МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2011
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2011
Proceedings of the Fortieth Jubilee Spring Conference
of the Union of Bulgarian Mathematicians
Borovetz, April 5–9, 2011

GENERATION OF TESTING SOURCE CODE FOR OBJECT-ORIENTED PROGRAMS[*]

Krassimir Manev, Anton Zhelyazkov, Stanimir Boychev

This paper presents the implementation of the last stage of a test data generator for structured testing of software system written in an object-oriented programming language – the generation of the testing source code. Some details of the implementation of the other stages that are important for implementation of the last stage are outlined. The algorithm used for generation of the testing source code is described.

1. Introduction. Automated testing (AT) of software and automated test data generation (ATDG) have no alternative nowadays. That is why the development of corresponding efficient AT-tools is an actual task – both for the theory and the practice. Being members of a project team developing a system for smart code analyzing and testing (SSA, [1]), the authors had to design and implement the ATDG part of the system. Some experience gained during the implementation of the ATDG for testing object-oriented programs is described in this paper. Further we will use the terminology from the systematic review of the state of the art in the domain [2].

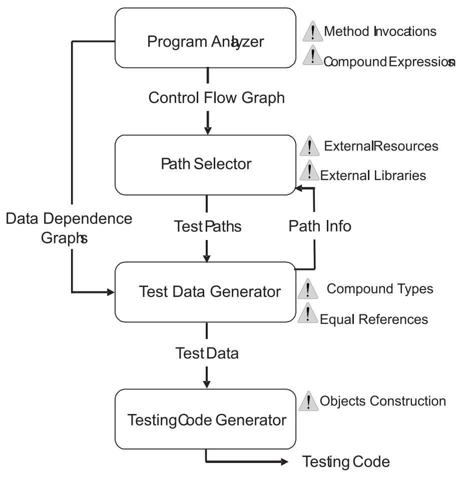

Fig. 1. Constraint solving test data generator scheme

Because the implemented ATDG is a part of a system for code analyzing, the *structured* (*white* or *transparent box*) *testing* was used, based on the *static analysis* of the software under testing. Among the different possible approaches for structured testing the *path-oriented approach* [3] with *constraints solving* [4] was chosen and among the different possible levels of testing – testing of a *set of modules*.

The traditional stages of ATDG for structured testing are the following (Fig. 1):

[*]**2000 Mathematics Subject Classification:** 68N30.

Key words: Automated Test Data Generation, Object-oriented Programs, Testing Source Code Generation.

This work was partly supported by the Bulgarian National Science Research Fund, Contract 02-102/2009.

Program analysis. At this stage a *control flow graph* of the program (CFG, [3]) and a *data dependence graphs* (DDG) for each local variable are constructed;

Paths selection. At this stage a set of paths is chosen, so as to "*cover*" the CFG of the program [3], following some *testing criteria*;

Generation of test data. At this stage a *system* (*conjunction*) of constraints is generated for each of the selected paths, corresponding to the branching conditions along the path [4]. Then the satisfiability of each conjunction of constraints is checked by some ***Sat*** *solver* and if the conjunction is satisfied, then an attempt is made to solve the system of constraints by some of the available *constraints solvers.* Finally a *test set* is constructed. Each of its *test cases* is a solution of the corresponding system of constraints.

The target language of the project was Java. That is why, we had to add to the traditional stages of ATDG one more:

Generation of testing source code. At this stage a *Java class* for testing the module is constructed (based on the *JUnit* technology [5]). The testing class has to arrange the generated test data in the necessary input objects, to execute the module under testing on each generated input and to collect the obtained results.

The difficulties that arose during the implementation of the traditional three stages (shown in Fig. 1 after exclamation signs) were discussed in [6]. In this paper we will concentrate on the implementation of the forth stage – Generation of the testing source code. Some steps of our algorithm for generation of the testing code depend on decisions made on some of the first three stages. That is why, in Section 2 and Section 3 we will briefly present these issues of the object-oriented programming that influence significantly developed algorithm. The algorithm itself is presented in Section 4. In Section 5 the conclusion and the results from some experiments are given.

2. Compound types. One of the main problems arising from the process of test data generation is that the existing Sat solvers are not able to resolve constraints involving instances of compound data types – strings, lists, trees, sets, maps, etc. In generation of test sets for testing object-oriented programs especially big problem is the resolving of constraints on instances of the input objects. For solving the problem we had to split each constraint involving such instances to the included elementary data – approach that we called atomization [6]. Atomization is not an easy and proper solution, because of the large number of constraints could make the obtained Sat problem enormously large and practically unsolvable (Sat problem is NP-complete). Something more, if some attributes of an object are objects too, then we had to apply atomization in depth,which makes the problem even harder. We will illustrate the typical difficulties that we have met in generation of instances of compound data types with the following:

Example 1.

```
int validate(String url,int minLen, String fName)
{
    if (fName.length() >= minLen {\&}{\&} url.contains(fName)
          && (url.startswith(''http://'') || url.endswith(''.pdf'')))
        return true;
    return false;
}
```

Given the module from the Example 1, we want to generate string variables url, fName and an integer variable minLen that satisfy the condition of if operator. This condition should be converted into constraints for the **Sat** solver. The parts of the condition that concern different variables cannot be treated separately, because they are mutually dependent. Following the atomization approach we represent each string by a list of variables comprising the values of the characters and a variable for the size of the string.

So the variable `url.startswith(''http://'')` should be transformed to:

```
url.size > 7 && url[0] == 'h' && url[1] == 't' && url[2] == 't'
&& url[3] == 'p' && url[4] == ':' && url[5] == '/' && url[6] == '/'.
```

The condition `url.endswith(''.pdf'')` should be treated in the same way demonstrating one of disadvantages of the atomization approach – the increasing number of constraints. The atomization of the condition `url.contains(fileName)` is even more serious problem.

Similar problems arise not only with string but with any aggregated data type, too. The discussion of possible solutions is out of the scope of this paper and will be a subject of future research.

3. Equal references. In object-oriented languages *objects* are composed of *attributes*. Each attribute could be an object composed of its own attributes too and so on. The constraints, which include object-variables, lead to some additional difficulties. Let us consider the following:

Example 2.

```
int check(Point a, Point b)
{  if(a == b)
   {  if(a.x > 5 && b.x < 10) return 1;
      return 2;
   }
   return 3;
}
```

When we have == constraint on two object-variables (which means that they refer to the same object), this can lead to some difficulties if we have also constraints that refer to attributes of these objects. In such case the references to the same attributes of two objects has to be considered as a same object too. Else we could generate different values for the same attribute, which is unacceptable (in our Example – value 11 for `a.x` and value 4 for `b.x`).

Suppose `a` and `b` are variables that can refer to the same object and $c_1, c_2, \ldots, c_n$ are their attributes. Following the atomization approach we have to form the constraint `((a == b) => (a.`c_1` == b.`c_1` && a.`c_2` == b.`c_2` && ...&& a.`c_n` == b.`c_n`))`. The same had to be recursively applied to the attributes, which are of reference type too. This could increase enormously the number of constraints.

The simple substitution `a`$\rightarrow$`b` could be a solution, but only when `a == b` is not combined with another constraint on the attributes (see again Example 2 but with a condition `a == b` || `a.y > b.y` of the outer `if`). Fortunately, this case could be eliminated at the analysis stage as a "hidden" branching.

4. Implementation of testing code generator. The part of the object-oriented ATDG that generates testing source code has to construct series of statements that will create the input variables and will invoke the module under testing. This part was implemented on the base of the *JUnit* technology [5]. The solutions of the constraints are the values of the elementary (atomized) variables. Using these data we have to generate the statements of the *JUnit*-code that instantiate the input variables. The input variables could be of primitive types but could be of reference types and some of its attributes can reference to other objects too. That is why, when generating the statements of testing code, the following should be taken into account:

- The order of the generated statements is important. For example, we have to instantiate first the object before setting its fields. Or if a field can be passed only in the constructor of the object, it should have already been instantiated;
- In OOP encapsulation allows hiding some fields and their initialization/modification can be done only through the constructor or a setter method. For example, if an attribute is defined as private it should be modified through a setter method. In order to create such object we have to find the proper constructor or setter;
- Some variables can point to the same object and there can be cyclic references (i.e. object `A` has an attribute referencing to object `B` and vice versa).

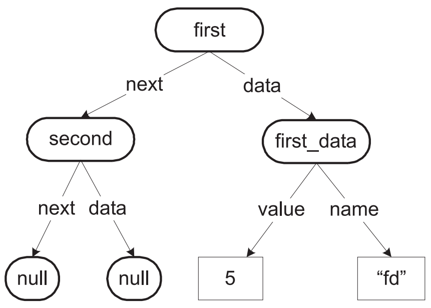

Fig. 2. Object representation

4.1. Representation of an object. A good way to represent a compound object is using a directed graph, where each node is an object and its ancestors are the nodes of its attributes. The class structure from Example 3 is illustrated with the graph from Fig. 2.

Exampe 3.

```
class Element {
  private Data data;
  { this.data = data; }
  public Element(Element next, Data data)
  { this.data = data; this.next = next; }
  public void setNext(Element next)
  { this.next = next; }
}
```

```
class Data {
  private int value, String name;
  public Data(String name,
int value) {
     this.name = name;
     this.value = value; }
}
```

4.2. Creation of the objects. The representation then can be used to generate the statements that will create an object with such representation. An object can be initialized using three different methods – with *constructor*, with *invocation of a setter* or with *reflection* (a Java interface that gives an access to private fields [7]).

The easiest way is to initialize everything using reflection, as it escapes the restrictions of the encapsulation, but this can create an invalid object. Thus we make an analysis of the given class constructors and setters and choose the best possibility. Reflection could be used when we don't have another way to set a field, because it is not visible to us (private field).

As an example, let us generate an object `first` of class `Element` using the following results received from the constraints solver (Fig. 2):

```
first is Element; first.data is Data;
second is Element;
first.next = second; first.data.value = 5; first.data.name = ''fd''.
```

In order to generate object `first`, we should first generate its attributes – `next` and `data`. After analyzing the setters and constructors of the class `Element`, we see that the attribute `next` can be initialized with the `setNext` method and the attribute `data` – with the constructor. The construction of `first.data` can only be done using the constructor, setting both its attributes – `value` and `name`. As they are primitives, they need no separate initialization and the generated code is:

```
Data a_data = new Data(5, ''a'');
Element b = new Element(null);
Element a = new Element(a_data);
a.setNext(b);
```

The process is described as Algorithm 1.

Algorithm 1.

```
generate (Var) {
   if (Var is primitive) {
     generate statement to create and initialize V;
     return;
   }
   for (each attribute in Attributes_of_Var) {
      generate (attribute);
   }
   find the best constructor and the attributes that
       will be set using setters and reflection;
   generate statement to construct the object;
   generate setter invocation statements;
   generate reflection invocation statements;
}
```

The above mentioned approach does not handle the case, when two variables have a reference to the same object or when we have cyclic references. For example, let us consider the three instances of the class `Element` that have to satisfy the description obtained by the constraints solver shown in Fig. 3.

```
a, b, c is Elment; a.next = b;
b.next = c; c.next = a.
```

In such case Algorithm 1 will result in infinite recursion, because as it can be seen on Fig. 3 we have a cycle in the representation graph. In order to escape the cyclic referencing, we can construct a new graph – *statements dependency graph* (SDG). Instead of objects the nodes of the SDG represent the statements that initialize it.

Each node of the SDG is from one of the following types:

- *constructor* – corresponding to the statement with object initialization through a constructor;
- *assignment* – used when more than one variable is referencing the same object;
- *setter* – setting attribute of an object;
- *reflection* – setting hidden attribute of an object.

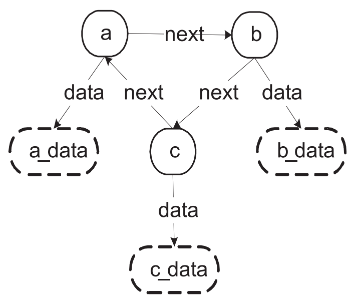

Fig. 3. Cyclic referencing

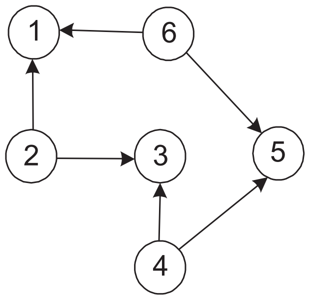

Fig. 4. Statements dependency graph

Each edge of SDG shows a dependency between two statements, i.e. the edge (v, w) means that w should be executed before v. We will have edges between the following nodes:

• between each assignment/setter/reflection node and the constructor node of the referred attribute;

• between a constructor node and each of the constructor nodes of its parameters.

How to determine when we have dependency between two statements? Before generating the statement we have the knowledge about the different variables, how they are constructed and the way they will be set if they are attributes. Denoting with *Var_statement* the construction statement of *Var* we have the Algorithm 2.

Algorithm 2.

```
//For each variable V invokes this method
build_statements(V)
  if (V statements are already built) return;
  if (V is constructed by constructor) {
    build V_statement;
    for (each parameter P of V's constructor) {
      build_statements(P);
      add the dependency (V_statement, P_statement)
    }
    for (each Var to be set with setter) {
      build_statements (Var);
      build Var's setter invocation statement Var_SIS;
      add the dependency (Var_SIS, Var_statement);
      add the dependency (Var_SIS, V_statement);
    }
    for (each Var to be set with reflection)
      build_statements (Var);
      build Var's reflection invocation statement Var_RIS;
      add the dependency (Var_RIS, Var_statement);
      add the dependency (Var_RIS, V_statement);

  }
  else  // V is constructed with assignment
```

```
    build $V$'s assignment statement;
    let $M$ be the variable that $V$ is pointing to;
    build_statements ($M$);
    add the dependency (V_statement, M_statement);
  }
}
```

As an example let us generate the statements dependency graph of the three instances of class `Element` from Fig. 3. The statements that are generated by Algorithm 2 for this example, listed in the order in which they are generated by the Algorithm and labelled from 1 to 6, are:

```
1.  Element a = new Element(null); 2.  a.setNext(b);
3.  Element b = new Element(null); 4.  b.setNext(c);
5.  Element c = new Element(null); 6.  c.setNext(a);
```

The corresponding SDG is shown on Fig. 4. Now for obtaining the order in which the statements will be included in the testing source code we need to sort topologically the graph.

One possible outcome from the topological sorting is the order: 1, 3, 5, 2, 4, and 6. The correct sequence of statements that will construct the necessary for the testing object is:

```
Element a = new Element(null);
Element b = new Element(null);
Element c = new Element(null);
a.setNext(b); b.setNext(c); c.setNext(a);
```

5. Conclusion. Implementation of a complex software analyzing tool with automated test data generator for object-oriented programming, as described above, is a big challenge even when the selected approach (path covering with constraints solving in our case) is relatively well developed and discussed in the literature. Beside the classical problems of the approach, some additional problems arise from the usage of objects with compound structure.

Straight forward solutions of the problems, as the mentioned above atomization in depth are sometime inevitable and usually lead to some of the classical problems of the approach – a very big system of constraints in our case. Something more, atomization leads to a *virtual deconstruction* of the considered objects and inevitable process of their *reconstruction* to real objects that will be used for testing.

With the proposed algorithm in the paper we succeed to solve (to some extent) the problem and to implement a working ATDG for testing modules written in Java. In order to estimate the performance of the implemented ATDG some experiments was conducted with two complex systems with open source code written in Java (592 classes, 3166 methods and 13034 instructions in the first, and 92 classes, 809 methods and 4300 instructions in the second, respectively). As it was recorded by the automated tool for control of the test generation process *EclEMMA* [12] we succeed to cover about 80% of the classes, 50 % of the methods and 35% of the lines of the tested systems in average. This statistics shows that there is still a large field for amelioration of the chosen approaches and algorithms.

REFERENCES

[1] Smart Source Analyser (*SSA*). Under development in Musala Soft, Ltd.: `http://www.musala.com`

[2] Sh. Mahmood. A systematic review of automated test data generation techniques, Master thesis MSE 2007:26, School of Engineering, Blekinge Institute of Technology, Sweden, Oct. 2007.

[3] A. H. Watson, J. T. McCabe. Structured testing: a testing methodology using the cyclomatic complexity metric, NIST Special Publication 500–235, September, 1996.

[4] R. A. DeMillo, A. J. Offutt. Constraints-based automatic test data generation, *IEEE Trans. on Software Engineering*, vol. **SE-17**, No 9, September 1991, pp. 900–910.

[5] Resources for test driven development: `http://www.junit.org/`

[6] Kr. Manev, A. Zhelyazkov, St. Boychev. Implementation of an Object-Oriented Test Data Generator, Proc. of Intern. Conference ov e-Learning and Knowledge Society – e-Learing'10, Riga, August 2010.

[7] Using Java reflection: `http://java.sun.com/developer/technicalArticles/ALT/Reflection/`

[8] A. Goldberg, T. C. Wang, D. Zimmerman. Application of feasible path analysis to program testing, Proceedings of the 1994 International Symposium on Software Testing, and Analysis, Seattle WA, August 1994, pp. 80–94.

[9] M. R. Garey, D. S. Johnson. Computers and intractability: a gide to the theory of NP-completeness, W. H. Freeman and Company, 1979.

Krassimir Manev
Sofia University
Faculty of Mathematics and Informatics
5, J. Bourchier Blvd
1164 Sofia, Bulgaria
`e-mail: manev@fmi.uni-sofia.bg`

Anton Zhelyazkov
Stanimir Boychev
Musala Soft
36, Dragan Tsankov Blvd
1057 Sofia, Bulgaria
`e-mail: anton.zhelyazkov@musala.com`
`e-mail: stanimir.boychev@musala.com`

ГЕНЕРИРАНЕ НА КОДА НА ТЕСТВАЩ МОДУЛ ЗА ОБЕКТНО-ОРИЕНТИРАНИ ПРОГРАМИ

Красимир Манев, Антон Желязков, Станимир Бойчев

В статията е представена имплементацията на последната фаза на автоматичен генератор на тестови данни за структурно тестване на софтуер, написан на обектно-ориентиран език за програмиране – генерирането на изходен код на тестващия модул. Някои детайли от имплементацията на останалите фази, които са важни за имплементацията на последната фаза, са представени първо. След това е описан и алгоритъмът за генериране на кода на тестващия модул.