

APPLICATION OF THE TEST IN TESTING PROGRAM

Branko Markoski*, Predrag Pecev **, Zdravko Ivanković*, I. Setrajcic **, J. Setrajcic ***, Stamatovski Antonio****

*University of Novi Sad, Technical Faculty "Mihajlo Pupin", Zrenjanin, Serbia
markonins@yahoo.com, ivankovic.zdravko@gmail.com

** University of Novi Sad, Faculty of Sciences, Dept. of Mathematics and Computers Sciences, Novi Sad, Serbia
predrag.pecev@gmail.com, seki_1976@yahoo.com

***University of Novi Sad, Department of Physics, Faculty of Sciences, Novi Sad, Serbia, bora@df.uns.ac.rs

****University "St. Kliment Ohridski"- Ohrid, Faculty of Technical Sciences, Bitola, Republic of Macedonia (FYROM)

Abstract - Within software's life cycle, program testing is very important, since quality of specification demands, design and application must be proven. Testing of large and complicated programs must be done as systematically as possible, in order to obtain reliability. In case of large and complex systems and their operating systems ad hoc testing is used, which often could not prove quality or validity according to specification, construction or application. Validation and verification are terms often connected to program testing. Verification is checkup of testing of objects (or programs) in order to determine are they in accordance with specifications. Verification contains analysis, inspection, trying, as well as testing of program. About testing the software, ordinarily we do statically analyses (exploring of basic programs, searching for primary problems and collecting data's without executing the program) and dynamic analyses (exploring behavior of program in executing, so we acquire the data about the ways of executing, chronological sections and integrity of testing). Every company, which educes the software, is performing tests of their products, and the software from market usually contents complex variants of defects. Sometimes it is difficult to understand how it is possible that the test omits so obvious error.

Key words: verification, validation, test information, specification, testing programs

I. INTRODUCTION

One of the most important aspects of the projects for software development is the strategy of integration. The integration can be performed at once, from the top to the bottom, from the bottom to the top, critical part first or with the first functional subsystem of integration and only then to integrate the subsystems in separate phases using any kind of basic strategy. In general, if the projects are bigger, the strategy of integration becomes more important.

Very small systems are often collected and tested in one phase. For the majority of real systems, this is not practical for two important reasons. The first is that the system would failed in many places at once and the attempts to debug and to re-test would be completely impractical [PRESSMAN]. The second is that the correspondence to the testing criteria of white box would be very difficult, for the big quantity of details separating the entrance data banks from the individual ciphers of the modules. In fact, the majority of integration testing is

traditionally limited to the techniques of "black box". The big systems can demand many phases of integration, beginning with the collection of modules in low-ranged subsystems, and then the gathering of the subsystems into bigger subsystems and finally the composition of the subsystems of higher level into the whole system.

In order to make the strategy of integration the most efficacious one, the techniques of integration testing have to go well with the whole strategy of integration. In the poly-phase integration the testing in every phase helps to discover the error before, and to keep the system under control. With the execution of the glance testing in the early phase of integration, and then with the application of rigorous criteria it is real only the big risk variant of the "big bang" approach. Nevertheless, the execution of rigorous testing of the whole software engaged in every phase of integration engages a lot of unnecessary effort doubled through phases. The solution is to overcome the complete integration systems, to execute rigorous testing in every phase and to reduce the effort doubling.

It is important to understand the relations between the testing modules and the integration system. On the one hand, the modules were tested with the use of drivers before any integration was tried. Then the integration system completely concentrates on the modules interaction supposing that the details in every module are correct. On the other, the modules and the integration testing can be combined, verifying the details of execution of each module in the integration context. Many projects compromise with the combination of testing modules with the lowest level of subsystem integration testing, and then execute the pure integration testing on higher level. The two aspects related to integration testing can be convenient for any kind of task project and the integration testing method has to be sufficiently flexible in order to be adequate for all of them. The rest of this section explains the techniques of equal integration structure testing, initially only in special cases and then completely.

II. WHEN THE PROGRAM TESTING IS TO BE STOPPED?

This is the most frequent question encountered by testers. Here are some possible answers:

- When you don't have time,
- When further testing provokes new denials,

- When further testing does not discover new errors,
- When you cannot create any new testing item,
- When you arrive to the point in which reduces the number of responses,
- When the requested covering is reached,
- When all the errors are eliminated.

Unfortunately, the first answer is the most frequent one, and the seventh cannot be guaranteed by anyone. This leaves the tester somewhere in the middle. The models of software reliability offer the solutions that support the second and the third answer, both of them are largely used in the industry. The fourth answer is insecure: if you followed the procedures and the instructions that we talked about, this is probably the good solution. On the other hand, if the reason is lack of motivation, this solution is unfortunate as much as the first one. The fifth solution is attractive: implies the continuation of serious testing, but the discovering of new errors reduces dramatically. The further testing becomes very expensive and maybe it will not be discovered any new error. If the costs (or risk) of the rest of the errors can be defined, the advantage is clear.

III. GENERALIYATION OF MODULE TESTING CRITERIA

The module testing criteria can often be generalized in some possible ways. As discussed above, the most frequent generalization is to correspond to the module testing criteria in the context of integration, using the whole program as the environment for testing of the drivers for every module. But, this trivial generalization does not use the difference between the module and the integration testing. The application on each phase of the poly-phase strategy of integration, for example, leads to excessive number of unnecessary testing.

More than the testing, separately, in the module, of all outs chosen on purpose, the structure testing on integration level focuses the purposes of the outs that are activated with the call of the module[MCCABE]. The design of the technique of reduction helps those outs chosen on purpose in the way that becomes possible to exercise them separately during the integration testing. The idea related to the designing of reduction is that it begins with the control of the course of the graph of the module, eliminates all structure controls that are not activated with the call of the module and then it has to use “reduced” course of the graph related to the integration testing. Although, but it is not obligatory, the rule of reduction, i.e. the rule of the *call* says that the function of the call (“black point”) of the knot cannot be reduced. The rest of the rule works together in order to eliminate the parts of the course of the graph that is not activated with the call of the module. *The deriving rule* eliminates the result of non called (“white point”) knots because the application of this rule eliminates one knot and one edge from the course of the graph and the cycle complexity remains unchanged. But, this generalizes the graph in the way that the rest of the rules can be applied. The rule of *repeating* eliminates the top-test of the rings that are not

involved in the call of the module. The rule of *conditioning* eliminates the declaration of conditionings that do not contain the call in their bodies. The rule of *twisting* eliminates the bottom-test that is not activated with the call of the module

IV. EFFICACIOUSNESS OF TESTING

The thing that we would certainly want to know about the sequence of testing items is how efficacious they are, but it is necessary to clear what does the “efficaciousness” mean. The easiest way is to be dogmatic: to define the method, to use it for the testing items generating and then to execute the testing items. But this can be corrected if we reduce the dogmatism and if we demand that the testers choose “appropriate methods”. We can get even bigger improvement if we compose appropriate hybrid methods.

The structure testing techniques give also another choice for the testing efficaciousness. We will be able to examine the sequence of testing items in the sense of ways that are passed in execution. When the certain way is passed more than once, we can talk about the redundancy. Sometimes the redundancies have also the purpose.

The best interpretation of the testing efficaciousness is (and this is not a miracle) the more difficult one. In fact, we want to know how much the sequence of testing items is efficacious in discovering errors in the program. This is problematic for the two reasons: the first it that it is supposed that we know all errors in the program. But this is moving into the same cycle: if we knew them, we would correct them. Since we don’t know all errors in the program, we will not know, maybe never, if the testing items, on the grounds of the given method, succeeded in discovering them. The second reason is more theoretic: demonstration that the program is without errors corresponds to the famous problem of stopping from the computer science, for which it is known that it doesn’t have the solution. The best think we can do is to go back, from the types of errors. When we have certain type of error, we can choose the testing method (functional or structural) and it is most probable that it will discover the errors of that type. If we connect it with the knowledge related to most probable types of errors, we will get the pragmatic approach to the testing efficaciousness. This implements furthermore if we follow the types (and the frequency) of errors in the software that we develop.

There must be test set chosen with every physical input and test set provoking simulation of every interface control. One more criterion is very interesting. Many authors, Cukarelas, Gerogianis, Ekonomides [9], call it a discrimination criterion. It demands random selection of input sequences until statistical representation of whole endless domain is obtained. Third stage is execution and evaluation of test scenario. Here we have two directions: built-in test program and formalism. In principle, every program should have two tests: one is hidden and not available to users, and other is visible. Some programs may contain self-testing programs also. Formalism mostly includes hard work at formalization of ways in which specifications are written. Structurally and object oriented programming both contains mechanisms for formal

expression of specifications, in order to simplify comparison between expected and real behavior. The way in which program responds to different input data is regulated by specification. For example, notions "input" and "output" may be regarded in widest sense if input values are noted with x and output values with y ; then specification may be understood as a relation connecting input set with output set. Concerning specification, it is not necessary to be even determined; meaning that numerical program may give results with different precision in different computers.

$$[S] \subseteq X \times Y, \quad X \equiv \text{dom}([S]) \quad (1)$$

If the specification S is noted as a relation $[S]$, and X is input set and Y output set, then $[S]$ is a sub-set of

Descartes product of sets X and Y , and set X is called domain of specification. Program is modeling by so-called programming function which copies set of inputs X into set of outputs Y , but now as a function and not as any relation:

$$[P]: X \rightarrow Y \quad (2)$$

where:

P is program, $[P]$ is notation of program function, $X \equiv \text{dom}([S])$ presents set of inputs for which a program terminates (as in endless loop). Program P is correct if:

$\text{dom}([P])[S] = \text{dom}([S])$, or if for every input

V. CONCLUSION

Software producers would like to anticipate the number of errors in software systems before their application in order to estimate the quality of acquired program and the difficulties in the maintenance. This work gives the summary and describes the process of program testing, the problems that are to be resolved by testers and some solutions for the efficacious elimination of errors. The testing of big and complex programs is in general the complicated process that has to be realized as systematically as possible, in order to provide adequate confidence and to confirm the quality of given application.

The testing activity shows if the given software is harmonized with the specification. The specification is key thing in testing. So, as the results of testing are collected, the proofs about quality level and program reliability appear. If the testing often discovers the important errors, the quality and the reliability of the program can be considered insufficient and the further testing is necessary. On the other hand, if the errors are minor and easy to be corrected, then the level of quality and reliability is acceptable. The testing cannot say definitively if the program is correct, because the not discovered errors can remain in the program even after the most voluminous testing. So, the usual point of view considers successful the testing that does not discover incorrectly any error and this is underlined with the following testing purposes:

- the testing is the process of program execution in order to find out the errors;

- the good testing item has the high possibility of covering the error;
- successful testing discovers the error that until then was not discovered.

The program testing is often identified with the discovering of any kind of errors. There is no sense to test errors that most probably do not exist. It is much more efficacious to think well about the types of errors that are most probable (or provoke the biggest damages) and then to choose the testing methods that will certainly be able to discover this kind of errors. The success of one set of testing data corresponds to the successful execution of detailed program testing. One of the main questions that appears in program testing is the reproduction of the error (the testers discover the errors and the programmers eliminate the bugs). It is evident that the coordination between the testers and the programmers should exist. The error reproduction is the case when the best thing to do would be to re-execute the problematic testing so that we know when and where exactly the error occurred. So, the ideal testing and the ideal product do not exist.

Lot of effort has recently been made in order to apply the constructive approach, if it is not possible completely, then partially, i.e. on certain parts of the program.

REFERENCES

- [1] Pressman R.S., "Software Engineering "A Practitioner's Approach", McGraw-Hill, New York, 1992
- [2] Gutjahr W., "Partition vs. Random Testing: The Influence of Uncertainty," IEEE Trans. Software Eng., Vol. 25, No. 5, 1999, pp.661-674
- [3] McCabe, Thomas J. & Butler, Charles W. "Design Complexity Measurement and Testing " Communications of the ACM 32, 12 (December 1989):1415-1425
- [4] Markoski B., Hotomski P., Malbaški D., " Symbolic Execution in program testing ", International 4E ZEMAK symposium, Struga 2002.,FR. Macedonia
- [5] Gabodi G. P. Camurati, Lavagno L., Quer S., "Disjunctive partitioning and partial iterative squaring: An effective approach for symbolic traversal of large circuits ". In 34th Design Automation Conference proceedings 1997, pages 728-733, Anaheim, Ca, USA, June 1997.ACM
- [6] D. Ada, " Software testing and software development lifecycles ", IEEE Transactions of Software Engineering
- [7] Radulovic B., Hotomski P., " Projecting Deductive Databases with CWA Management I Baselog system" , Novi Sad J. Math Vol.30, No 2, 2000, pp. 133-14
- [8] Hotomski P., Berković I., " Symbolical execution of Pascal programs in theorem prover of Baselog system, ", Tara 2002.
- [9] Gerogiannis V.S, Economide K.D., Cukarelas A., " Systematically Testing a real Time Operating system ", IEEE Trans. Software Eng., Vol. 15, No. 5, October 1995, pp. 50-60.
- [10] Carver, D., "Producing Maintainable Software," Computers and Industrial Engineering, April 1987
- [11] Chidamber, S. and C. Kemerer, " Towards a Metrics Suite for Object Oriented Design," Proceedings of OOPSLA, July 1991
- [12] Coleman, D. and D. Ash and B. Lowther and P. Oman, "Using Metrics to Evaluate Software System Maintainability," IEEE Computer, August 1994