Automatic Testing Based on Design by Contract™

Ilinca Ciupa, Andreas Leitner

ETH (Swiss Federal Institute of Technology) Zurich, Chair of Software Engineering
RZ Building, CH-8092 Zurich, Switzerland
{Ilinca.Ciupa, Andreas.Leitner}@inf.ethz.ch

Abstract. Although its importance is widely recognized, testing is seldom done properly. The reasons for this include under-allocation of resources for the testing activity, lack of proper tool support, and developers' reluctance towards testing. To tackle these issues, we propose the full automation of the testing process for contract-equipped classes. According to the principles of Design by Contract™, assertions contain the specifications of the software elements. As such, they can be used to ascertain the correctness of these elements. In this paper we discuss the various issues involved in the full automation of the testing process and present our technical solution and its implementation in the tool called AutoTest[1]. We also look at ways of improving the current approach.

1 Introduction

The importance of testing in the software development process is recognized as a matter of fact nowadays. However, practice shows that, in spite of on-going research in the field, testing methodologies and tools have not yet managed to provide software developers with adequate support for testing activities.

We suggest a different approach to testing: a fully automatic testing process. Our methodology is based on Design by Contract™. Contracts (routine preconditions and postconditions, class invariants, loop variants and invariants, and `check` instructions) are a valuable source of information regarding the intended semantics of the software. By checking that the software respects its contracts, we can ascertain its validity. Therefore, contracts provide the basis for the automation of the testing process.

The goal of our work is to develop and implement this fully automatic contract-based testing process. However, an automatic testing process is not by itself a guarantee of quality for the system under test. This process must, in turn, satisfy some high quality criteria. When the testing process becomes fully automated, there are several issues that we have to consider:

- Generation of test data;
- Tailorability of the testing process: identifying the parameters that the user must be able to tweak to adapt the process to his own needs and preferences;

[1] Previously called TestStudio

- Automatic integration of results: test results must be evaluated automatically and we must develop a classification of possible results and determine the other information to provide to the user;
- Finding failure-reproducing examples;
- Fault isolation;
- Estimation of the quality of the test suites and identifying a means to adapt test generation given such an estimation.

These are the challenges we have identified in our work. We have already found solutions to some of them and we describe these in the rest of this paper. On the others there is work in progress, as presented in section 6.

Although the idea of contract-based automatic testing is not new, our approach improves on previous and related work (as described in section 3) on three very important factors:

- It is completely automatic (requires no intervention of the user whatsoever).
- It has been applied to the programming methodology which introduced contracts in the software development cycle (Eiffel).
- It has been tested on full-fledged, industrial-scale applications.

Our approach is currently only applicable to the Eiffel language, but in the future we will extend it to other methodologies which support contracts.

We must also mention that the target of our work is developing a complete process for automatic testing, not just one or several of its separate stages. It is with regard to this that our work is fundamentally different from other approaches, which focus on automating only certain aspects and do not deal at all with the others, as we show in section 3.

In this paper we describe the complete testing process we have developed. Space constraints do not allow us to go into detail on all points presented, but we prefer to give an overview of the approach as a whole and of the way the different stages are integrated into a seamless process. The next section outlines the most important results we have obtained so far. Section 3 describes related work and in section 4 we provide an overview of the proposed testing process. Section 5 contains a detailed discussion of the results. We outline the directions of our future research in section 6. Finally, we draw some conclusions in section 7.

2 Main Results

The tool we have built to put our ideas into practice and to test our approach is called AutoTest[2] [6, 8, 11]. It implements the fully automatic testing process that we have developed. Using AutoTest, we ran tests on some Eiffel libraries. Although these libraries had been thoroughly tested by developers, our tool was able to find several bugs in them, with various degrees of severity. We give examples of such bugs in section 5. This work has helped us identify several reasons why bugs had escaped manual testing:

[2] Available at http://se.inf.ethz.ch/people/leitner/test_studio/

- Numerous bugs are exposed when testing with extreme values (at either end of the valid interval) as inputs.
- Other bugs were related to insufficient specification of routines' preconditions. The developers had made some implicit assumptions about the use of those routines, but did not state these assumptions in preconditions.
- Developers had tested certain routines only as part of specific usage scenarios. Running those routines outside the intended context (when their export status allowed it) revealed errors.

Although some of these causes (or closely related ones) have already been identified in the literature, we want to highlight here that our experience has confirmed them and that automatic testing suffers from none of these disadvantages.

Although the tool is not fully implemented yet and several aspects are still the subject of ongoing research, the results we have obtained so far are very encouraging. They prove the value of the approach and the effectiveness of fully automatic testing.

3 Related Work

The results published in several papers support the idea of using contracts for testing. [5] tries to estimate the improvement in the fault isolation effort that contracts bring. The authors use the notion of diagnosability to illustrate this concept: "diagnosability captures the fault isolation effort after the occurrence of a failure". The results presented in the paper show that diagnosability improves considerably (about 8 times) in the presence of contracts defined with very high precision. [17] investigates the question of which types of assertions are most effective at detecting faults. The author develops a classification of assertions from this perspective. This classification is meant as a first step in creating a methodology of programming with assertions.

Researchers have dedicated a large amount of work to automatic testing. The idea of using contracts as built-in test (as discussed in [3]) has been exploited as part of several approaches. [4] presents a method that is very similar to ours: the testing tool called Korat uses preconditions to generate valid and non-isomorphic inputs for each method under test and postconditions to evaluate the correctness of the methods. They place special emphasis on the technique of generating inputs based on parsing the method preconditions and on a user-provided finitization function on the inputs. Korat can automatically generate test cases for Java programs with associated JML [10] assertions. The disadvantage of this approach is that it is not fully automatic: although a skeleton for the finitization function is generated by the tool, in many cases users are required to edit it to get the behavior they want.

Various research groups have investigated the idea of using specifications for testing. [16] is the original paper formalizing specification-based testing. They propose the extension of structural testing with specification-based techniques. Test cases contain preconditions and postconditions. Two of the approaches they develop (specification/error-based testing and specification/fault-based testing) assume that the specification itself may be faulty. The other two approaches (oracle/error-based testing and oracle/fault-based testing) treat the specification as an oracle, as is the case

in our work. However, this is the only similarity between the work described in [16] and ours, as they only address the issue of automating the test oracle.

[12] also uses method preconditions and postconditions to model the functional correctness of systems. The author views testing as a constraint solving or satisfiability problem: determining values for the input variables so that the system under test terminates and the conjunction between the precondition and the negated postcondition is satisfied. The approach is based on building an approximate model of the system by running several tests on it and then using this knowledge to estimate where a successful test case might be found. The idea is interesting, but evaluation of the performance of the algorithm was still in progress when the paper was written. The author provides only a simple case study, from which it is difficult to make any estimation about the general performance of the solution.

[9] addresses the question of evaluating test quality. The authors focus on building trust into components by associating tests with them and evaluating the quality of these tests. The authors measure the quality of a test sequence by injecting faults in the components and evaluating the fault revealing power of the tests. In this approach, the user has to manually write the test code; class invariants and method postconditions are used as oracles. The focus of this paper is estimating test quality, not automating the testing process. As such, this work is complementary to ours.

Several approaches focus on automating only certain aspects of testing. The methodology described in [1] requires the user to write a formal test specification and from this specification are generated test cases and test scripts. [18] focuses on automatic generation of test cases: they are represented as chromosomes and passed to a genetic algorithm, which mutates them to maximize coverage.

4 Our Approach to Automatic Testing

A testing process requires the following steps:

- Determining the test scope
- Generation of input data and establishing the conditions (the state of the system) under which the test case should be executed
- Generation and compilation of the test code (the test cases and the test driver)
- Running the test code and recording results
- Interpretation of the test results

When the testing process becomes automatic, all these individual stages must be performed without human intervention and they must be integrated automatically to form a continuous flow. The user is only involved at the beginning of the process, when he decides on the elements under test, and at the end, when he views the results.

In the following subsections we describe our solutions for automating each of these stages, for integrating them into a seamless process, and our test configuration mechanism.

4.1 The test scope

For convenience, our tool allows users to set the test scope at various degrees of granularity: cluster[3], class, or feature[4]. Any choice is reduced to the features contained in the tested elements, and it is at the feature level that we perform our tests. However, there are two issues to be considered when a user wants to test feature `f` of class `A`:

- Do we also test `f` in descendants of `A`? Because of polymorphism this is a valid concern.
- If class `A` is generic, which instantiations of the generic parameter(s) do we use to run the tests?

In the case of the first question, if `A` is deferred (abstract), then the answer must be yes. Still, even in this case we have to ask if we stop at the first descendant that is effective (concrete) or also recursively test its descendants. The default that our tool offers is testing only `A` if it is effective and testing only its first level of effective descendants if it is deferred, but users can change these settings.

The second question also arises when we have to instantiate feature arguments that take generic parameters. Again, the tool provides a default (instantiating the constraining type only), but users can change this behavior to using instances of:

- The basic types if they conform to the constraining type.
- All types in the system that conform to the constraining type.

4.2 Generation of input data

When testing a feature of a class, we need an instance of the current class and instances of the arguments. Our current algorithm for input data generation uses a random strategy: when an object of a certain type is needed, it is created by randomly calling one of the creation procedures of its class and it is added to the pool of available objects. Then a random modifier (routine with side-effects) is called on an object in the pool (to diversify it) and an object is randomly selected from the pool.

We designed this algorithm as a first and temporary solution to the issue of test data generation. In spite of its obvious drawbacks (one of which is that it does not take into account any measure of code or data coverage), when integrated into the testing process it produces surprisingly good results, as shown in section 5. Nevertheless, improving our strategy for data generation is one of the main targets of our future work.

4.3 Conditions for executing a test case

As our current approach is targeted at testing individual features, the conditions for executing a certain test case reduce to fulfilling the precondition of the tested routine. A major drawback of our random strategy for generating input values is that it does

[3] A cluster is a means of grouping several classes that usually have a high degree of coupling.
[4] A feature is, in Eiffel terminology, an attribute or method of a class.

not take the precondition into account. Objects are generated randomly and therefore for some routines we might be unable to fulfill the preconditions. However, there is ongoing work in our project which focuses on using planning to obtain objects that satisfy routines' preconditions. We do not describe this work in the present paper.

4.4 Generation, compilation and running of the test code

Currently we generate a class for every feature under test and every other feature that may be needed. The test code is then compiled and for execution AutoTest uses a two-tier approach: a test driver (master) and an interpreter (slave). The driver tells the interpreter which routine must be tested and provides it the identifiers of the slots in the object pool where the target object and the arguments are located. The interpreter is responsible only for running the tested routine and for returning the result to the test driver. The advantage of this approach is that test running does not have to start over for the whole system if the execution of one routine fails. In such a case, only the interpreter will be affected and the driver, seeing that it receives no response or an unexpected one, will reinitiate the interpreter. If several calls to the same routine fail (the interpreter does not respond), then the driver will decide to skip testing that particular routine and go on to the next.

4.5 Interpretation of the test results

Our tool provides the following information about the outcome of running the tests:

- A classification of the test result:
  - Failed - the feature produced an assertion violation or some other exception occurred during its execution;
  - Could not be tested – AutoTest could issue no call to the feature, because it could not instantiate the target object or one of the arguments;
  - No call was valid – no call to the feature fulfilled its precondition, so the feature does not fail the test but does not pass either; it simply was not actually tested;
  - Passed – at least one of the feature calls succeeded (fulfilled the precondition) and its execution did not produce any exceptions or assertion violations.
- The numbers of calls for which:
  - There was no exception
  - The test case violated the precondition of the routine under test
  - There occurred an assertion violation
  - There was some other exception

Evaluation of test results is straightforward due to the presence of contracts in the code: we check the routine preconditions and postconditions, class invariants and other assertions present in the routine body (loop assertions and `check` instructions); if any test execution violates any of these assertions (except when the test case directly violates a routine's precondition), we have found a bug and we report it to the user, indicating the routine where the violation occurred and the tag of the assertion. We also monitor other exceptions triggered during test execution and report them to the user separately.

The cases mentioned above do not account for all possible bugs that may be present in the code. An example is that of infinite loops. Currently, we try to avoid them by using a timer; if the execution of the routine does not finish within the preset interval, we report an infinite loop. The evident drawback of this technique is that some correct executions which just require more time than our limit will be categorized as infinite loops. However, we must mention that this is the only case of false alarm we have identified. Our goal is to minimize such false positives and at the same time identify all real positives.

4.6 Configuring the testing process

In fully automatic testing there arises the need for mechanisms for tuning the process. Users must have means by which to adapt it to their particular requirements. Therefore, we have introduced a set of parameters which lets users configure the testing process:

- The stress level associated with the various elements under test (this is currently mapped to the number of calls to tested features and whether or not we test features in descendants of a class)
- The level of assertion checking: what type of assertions (preconditions, postconditions, class invariants, loop variants and invariants, and `check` instructions) are monitored
- The testing order:
  - run all tests on a feature before testing the next one
  - test all features of a class before proceeding to the next one
  - execute each feature under test once before calling them a second time
- The level of support for genericity (determines the types used for instantiating generic parameters)
- The values used for instantiating basic types (`INTEGER`, `REAL`, `DOUBLE`, `BOOLEAN`, `CHARACTER`, `STRING`)
- Instances for other types that will be part of the context[5]

All these parameters have defaults, but users can also change their values.

5 Discussion of Results

One of the biggest advantages of our approach is that we were able to test it on existing large-scale applications and libraries. This is not the case for any of the related work (mentioned in section 3), because it is targeted at programming languages that do not natively incorporate assertions. Since assertions are added artificially on top of existing programming languages and they are mostly used in the research community, there are no real life applications on which the proposed methods were tested.

[5] The context is the pool of objects from which instances are selected at test execution time.

```
is_inserted (v: G): BOOLEAN is
              -- Has `v' been inserted at the end by
the
              -- most recent `put' or
              -- `extend'?
             do
              check
                put_constraint: (v /= last) implies
not off
              end
              Result := (v = last) or else (v =
item)
             end
```

Listing 1: Function `is_inserted` of class `ARRAYED_LIST [G]`

To give the reader an idea of the testing capabilities of our tool, we provide the following examples: in the test of class `ARRAYED_LIST [G]` from cluster `list` of the EiffelBase library, tests failed for 13 features, out of a total of 98 exported features that the class has; when testing class `STRING` from cluster `kernel_classic` of the EiffelBase library, tests failed for 25 features out of 157 exported features.

We present in the following a few examples of bugs we identified. When testing class `ARRAYED_LIST [G]` we noticed that all calls to feature `is_inserted` (whose code is shown in Listing 1) failed and all because of the same reason: the `check` instruction at the beginning of the routine body failed. The header comment of the routine points us to the most probable cause of the bug: the developer of the class intended for `is_inserted` to only be called as part of the postconditions of routines `put` (and its variants `sequence_put` and `bag_put`) and `extend`. However, the export status of `is_inserted` allows it to be called by any client of class `ARRAYED_LIST [G]`, and in this case the condition contained in the `check` instruction does not make sense anymore. It is very likely that this bug escaped manual testing because the developers had a specific usage scenario in mind and they only tested the routine as part of that scenario.

Another feature of `ARRAYED_LIST [G]` for which our tests failed was `put` (see code in Listing 2). The precondition of this feature only requires that our instance of `ARRAYED_LIST [G]` is extendible, and it checks this by calling the `extendible` feature, which returns a boolean value. Class `ARRAYED_LIST [G]` inherits this feature from `DYNAMIC_CHAIN`, in which it is defined as a constant and its value is set to `True` (as shown in Listing 2). Therefore, routine `put` has the weakest possible precondition (`True`). Its implementation does nothing else than call `replace`, but `replace` requires that the target object is writable (meaning that the current index has a valid value, i.e. it points to an element in the list, not before its

beginning or after its end). Therefore, when calling `put` on an instance of `ARRAYED_LIST [G]` whose index is 0 and minimum valid index is 1, the precondition of `replace` is violated. The obvious problem here is the lack of a precondition for `put`. Actually, since all `put` does is call `replace` (and, as stated in the header comment of `put`, it should be a synonym of `replace`), the two routines should have the same preconditions.

```
put (v: like item) is
            -- Replace current item by `v'.
            -- (Synonym for `replace')
            -- (from CHAIN)
          require -- from COLLECTION
            extendible: extendible
          do
            replace (v)
          ensure -- from COLLECTION
            item_inserted: is_inserted (v)
          ensure then -- from CHAIN
            same_count: count = old count
          end
replace (v: like first) is
            -- Replace current item by `v'.
          require -- from ACTIVE
            writable: writable
          do
            put_i_th (v, index)
          ensure -- from ACTIVE
            item_replaced: item = v
          end
Extendible: BOOLEAN is True
            -- (from DYNAMIC_CHAIN)
```

Listing 2: Features `put`, `replace` and `Extendible` of class `ARRAYED_LIST [G]`

Another category of bugs we found were revealed by testing routines with extreme values for parameters, such as the minimum and maximum values for integers.

Creation procedure `make` of class `ARRAY [G]` (whose code is shown in Listing 3) takes 2 parameters: a minimum index and a maximum index, both integers, and it should allocate an array whose indexes vary between the given values. When testing this feature, our tool tried to call it with the values -2147483648 (the minimum value for integers) and 10, respectively. The call was valid because these values fulfill the precondition of the routine. However, when the execution reached the call to `make_area (max_index - min_index + 1)` the precondition of routine `make_area` of class `TO_SPECIAL [T]` was violated, as it was called with argument -2147483639. What happened was that the calculation of the expression `max_index - min_index + 1` produced a number that was greater than the maximum value for integers, so it was interpreted as a negative number. However, as a client of feature `make` of class `ARRAY [G]`, our test fulfilled its precondition, so it was expecting `make` to fulfill the postcondition in return. As this was not the case, we conclude that we have found a bug. A possible correction for it would be to change the precondition of `make` to `max_index - min_index + 1 >= 0`. As the example shows, this is not the same as evaluating its current precondition (`min_index <= max_index + 1`) and it would ensure that the precondition of `make_area` is fulfilled.

```
class ARRAY [G]
…
make (min_index, max_index: INTEGER) is
           -- Allocate array; set index interval to
           -- `min_index' .. `max_index'; set all
values to default.
           -- (Make array empty if `min_index' =
`max_index' + 1).
         require
           valid_bounds: min_index <= max_index + 1
         do
           lower := min_index
           upper := max_index
           if min_index <= max_index then
             make_area (max_index - min_index + 1)
           else
             make_area (0)
           end
         ensure
           lower_set: lower = min_index
```

```
            upper_set: upper = max_index
            items_set: all_default
          end
…
end

class TO_SPECIAL [T]
…
make_area (n: INTEGER) is
            -- Creates a special object for `n'
entries.
          require
            non_negative_argument: n >= 0
          do
            create area.make (n)
          ensure
            area_allocated: area /= Void and then
area.count = n
          end
```

Listing 3: Feature `make` of class `ARRAY [G]` and feature `make_area` of class `TO_SPECIAL [T]`

6 Future Work

One direction for our future research that has already been mentioned concerns the generation of input values for the tests. Currently this is done randomly, but we intend to also implement a measure of data coverage, which is strongly linked to code coverage. This development will also mean changing our test strategy from black-box to white-box. Currently we use no knowledge about the internal structure of the code, only its contracts. Any coverage solution has several parts: determining the actual coverage of the tests; feeding this measure back to the object generator, so that it will give us values which will increase coverage; determining the value of the coverage that we are aiming for and stopping when our tests reach that value. Naturally, this technique will only be applicable to systems for which we have the source code. For the systems for which we only have a binary form, we will not be able to perform any white-box testing.

In section 1 we identified finding failure-reproducing examples as an important issue involved in fully automatic testing. In our approach, such examples are already available when the execution of the tests ends (provided that at least one test fails). However, the size of these examples might not be optimal. We want to develop a method which, starting from these existing examples, trims them down to the minimal number of calls necessary to reproduce the failure.

As we use contracts to evaluate test results, the quality of the test oracles is entirely dependent on the quality of the contracts. We will not be able to check any part of the functionality of the system that is not specified in its contracts. Consequently, we will try to develop a methodology of writing contracts with respect to their use as test oracles.

When developers are introduced to Design by Contract™, they are at first reluctant to using contracts to specify the requirements and output of their systems. A great number of software developers novice in the Eiffel method fail to see the benefits that contracts bring. We believe that the use of contracts for testing can change this. We intend to conduct a survey to determine if the advantages of automatic testing using contracts do indeed increase their use and quality.

One further direction of research concerns extending our approach to other programming languages and methodologies which support contracts. We believe that our method is general enough to allow this extension. Still, some details will need to be adapted according to the characteristics of each language regarding issues like inheritance of assertions or assertion actions (triggered when the condition is not fulfilled).

7 Conclusions

We have described our solution for developing and implementing a fully automatic testing process. We use contracts as the basis of our approach. By checking that a software element fulfills its contracts, we can determine if it meets its specification. We have also discussed our solutions to other issues involved in automatic testing, such as data generation, test execution, interpretation of results, and tailorability of the testing process.

As demonstrated by the results, our current approach and its implementation are promising. The AutoTest tool is proof of concept for push-button testing. It implements a completely automatic testing process, by which we have been able to identify a great number of bugs in Eiffel libraries.

Acknowledgments

We wish to thank Prof. Bertrand Meyer whose ideas have greatly influenced and contributed to the work presented in this paper. We also want to thank Dr. Karine Arnout, Xavier Rousselot, and Nicole Greber, who have provided innovative ideas, feedback and implementation work to the research described here.

References:

1. Baker, M. J., Hasling,W. M., and Ostrand, T. J. Automatic generation of test scripts from formal test specifications. In Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification, ACM Press, New York, NY (1989) 210-218
2. Bezault, E. *Gobo Eiffel Project*, retrieved November 2004 from http://www.gobosoft.com
3. Binder, R. V.: Testing Object-Oriented System. Models, Patterns, and Tools. Addison-Wesley (1999)
4. Boyapati, C., Khurshid, S. and Marinov, D. Korat: Automated Testing Based on Java Predicates. In ACM SIGSOFT Software Engineering Notes, 27, 4, ACM Press, New York, NY (July 2002) 123-133
5. Briand, L. C., Labiche, Y. and Sun, H. Investigating the Use of Analysis Contracts to Support Fault Isolation in Object Oriented Code. In Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '02). ACM Press, New York, NY (2002) 70-80
6. Ciupa, I. TestStudio: An Environment for Automatic Test Generation Based on Design by Contract™, Diploma Thesis, ETH Zurich (Swiss Federal Institute of Technology Zurich), 2004
7. Cousot, P., and Cousot, R. Abstract Interpretation Based Program Testing. In Proceedings of the SSGRR 2000 Computer & eBusiness International Conference (SSGRR 2000), CD ROM paper 248, Scuola Superiore G. Reiss Romoli, 2000.
8. Greber, N. Test Wizard: Automatic test generation based on Design by Contract, Master Thesis, ETH Zurich (Swiss Federal Institute of Technology Zurich), 2004
9. Jezequel, J. M., Deveaux, D. and Le Traon, Y. Reliable Objects: a Lightweight Approach Applied to Java. In IEEE Software, 18, 4, (July/August 2001) 76-83
10. Leavens, G. T., Baker, A. L. and Ruby, C. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998. (last revision: Aug 2001)
11. Leitner, A. Strategies to Automatically Test Eiffel Programs, Master Thesis, Technical University of Graz, 2005
12. Meinke, K. Automated Black-Box Testing of Functional Correctness Using Function Approximation. In Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '04), ACM Press, New York, NY (2004) 143-153
13. Meyer, B.: The Grand Challenge of Trusted Components. In Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA (2003) 660-667
14. Meyer, B.: Object-Oriented Software Construction, 2nd edition. Prentice Hall (1997)
15. Meyer, B., Mingins C., Schmidt, H.: Trusted Components for the Software Industry, available at http://www.trusted-components.org/documents/tc_original_paper.html
16. Richardson, D. J., Owen O'Malley, O. and Tittle, C. Approaches to Specification-Based Testing. In Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification. ACM Press, New York, NY (1989) 86-96
17. Rosenblum, D. S. A Practical Approach to Programming with Assertions. In IEEE Transactions on Software Engineering, 21, 1 (Jan. 1995) 19-31 (minor correction published in vol. 21, no. 3, Mar. 1995, p. 265)
18. Tonella, P. Evolutionary Testing of Classes. In Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '04), ACM Press, New York, NY (2004) 119-128