Implementation of an Object-Oriented Test Data Generator

Krassimir Manev, Anton Zhelyazkov, Stanimir Boychev

Abstract: *The paper summarizes the experience gained from implementation of a test data generator for structured testing of a software system written in an object-oriented programming language. Test generator is based on the static analysis of the source code under testing. Among the different possible approaches for structured testing the path-oriented approach with constraints solving was chosen and among the different possible levels of testing – the testing of a set of modules (methods). The main stages of implementing the test data generator are considered and for each of the stages some of the arising problems and the implemented solutions are described.*

Key words: *Automated Test Data Generation, Structured Testing, Path-oriented Approach, Constraints Solving.*

INTRODUCTION

Automated testing (AT) of software and automated test data generation (ATDG) are obligatory for contemporary software industry. However, they are significant for other domains too, for example, the education in programming. Checking of hundred programming assignments manually is extremely difficult. That is why some teachers use special purpose software – the so called *grading systems* – for automated checking of students programs [7]. The design and development of automated grading systems could be considered as a new, promising, sub domain of e-learning. Test data generation is an essential part of the grading process, which is still not entirely automated.

As members of a team developing a system for *smart object-oriented code analyzis and testing* (SSA, [9]), we design and implement an ATDG tool, which could be used both for testing the projects of a software company and for checking students programs. Our experience gained through implementation of a selected approach is described below. Next we will use the terminology from [9] - a systematic review of the state of the art in the domain. The implemented tool has been designed and developed according to the requirements identified and described in [8].

Our ATDG tool is a part of a system for code analyzis. So, the *structured* (*transparent box*) *testing* is implemented, based on the *static analysis* of the *software under testing*. Among the different possible approaches for structured testing the *path-oriented approach* [12] with *constraints solving* [1] has been chosen.

Traditional stages of ATDG for structured testing, summarized in [9] are the following (Fig.1):

- **Program analysis.** At this stage a *control flow graph* of the program (CFG) 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 [12], following the *testing criteria* "each edge at least once";
- **Generation of test data**. At this stage a set (*conjunction*) of constraints is generated for each of the selected paths, corresponding to its branching conditions [1]. Then the satisfiability of each conjunction is checked by a ***SAT** solver* and if it is satisfied then an attempt is made to

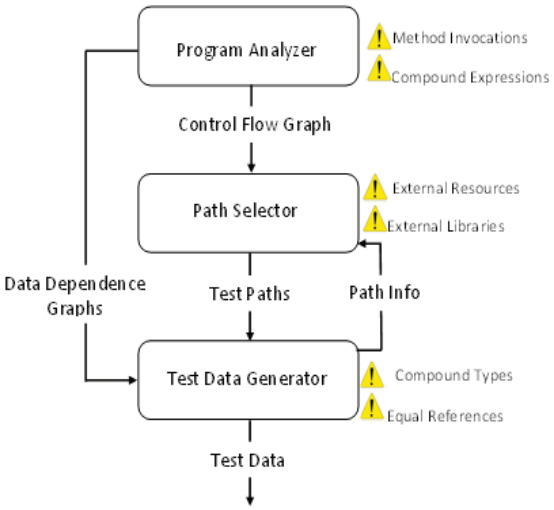

Fig.1. Stages of the test data generator

solve the constraints by a *constraints solver*. Finally the *test cases* are constructed as solutions of the corresponding system of constraints.

The path-oriented approach for structured testing was created in the age of the procedure programming. It was developed for testing of a single module. From a practitioner's point of view it is a great challenge to apply this approach for testing of a *set of modules* written in an object-oriented language (Java in our case).

There are some well known and widely discussed, "classical", problems of the path-oriented approach with solving of constraints, namely:

- Practical impossibility to test all paths of the CFG of the module, because of the exponential growing of their number [12] – a problem that we denote as EXP;
- Problems arising from the existence of unfeasible paths [5] – UNF problem;
- NP-completeness of the task for solving an arbitrary conjunction of constraints [4] – NPC problem;
- Principle impossibility to solve some constraints – SOL problem, etc.

In this paper we will not consider the above mentioned classical problems. We will rather stress the ***less commented*** (or ***not commented*** at all) problems due to *testing a set of modules* (methods) instead of testing a single module and testing of *object-oriented programs*. The OO-paradigm appends new problems to the classical ones and increases some negatives inherited from the classical problems.

In the next three parts of the paper a number of problems observed during the implementation of the corresponding stages are presented and some possible solutions are discussed. In the conclusion the results of our work are summarized and some experiments are briefly described.

IMPLEMENTATION OF A PROGRAM ANALYZER

Methods invocations. Methods invocations cause problems when we construct the CFG of the tested module. This is valid not only in the OO-case but in the procedure programming, too. If an invocation of method (or function) does not change the state of the method (module) under testing it could be ignored using the *mocking* technique [11]. Mocking means that the necessary value is chosen automatically as returned by the invoked method without selection of appropriate values for its parameters. Mocking does not generate good test cases. If we want to generate a useful test set we can't mock all invocations. If an invocation changes the state of the module, it could not be mocked at all. One possible solution of this problem is to replace the node of the method invocation with the CFG of the invoked method.

Replacement leads to serious problems in the case of recursive invocations. Assume that two functions invoke each other recursively. If we decide to replace invocation node with the CFG of the invoked function, then it will be difficult to predict the number of replacements. We can handle this by limiting the number of replacements to some extent, using again the mocking technique. Such solution could lead to generation of large and unusable CFG, as well as to appearing of more unfeasible paths (i.e. the problem is transformed to EXP or UNF).

The authors of path-oriented approach do not consider the treatment of the invocations at all. Our attempt to solve the problem is based on the fact that the developed tool will be used for smart analysis of a set of modules. We decided to mark the node in the CFG of the analyzed module as an *invocation node* and ***to postpone solving of the invocation problem to the path selection stage***.

When during the path-selection process we reach a node, marked at the previous stage as invocation node, we first have to decide to mock or to expand, depending on our expanding criteria ("expanding only once", for example). If we choose to expand, we save the current state of the path-selection in a stack (so as to be able to go back after

processing the invoked method) and the process goes to the CFG and DDG of the invoked method. If the graphs of the invoked method are not built yet, this is the moment to build it. In such a way we do not connect the graphs of the two (or more, if we have to expand in depth) methods physically. The path-selection process is working on separate graphs of the modules, which are simpler. So, in our implementation ***the process of constructing CFG and DDG of the different modules is not an independent stage***. It is interfering with and is dynamically driven by the process of paths selection.

Compound expressions. When a CFG of a method is constructed, each statement is usually represented by one node of the CFG. If a statement contains one or more compound expressions, different problems arise. If we have more than one method invocation in an expression (as in Example 1.A) the order in which the compiler evaluates arguments of the corresponding expression should be taken into account (the first executed invocation could change the data or to return data that are used in the other invocations). The order is also important because we have to expand the invocations on the next stage of the process, as mentioned above.

EXAMPLE 1.A.

```
int a=f(x)+g(x);
int b=f(g(x));
```

EXAMPLE 1.B.

```
int a=(x > 3)?f(x):g(x);
```

EXAMPLE 2.A.

```
int t1 = f(x);
int t2 = g(x);
int a = t1 + t2;
     int b = f(t2);
```

EXAMPLE 2.B.

```
int a;
if(x > 3) { a = f(x); }
else { a = g(x); }
```

Problems arise from the statements containing (as in Example 1.B) the conditional operation (… ? … : …). It is really a *"hidden" branching* of the control flow. Representing such statements in CFG by single nodes will be inappropriate. A similar problem is generated by the polymorphism of the OO-language. The natural solution of such problems is given in [12]: "… if an object reference could result in any one of several alternative methods … it is possible to represent the reference as … case statement."

Another kind of problem – *"hiding" of data dependences* – is generated by the expressions, containing operations that change the value of its argument – the incremental (++) and decremental (−−) operations, as well as operations += , −=, etc.

A possible solution of these problems is to split each statement, containing complex expressions, into a proper sequence of statements, evaluating the simple expressions incorporated in the complex one. In such a way all hidden branchings and all hidden data dependences are made explicit. Then we replace the node of each complex statement in CFG with the subgraph of the sequence of statements (Examples 2.A and 2.B). Such solution does not make the CFG too complex because the obtained not branching statements will be reduced and will not be included in the CFG.

IMPLEMENTATION OF A PATH SELECTOR

External Resources. Each application uses some external resources (including such located in a network) – different kinds of files (text files, XML files, etc.), databases, graphic interface forms, etc. Data, coming from such external resources, could have a big impact on the control flow and are significant for the path selection and constraints generation process.

If such resource or its formal specification is not available to the test generator then the information necessary for the generation of corresponding external data should be extracted from the source code. Such task is very difficult and could be an object for serious research. In our current version of the ATDG tool, if the user cannot provide formal description of an external resource the solution is to mock the objects that are dealing with this resource.

External libraries without source code. Another problem is the usage of external libraries. A project in Java can use additional libraries provided as JAR of class files

without source code. A problem arises when the generator has to create a constraint on an invocation of a method from the library without source code.

One possible solution to this problem is to execute the method and to analyze it dynamically. Another solution is to analyze the executable code. Java Debug Interface tool of the Java Platform Debugger Architecture [6] provides functionality to run and to trace program, statement by statement, and functionality to inspect the objects and their attributes on each step. Both solutions are not consistent with the project static concept.

Our solution is to apply static analysis to the "byte-code" of the corresponding external module. This is not only consistent with the concept of the project but it is an easy to implement solution, because of our existing static analysis tools.

IMPLEMENTATION OF TEST DATA GENERATOR

Compound types. One of the main problems of test data generation is that the existing solvers are not able to solve constraints, including instances of compound data types – strings, lists, trees, sets, maps, and objects. That is why we have to split each constraint involving such instances to constraints including only elementary data. We called this *atomization*. Atomization is not an easy operation because leads to NPC problem – it increases the number of constraints and makes the obtained SAT problem practically unsolvable. *Atomization in depth* (some attributes of a compound data type could be of compound types too) makes the problem even harder. Let us illustrate some dificulties in data generation of instances of compound data types.

EXAMPLE 3.

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

Given the program module from Example 3 we want to generate values for the string variables `url`, `fName` and the integer variable `minLen` that satisfy the condition of `if` operator. This condition should be converted into constraints for the SAT solver. They cannot be treated separately because their values are mutually dependant.

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 `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 – increasing the number of constraints.

A serious problem is the representation of the condition `url.contans(fileName)`. The straight forward transformation into constraints is:

```
fName.size == 1 && url[0] == fName[0] || fName.size == 2 &&
url[0] == fName[0] && url[1] == fName[1] || … ||
fName.size == k && url[0] == fName[0] &&
url[1] == fName[1] && … && url[k - 1] == fName[k - 1]
```

where *k* is a preliminary fixed constant. If we can't solve it we have to try with a bigger value of *k*. This is an example for only one of many possible conditions on strings that we have to be able to recognize and to transform to constraints.

Similar problems arise with any aggregated data type. The discussion of possible solutions is out of the scope of this paper and will be a subject of another research.

Equal references. In OO-languages one object is composed of attributes. Each attribute could be an object composed of attributes too, and so on. The treatment of the constraints, which include object variables, cannot be the same as the treatment of the constraints, including variable of primitive types. Assume we have == constraint on two object variables (i.e. they have to refer to the same object). If we have also constraints that refer to attributes of these variables, then each equivalent attributes of the two variabless have to refer to same object too. Else we could generate different values for the same attribute, which is unacceptable (See Example 4 – value 11 for `a.x` and value 4 for `b.x`).

EXAMPLE 4.

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

Suppose `a` and `b` are variables that can refer to the same object and c_1, c_2,…, c_n are their attributes. The atomization forms the constraint `((a==b)=>(a.`c_1 `== b.`c_1 `&& a.`c_2 `== b.`c_2 `&& … && a.`c_n `== b.`c_n`))`. The same has to be recursively applied to the attributes, which are of reference type too increasing enormously the number of constraints (NPC problem).

The simple substitution `a`→`b` could be a solution, but only when `(a==b)` is not combined with another constraint on the attributes (see again Example 4, when the condition of the outer `if` is `a==b || a.y>b.y`). Fortunately, this case could be eliminated during the analysis stage as a "hidden" branching.

CONCLUSION

Implementation of a complex software analyzing tool with automated test data generator, as described above, is a great challenge even when the selected approach is relatively well developed and discussed in the literature. The main problems arise from treatment of methods invocations (INV problem), especially direct or indirect recursive invocations, and the usage of objects with compound structure (COMP problem).

Some trivial solutions (e.g. mocking), even inevitable in some cases, are useless if we would like to generate a consistent test set. Unfortunately, the development of alternative solutions is not easy. The straightforward solutions lead to some of the mentioned classical problems – very big CFG, very big system of constraints or unsolvable constraints. That is why we applied some new, as far as we know, techniques during the implementation of our ATDG tool.

For INV problem ***the processing of all methods is applied instead processing of a single method***. It gives the possibility to postpone constructing of the CFG of the module to the moment when we find the first of its invocations in some other module. As a result, in our implementation the traditional first two stages of the approach – Program analysis and Path selection – have been merged in one stage.

For COMP problem an atomization of the compound objects to elementary is applied, which is not a general solution of the problem. First, because the atomization has to be used carefully in order to escape enormous growing of the number of constraints. Second, because the atomization is working well for generation of numerical data but our experience showed that it is inappropriate for generation of strings. For such case some methods of Theory of formal languages and Artificial intelligence can be applied.

As it is seen, most of the discussed problems are hard. Some of the proposed by us solutions do not solve them entirely or lead to some other of the above mentioned problems.

In order to estimate the performance of the implemented ATDG tool, some experiments were conducted. 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) were used and the results have been estimated by *EclEMMA* [3] – an automated tool for control of the test generation process. As it was reported by *EclEMMA,* we succeed to cover with tests about 80% of the classes, 50 % of the methods of uncovered classes and 35% of the lines of the uncovered methods of the tested systems. The experiments showed that there is still a field for amelioration of the proposed approaches and implementation algorithms.

ACKNOWLEDGEMENTS

This work was partially supported by the National Innovative Fund attached to the Bulgarian Ministry of Economy and Energy (project № 5*И*Ф-02-3 / 03.12.08).

REFERENCES

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

[2] J. Edvardsson, "A survey of automatic test data generation," Proceeding of Second Conference on Computer Science and Engineering in Linkoping, October 1999, pp. 21-28.

[3] EclEMMA 1.4.3: http://www.eclemma.org/

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

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

[6] Java PDA: http://java.sun.com/javase/6/docs/technotes/guides/jpda/

[7] Kr. Manev, M. Sedkov and Tz. Bogdanov. "Grading Systems for Competitions in Programming", Mathematics and Education in Mathematics, Proc. of 38-th Spring Conference of UBM, Borovetz, 2009.

[8] N. Maneva. "Main trends in software testing tools development", Journal of Automation and IT, No 3, 1987, pp. 77-84 (in Bulgarian).

[9] 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.

[10] Smart Source Analyzer (SSA),Musala Soft, Ltd.: http://www.musala.com

[11] Unit testing with mock object: http://ibm.com/developerworks/library/j-mocktest.html

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

ABOUT THE AUTHORS

Assoc. Prof. Krassimir Manev, Ph.D., Faculty of Mathematics and Computer Science, Sofia University, 5 J. Bourchier blvd., Sofia, Bulgaria, Phone: +359 2 8161530, E-mail: manev@fmi.uni-sofia.bg

Anton Zhelyazkov, Senior Software Engineer, Musala Soft Ltd., 36 Dragan Tsankov blvd., Sofia, Bulgaria +359 2 969 58 21, anton.zhelyazkov@musala.com

Stanimir Boychev, Vice-president, Musala Soft Ltd., 36 Dragan Tsankov blvd., Sofia, Bulgaria +359 2 969 58 21, stanimir.boychev@musala.com