). So, the attributes of primitive data types can be set directly and the object parameters can be selected from the all nodes list or created recursively using the COG algorithm. The main difference is that the values (both primitive and object) can already have meaningful values set by the invoked constructor. So, the user can change only some values or no values at all [3]. In the latter case, the instance being generated is created solely using its constructor. The processing of constructor parameters and generated instance attributes is depicted in Fig. 4.

```
processFields(fields, allNodes) {
  for (field: fields) {
    if (field.isPrimitive() || field.isEnum()) {
      value = readFromInput() //User input
      field.setValue(value)
    }
    else {
      if (field.isInterface()) {
        classes = findClassesFor(field)
        selected = readFromInput()
        field = classes[selected]
      }
      createNew = readFromInput()
      if (createNew) {
        value = createInstance(field.class,
          field.genericTypes, allNodes)
        field.setValue(value)
      }
      else {
        values = allNodes.findValuesFor(field)
        selected = readFromInput()
        field.setValue(values[selected])
      }
    }
  }
}
```

Fig. 4 Pseudocode of the parameters and attributes processing

```
import java.awt.Point;

public interface IShape {
  public Point getPosition();
}

public class Circle implements IShape {
  private int radius;
  private Point position;

  public Circle(int radius, Point position) {
    this.radius = radius;
    this.position = position;
  }

  @Override
  public Point getPosition() {
    return position;
  }

  @Override
  public String toString() {
    return "Circle at " + position + ", r = " +
      radius;
  }
}
```

Fig. 5 The Java codes of the `IShape` interface and the `Circle` class

When the user changes the value of an object attribute, the reference to the corresponding `ObjectNode` instance (containing the new value) is set. However, when the user does not change the value of an object attribute, the reference to the `ObjectNode` remains set to `null`. The reason is that the value of this object attribute was created outside of the control of the COG algorithm. For example, it is not known, which constructor was used for the object creation and which parameter values were used. Any changes to the attribute values are stored to the list of the attributes in the `ObjectNode` instance and also directly to the generated instance.

At this point, the `ObjectNode` instance is added to the all nodes list. The finished instance referred from this `ObjectNode` instance can be used as a parameter value for the method invocation.

A simple example of the creating an instance of the `Circle` class (see Fig. 5) using the COG algorithm is depicted in Fig. 6. The black color is used for the generated instances and the gray color for the COG data structures.

We assume that the all nodes list is empty and the user chooses the first (and only) constructor. The `ObjectNode` instance for the `Circle` instance is created and the user must specify the constructor parameters. The `radius` parameter is of primitive data type and is set directly to 42. However, the `position` parameter is an object. Since the all nodes list is empty at this point, the user can select `null` value or can create a new `Point` instance using the COG algorithm. He or she chooses the `null` value. A new `Circle` instance is created and stored together with the constructor parameters to the `ObjectNode` instance (see Fig. 6a).

The user then inspects the attribute values of the `Circle` instance and sets a new value to the `position` attribute. He or she creates a new `Point` instance using the COG algori-

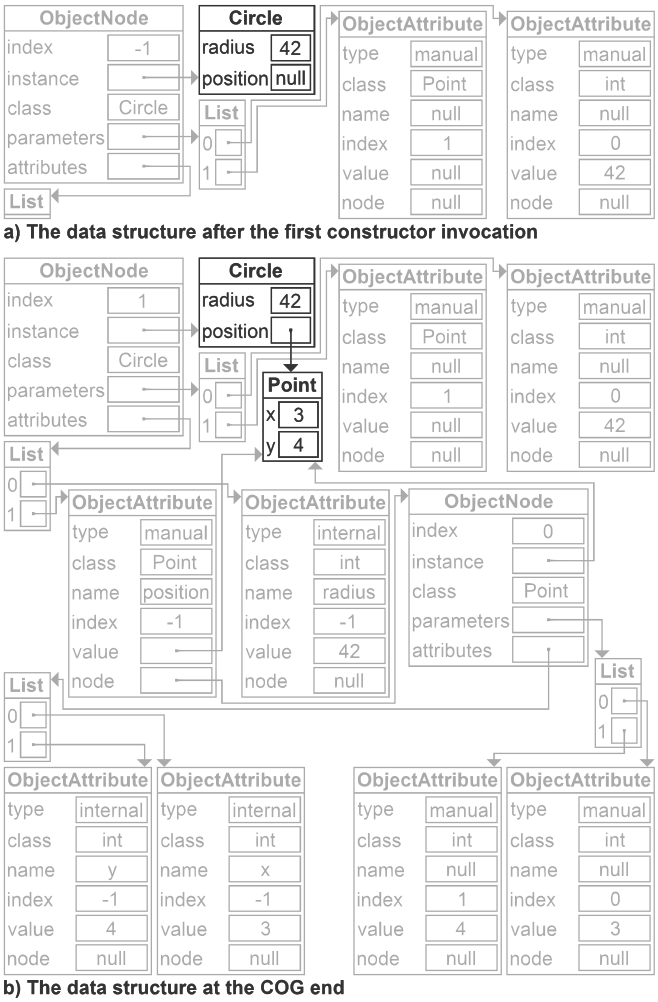

Fig. 6 The changes to the data structures

thm. The user chooses the constructor of the `Point` class with two `int` parameters. The `ObjectNode` instance for the `Point` instance is created and the user sets constructor parameters to 3 and 4. A new `Point` instance is created and stored with its constructor parameters to the `ObjectNode` instance. The user does not perform any changes of the attribute values and they are only stored to the `ObjectNode` instance. The `ObjectNode` instance for the `Point` instance is added to the all nodes list with index set to 0. The COG algorithm for the `Point` class ends and the created instance is set as the new value of the `position` attribute of the `Circle` instance. The user does no further changes. The attributes are stored to the `ObjectNode` instance for the `Circle` instance, which is then added to the all nodes list with index set to 1 (see Fig. 6b). It can be observed that the `ObjectNode` instance is created only for the attributes set by the user. The remaining attributes are considered values only, regardless of their data type (primitive or object).

The description of the COG algorithm and the example above shows only the most straightforward course of the COG algorithm. However, there are several aspects, which should be mentioned and which are discussed in following subsections. Some can be seen in the pseudocode in Fig. 3 and 4.

C. Array Handling

One of the important aspects is that the main input of the COG algorithm does not have to be only a class. It could be also an interface, which is discussed in Section IV.E, or an array. In Java, the arrays are basically instances of special classes containing predefined attributes (such as length of the array) and the indexed elements.

The COG algorithm handles the arrays (regardless whether of primitive or object data type) as classes with two main differences. The arrays do not have constructors, which can be used for the creation of a new instance of the array. However, it is possible to create an array of a specified component type with a specified length using the reflection. Hence, the user only specifies the length of the array instead of choosing the constructor. The elements of the array are not stored as named attributes, but instead as indexed cells. However, these cells are stored as the `ObjectAttribute` instances (one per array cell), only the indices are stored as the identification instead of the nonexistent attribute names. Besides these two differences, the COG algorithm proceeds in the same manner for the arrays as for the classes.

Since the multidimensional arrays are “arrays of arrays” in Java, they are handled in the same manner as the one-dimensional arrays, but with arrays in the cells.

D. Generics Handling

The class, which is the main input of the COG algorithm, can be generic, meaning that it has one or more parameter types. These parameter types cannot be determined universally using the Java reflection, but can be extracted in several cases. More specifically, it is possible to determine the parameter types for the parameters of the constructors and methods, for the return values of the methods, and for the attributes of a class. Since these cases cover all needs of the COG algorithm, we employ the parameter types checking to allow the user to select only compatible instances of generic classes.

The parameter types of input class are passed as a separate parameter of the COG method, since this information is not stored in the class. For this reason, it would not be possible to use the generic parameters for any general class with unknown parameter types. However, since the COG algorithm is designed for the generation of instances for method parameters, the parameter types of the class can be determined from the method parameter using the reflection.

E. Interface-Implementing Class Searching

Another quite a common possibility is that the method parameter, for which the COG algorithm shall generate the instance, is an interface, not a class. In that case, it is not possible to create the instance of this interface and it is not known, which class implementing this interface shall be used for the instance creation instead. The reason is that the input of the COG algorithm is the class, not its instance. The problem can arise also for the parameters of the constructor or during the changing of the value of an attribute.

```
findClassesFor(interface) {
  classes = findClassesInDirectories()
  classes.addAll(findClassesInJars())
  classesForInterface = List()
  for (c: classes) {
    if (interface.isAssignableFrom(c) &&
        !c.isInterface()) {
      classesForInterface.add(c)
    }
  }
  return classesForInterface
}
```

Fig. 7 Pseudocode of the finding the classes implementing an interface

For this situation, it is first checked, whether the input class is an interface. If so, the available classes implementing this interface are found and the user can select one of these classes. The selected class is then used instead of the interface in the remaining course of the COG algorithm (see Fig. 3). If there are no classes implementing the interface, the algorithm ends with a failure.

In the current implementation (see Fig. 7), all directories and `.jar` files in manually specified paths are searched for the `.class` files and the availability of each corresponding class is checked by the class loader. So, it is possible that some classes implementing the interface are missed. A straightforward solution is to find all classes implementing the interface available in the application context. The problem is that the function for the discovering of all implementing classes of an interface is not directly available in Java reflection. However, since the COG algorithm is primarily intended for the OSGi, it may be possible to use its services and find all implementing classes in all bundles of the OSGi framework. If we want to achieve similar task in plain Java, it is possible using the exploration of the class-path or using third-party solutions, such as [28].

A last resort solution is to create a mockup implementation of the interface using for example the `Proxy` class from the Java reflection [29]. This can be done manually for each interface, which can be very time consuming. Another possibility is to create a generic implementation utilizable for all interfaces. In both cases, it cannot be expected that the mockup implementation of the interface will have similar behavior to a real implementation. To find and implement the optimal solution is a part of our future work.

F. Exception Handling

In each phase of the COG algorithm, the user can use a `null` value instead of the creation or the selection of an instance. This value may be valid in many cases, but can also cause problems, usually in the form of a thrown exception. The problems occurring while utilizing the generated instance for the testing is outside the scope of this paper, but the problems can occur also during its generation – when the `null` value is used as a constructor parameter and the constructor does not permit this value. The problem does not occur while setting an instance attribute to the `null` value, since no methods of the generated instance are invoked.

The problem can occur not only because of the `null` value. Other values, such as a primitive type value outside an acceptable range or an object with unexpected attribute values can cause similar problems, typically in form of a thrown exception. For this reason, when an exception is thrown during the COG algorithm, the user is notified and can repeat the action, which caused the exception (typically invocation of a constructor) with different parameters. The user can also choose to interrupt the COG algorithm.

G. Storing and Loading to/from Disk

Since the creating of a very complex object can be quite time consuming for the user, it is possible to save the created `ObjectNode` instances from the all nodes list to disk. The `ObjectNode` instances are stored to an XML file, which is legible by humans. The generated instances contained in these nodes are not stored. Similarly, the values of their attributes not changed by the user are not stored. The storing of this information would require full scale serialization of general objects.

More importantly, storing this information is not necessary, because the generated instances can be reconstructed in the memory during the loading of the XML file using the remaining attribute values of the `ObjectNode` instances. These values are the parameter values of the utilized constructor and the attribute values changed by the user. All these values are either stored as other `ObjectNode` instances or are of primitive data type meaning they can be easily stored in a textual form. The references to the `ObjectNode` instances are replaced by the indices in the XML file. These indices correspond to the order of the instances in the all nodes list (and are also stored in each instance).

When the `ObjectNode` instances are loaded from the file, they are created in the order they were in the all nodes list prior to their storing. For each `ObjectNode` instance, the contained generated instance is created using the constructor corresponding to the stored parameter types and their stored values. At this point, the attribute values not changed by the user should be equal to their values prior to the saving to the XML file. Then, the attribute values changed by the user are set directly to the values stored in the XML file for the primitive data types and set to the correct references for the objects.

V. TESTS AND RESULTS

The functioning of the COG algorithm was tested using two sets of tests. In first set of tests, the very functioning of the algorithm was demonstrated using several different classes and arrays. In second set of tests, the storing and the loading of the generated instances were investigated. All the tests were performed on a notebook with dual-core Intel i5-6200U at 2.30 GHz with 8 GB of RAM, and a 250 GB SSD and 500 GB HDD. The installed software was Windows 7 SP1 64bit, Java 1.8 (64 bit), and Equinox OSGi framework.

A. Object Generation Testing

The correct functionality of the COG algorithm was tested in two scenarios. In the first scenario, the `Circle` class (see Fig. 5) and the `Rectangle` class, which also implements the `IShape` interface (see Fig. 5), together with the standard `Point` and `ArrayList` classes from the Java Core API were used. All the objects were created together in a single run, so all were placed to the all nodes list.

The user used the COG algorithm to create the instances in several steps (`<X>` denotes a reference and index):

1. Create `<0> = Point(10, 20)`
2. Create `<1> = Point(<0>)`
    a. Set `x = 42`
3. Create `<2> = Circle(30, <0>)`
4. Create `<4> = Shape()` (interface)
    a. Select `Rectangle` as implementing class
    b. Create `<3> = Rectangle(3, 4, null)`
        i. Set `sideA = 10`
        ii. Set `position = <1>`
5. Create `<5> = ArrayList()`
    a. Create `<4> = Object[2]`
        i. Set `0 = <0>`
        ii. Set `1 = <3>`
    b. Set `elementData = <4>`
    c. Set `size = 2` *(c. Set `size = 3`)*

The structure of the instances, which was intended to be generated, is shown in Fig. 8. The numbers of the instances are the indices in the all nodes list of the `ObjectNode` instances, in which the generated instances are contained. To determine, whether the created structure is correct, the entire `ArrayList` was printed using its `toString()` method. Its result is depicted in Fig. 9. The resulting structure was also inspected using the DOC algorithm [4]. The actual and the expected structures were manually compared. The structures were identical.

In order to show the possible issues caused by the direct involvement of the user, the instances were created again using the steps above, but with the last step (5.c) displayed using italics in parentheses. That means that the size of the `ArrayList` was set incorrectly (not corresponding to the actual size of its inner array). This inconsistency leads to an exception (a `ConcurrentModificationException`)

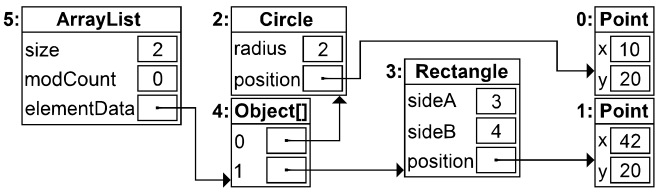

Fig. 8 The expected structure of the generated instances

```
[Circle at java.awt.Point[x=10,y=20], r = 30,
Rectangle at java.awt.Point[x=42,y=20]),
a = 3, b = 4]
```

Fig. 9 Result of the `toString()` method of the created `ArrayList`

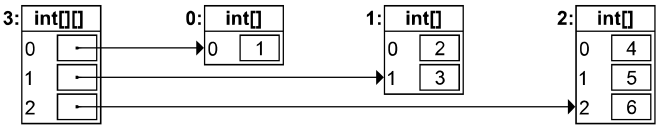

Fig. 10 The expected structure of the 2D array

```
[[1], [2, 3], [4, 5, 6]]
```

Fig, 11 Result of the `Arrays.deepToString()` method

when the `toString()` method is invoked. This shows that the user an easily set the attributes of the instances inconsistently. Since the COG algorithm does not understand the internal functioning of the created objects, it is not possible to perform an automated consistency control. So, the user must proceed with caution.

In second scenario, the user attempted to create 2D array with three rows and the length of each row increasing with its index (see Fig. 10). The following steps were taken:

1. Create `<3> = int[3][]`
    a. Create `<0> = int[1]`
        i. Set `0 = 1`
    b. Set `0 = <0>`
    c. Create `<1> = int[2]`
        i. Set `0 = 2`
        ii. Set `1 = 3`
    d. Set 1 = <1>
    e. Create `<2> = int[3]`
        i. Set `0 = 4`
        ii. Set `1 = 5`
        iii. Set `2 = 6`
    f. Set `2 = <2>`

The expected structure of the 2D array is depicted in Fig. 10. To verify the correctness of the created array, it was printed using `Arrays.deepToString()` method. The result is depicted in Fig. 11. Again, the resulting structure was also inspected using the DOC algorithm [4] and manually compared to the expected structure. The structures were again identical.

B. Storing and Loading Testing

To tests the correct functionality of storing and loading to/from the XML file, the `ObjectNode` instances stored in the all nodes list in first scenario (see Section V.A) were saved to an XML file and then loaded from it. The original all nodes list was copied elsewhere prior the XML loading to preserve the original generated instances. These original generated instances were compared to the saved and loaded

TABLE I THE RESULT OF THE COMPARISON OF THE ORIGINAL GENERATED INSTANCES AND THE SAVED AND LOADED GENERATED INSTANCES

Instance index	Instance class	DOC result
0	`Point`	equal
1	`Point`	equal
2	`Circle`	equal
3	`Rectangle`	equal
4	`Object[]`	equal
5	`ArrayList`	equal

generated instances using the DOC algorithm (each original generated instance compared to its corresponding saved and loaded counterpart). The results of the comparisons are summarized in Table I. It can be observed that all the pairs of the corresponding generated instances are equal suggesting that the storing and loading works well.

VI. Conclusion and Future Work

In this paper, we described the COG algorithm for the generation of complex objects utilizable as parameter values for the generated method invocations. The algorithm is semi-automated and the user actions are required. However, since the approach is intended for the black box testing, the expertise of the user can help to create instances with (at least partially) realistic attribute values. The generated instances can be stored to the disk for the future utilization.

In our future work, we plan to improve the handling the cases when the input class for the algorithm is in fact an interface. We also plan to enhance the user comfort by storing the information about failures. For example, when a value inputted by the user leads to an exception, this information is stored and used for a warning when the user attempts to make the same mistake again. We will also incorporate the COG algorithm to our interface-based regression testing of software components.

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,” in 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] T. Potuzak and R. Lipka, “Algorithm for Generation of Complex Test Data for Interface-based Regression Testing of Software Components,” SAC '21: Proceedings of the 36th Annual ACM Symposium on Applied Computing, Virtual Event, Republic of Korea, March 2021, pp. 1305-1308, http://dx.doi.org/10.1145/3412841.3442118

[4] T. Potuzak and R. Lipka: “Deep Object Comparison for Interface-based Regression Testing of Software Components,” in Proceedings of the 2018 Federated Conference on Computer Science and Information Systems, Poznan, September 2018, pp. 1053-1062, http://dx.doi.org/10.15439/2018F51

[5] J. T. Chan and W. Yang, “Advanced obfuscation techniques for Java bytecode,” Journal of Systems and Software, vol. 71, No. 1-2, 2004, pp. 1-10, http://dx.doi.org/10.1016/S0164-1212(02)00066-3

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

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

[8] 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

[9] D. Rubio, Pro Spring Dynamic Modules for OSGi™ Service Platform, Apress, USA, 2009.

[10] G. J. Myers, T. Badgett, and C. Sandler, The Art o Software Testing, Third Edition, John Wiley and Sons, Inc., Hoboken, 2012.

[11] P. G. Sapna and H. Mohanty, “Automated Scenario Generation based on UML Activity Diagrams,” International Conference on Information Technology, 2008, December 2008, pp. 209–214, http://dx.doi.org/10.1109/ICIT.2008.52

[12] S. J. Cunning and J. W. Rozenbiit, “Test Scenario Generation from a Structured Requirements Specification,” IEEE Conference and Workshop on Engineering of Computer-Based Systems, 1999, Proceedings, March 1999, pp. 166–172, http://dx.doi.org/10.1109/-ECBS.1999.755876

[13] 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

[14] Mockaroo. Accessed: 2018-35-05. [Online]. Available: https://www.mockaroo.com

[15] Dtm test xml generator. Accessed: 2018-05-05. [Online]. Available: http://www.sqledit.com/xmlgenerator

[16] Redgate. Accessed: 2018-03-05. [Online]. Available: http://www.red-gate.com/products/sql-development/sql-data-generator

[17] Podam - pojo data mocker. Accessed: 2018-03-05. [Online]. Available: https://github.com/mtedone/podam

[18] R. Lipka, “Automated Generator for Complex and Realistic Test Data,” in 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), July 2017, pp. 628-629, http://dx.doi.org/10.1109/QRS-C.2017.122

[19] R. Lipka and T. Potuzak, “Automated generator for complex and realistic test data - a case study,” in Communication Papers of the 2018 Federated Conference on Computer Science and Information Systems, Poznan, September 2018, pp. 1053-1062, http://dx.doi.org/-10.15439/2018F214

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

[21] A. Infante and A. Bergel, “Object Equivalence: Revisiting Object Equality Profiling (An Experience Report),” in 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

[22] G. M. Rama and R. Komondoor, “A Dynamic Analysis to Support Object-Sharing Code Refactorings,” in 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

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

[24] N. Grech, J. Rathke, and B. Fischer, “JEqualityGen: Generating Equality and Hashing Methods,” in 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

[25] F. Naseer, S. U. Rehman, and K. Hussain, “Using Meta-data Technique for Component Based Black Box Testing,” in 2010 6th International Conference on Emerging Technologies, Islamabad, 2010, pp. 276–281, http://dx.doi.org/10.1109/ICET.2010.5638474

[26] Dynamic JavaFX form generation. Accessed 2019-05-02. [Online]. Available: https://github.com/dooApp/FXForm2

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

[28] Java runtime metadata analysis. Accessed 2019-05-03. [Online]. Available: https://github.com/ronmamo/reflections

[29] Class Proxy. Accessed 2019-05-03. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html