Test Suite Reduction for Regression Testing of Simple Interactions between Two Software Modules

© Dmitry Kichigin

dkichigin@gmail.com

Abstract

This paper presents a new test suite reduction technique for regression testing of simple interactions between two software modules. The idea of the technique consists in building models of interactions between two modules and using those models for the test suite reduction. Interaction models are built using sequences of interface functions, invoked during software execution.

1 Introduction

Increasing level of software modularity makes integration and regression integration testing to be a very important step in software development and maintenance process. At the same time, high rate of software evolution, and, especially, a big number and high frequency of its modifications, significantly increase the cost of regression integration testing. Test suite reduction [13, 15, 7] is one of solutions to this problem.

Test suite reduction seeks to reduce the number of test cases in a test suite while retaining a high percentage of the original suite's fault detection effectiveness [13]. Such a reduction lowers the costs of a software execution on the tests as well as the costs of a test suite maintenance [13], thereby reducing the overall cost of regression testing.

A traditional approach to test suite reduction was initially developed for unit testing and is based on filtering individual tests out of a test suite while preserving its adequacy level (in terms of selected adequacy criteria) and involves a static analysis and/or instrumentation of a source code. This approach usually consists of three main steps. On the first step, the software code is instrumented. On the second step, the software is executed on the original test suite and the software coverage information is collected. On the third step, the coverage information is analysed and, based on this, the original test suite is reduced [13]. In case of integration testing, the majority of test suite reduction methods are an adaptation of existing methods for unit testing. The main difference is that they consider coverage of not all code but the part of it, which is responsible for interactions of software elements being integrated.

Recent trends in software development, such as component-based software development and use of commercial off-the-shelf (COTS) components, and increasing size and complexity of the software being developed, present new challenges for existing test suite reduction techniques that may limit their applicability. The source code is usually unavailable for COTS components, thus making it impossible to use methods which are based on analysis or instrumentation of a source code [13]. It is also can be very expensive and hard to conduct source-code based coverage analysis for large software [11, 14]. An instrumentation of source code can be nicely done with a few modules, but the instrumentation of large-scale software is far more complex and time consuming [11]. These problems limit the applicability of existing test suite reduction techniques, thus exposing the need for new methods which are able to work without access to a source code.

To solve this problem, we present a new test suite reduction technique for regression integration testing which is based on analysis of interactions between modules being integrated and uses sequences of interface functions, invoked during software execution. In this paper, we consider simple interactions between two modules. We define simple interactions as interactions via interface of functions which can take only scalar parameters. Our method does not require an access or instrumentation of a source code and that significantly increases the area of its applicability.

The paper is organised in the following way. In Section 2, we survey existing test suite reduction methods for regression integration testing. Section 3 presents our test suite reduction method. Section 4 contains a detailed description of our experiment and results, and Section 5 summarizes our research.

2 Related work

A traditional approach to the test suit reduction consists in building a test suite of a smaller size but equivalent to the original one in terms of a selected coverage metric [10]. For integration testing, the coverage metric demonstrates the coverage of program structural elements involved into interactions being tested. Existing test suite reduction methods for integration testing can be split into two main groups depending on a type of a coverage metric they use.

2.1 Mutation analysis

Mutation analysis was initially developed as a technique to measure test suite adequacy for unit testing. The technique is based on automated seeding of faults into the original program and examining whether these faults are discovered by the test suite or not [17]. Later, mutation analysis was adapted for the integration testing. This adaptation got a name *interface mutation*. Interface mutation measures the coverage of interactions between software modules and is based on the following ideas [3]:
1) mutants are created to model integration errors,
2) one at a time, connections between two units are tested separately, and
3) the integration mutant operators are applied only on those parts of the units that are related to unit interfaces, such as function calls, parameters, or global variables.

Mutation analysis works in terms of a source code thereby it is inapplicable in case when there is no access to software's source code.

2.2 Structural metrics

Structural metrics examine the ability of a test suite to test software structural elements involved into interactions being tested. Calculation of structural metrics usually consists of two steps: on the first step, module's source code is analysed to discover dependencies between software elements being integrated; on the second step, the software is executed on a test suite and its behaviour is analysed to check whether the dependencies, discovered on the first step, were actually executed or not. The result of this check tells the level of coverage of interactions by the test suite. There are two major groups of structural metrics which differ by the type of software model they use. The first group is represented by metrics which use *program control flow model* [16, 12], the second group consists of metrics which use *program data flow model* [6, 12, 9].

Program control-flow graph, which is used by coverage metrics from the first group, is a directed graph, where each node represents a software element being integrated and each edge represents transfer of control between elements [17]. The graph is built by reverse-engineering of software source code. The type of elements depends on the type of tested interactions. Thus, in [16], authors test interactions between functions, so nodes correspond to functions and edges correspond to control flow between them. In [12] authors work with interactions between modules, so the nodes of the control-flow graph correspond to modules and edges correspond to external invocations of module's interface functions. Execution of software on test suite data is modelled as a path in the graph and is called *an execution path*. Control flow-based coverage metrics reflect how well software execution path covers graph's elements thus giving the information about coverage of internal software elements involved into interactions being tested.

Data flow model-based coverage metrics consider data flow through interfaces of software elements being integrated and are based on the investigation of the ways in which values are associated with variables and how these associations can affect the execution of the program. This analysis focuses on the occurrences of variables within the program. Each variable occurrence is classified as either a definition occurrence or a use occurrence [17]. A definition occurrence of a variable is where a value is bound to the variable. A use occurrence of a variable is where the value of the variable is referred to. Data-flow metrics are concerned with the coverage of paths in a data-flow graph from variable definition to variable use. A data-flow graph is built from a control-flow graph a by adding data-flow information into it. The exact type of data-flow graph, as in case of control-flow metrics, depends on a type of an interaction being tested and a type of metric being used. In [6], authors use interprocedural dataflow metric, which is an extension of definition-use coverage initially developed for unit testing [17]. In this case nodes correspond to functions of software and edges correspond to control flow between them. The metric, which was presented in [12], is also an extension of definition-use coverage metric, but it analyses the interactions between modules, so nodes of the control-flow graph correspond to modules and edges correspond to external invocations of module's interface functions. In [9], authors also consider interactions between modules, but they do not use data-flow graph model. Instead, they introduce a system of 12 types of interactions (*couplings*) between modules, and, for each type of interaction, they define data-flow coverage requirements.

2.3 Conclusions

As a result of the related work review, we can see that the existing test suite reduction methods for integration testing require an instrumentation or access to source code. This makes it complex and time consuming to use these methods for large software, and even impossible when source code is unavailable as in the case of the component-based development using COTS components.

3 Method description

To solve problems, highlighted in section 2, we present a new test suite reduction method for regression testing of interactions between two software modules. The method is based on modelling of interactions "behaviour" on a test suite and does not require source code access or instrumentation.

3.1 The model of interaction between two modules

To model module interactions behaviour, we use sequences of module's interface functions, invoked during the execution of a program. We consider not only the names of the functions, but also parameter values passed to them on invocation.

Before we start, let's make some definitions. Let's say we have two modules, *A* and *B*, and we consider interactions between them which are done via the interface of module *B*.

Interface functions are the functions, which form a part of module's programming interface.

Interaction trace of interaction between module *A* and module *B* is a sequence of interface functions of module *B*, which are invoked by module *A* during software execution on a test case *t*. We assume that these functions are presented in the trace according to the order of their invocation, and, together with their names, contain parameter values passed to them on invocation.

K-length sequence of interface functions is an arbitrary continuous sequence of length *K*, which can be located in an interaction trace.

Set of sequences of interface functions, which corresponds to interaction between modules *A* and *B* on the test *t*, is a set of all possible *K*-length sequences, which can be located in the interaction trace for the interaction between *A* and *B* on *t*.

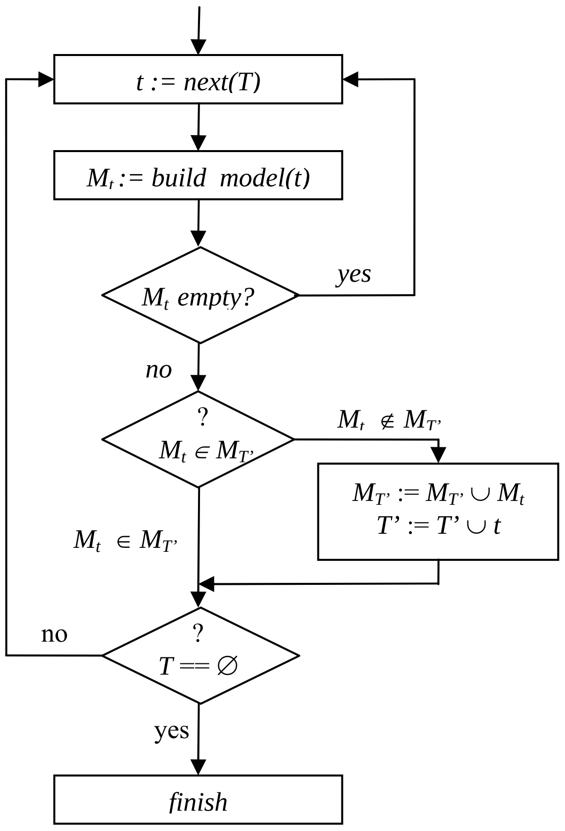
To construct the set of *K*-length sequences we use "sliding window" technique [8] with window size equal to *K*. By this approach, the set is constructed in the following way: the first *K* functions from the interaction trace form the first sequence; the next *K* functions, starting from the second function, form the second sequence; and so on, until the trace is passed.

We take the built above set of sequences as a *model of interaction* between module *A* and module *B* on a test *t*.

3.2 Test suite reduction method

The idea of our test suite reduction method is quite simple. Intuitively, those tests, which initiate the same sequences of interface functions, also repeat themselves in the module interactions they test. Moreover, the tests, which do not initiate any interactions between modules, are also of a little interest to us. Our method is based on these assumptions and "filters out" individual tests which either do not generate new sequences of interface functions, or do not invoke interface functions at all, or, in other words, following our assumptions, do not test an interaction in a new way.

The method uses the model of module interaction built in the previous section and implements the following algorithm (see Pic.1):



Pic.1 The method's algorithm.

where:

T – original test suite;

T' – reduced test suite, $T' \subseteq T$;

t – next test case from the original test suite, $t \in T$;

M_t – model of interaction behaviour on the test case *t*;

$M_{T'}$ – set of models of interaction behaviour on test cases from *T'*.

1. First, the next test case is picked up from the original test suite and the program is executed on it;
2. During the program execution, the model M_t of modules' interaction on a test case *t* is built; if the model is empty, the algorithm returns to the first step;
3. Then, M_t is checked to understand whether it is contained within the set of models $M_{T'}$ or not;
4. If M_t does not belong to $M_{T'}$, then M_t is inserted into $M_{T'}$ and test case *t* is added to the reduced test suite *T'*;
5. If *T* is not exhausted, the algorithm returns to the step 1; otherwise, the reduced test suite *T'* is considered as constructed and the algorithm stops.

We need to note, that using the set of sequences instead of interaction traces allows avoiding the "loops problem". This problem initially appeared in a calculation of path-based test suite adequacy criteria: if program's code contains loops, the number of different execution paths can be very big, which leads to the fall of criterion efficiency and to the rise of the testing costs [2]. The same problem can appear in our method if we use interaction traces instead of sequence sets. In this case, if interface function calls occur inside loops, a different number of loop executions can lead to very big amount of different interaction traces. This situation may affect results of a test suite reduction because interactions, which differ only by the number of executed loops, will be considered as different while they do not test new interaction behaviour between modules. Using sets of sequences solves this problem as, even for the arbitrary growth of the number of loop executions, the number of unique interaction behaviours will quickly stabilise and the model will stop changing.

3.3 Equality relation between interaction behaviours

In order to implement our method, we need a way to compare interaction models. To do this, we define the equality relation between two models, which tells us whether two models are equal or not.

We define the equality relation between two interaction behaviour models as following:

$$\delta(M_1, M_2) = \prod_{s_1 \in M_1} \sum_{s_2 \in M_2} \delta(s_1, s_2) \times \prod_{s_2 \in M_2} \sum_{s_1 \in M_1} \delta(s_2, s_1) \tag{1}$$

where M_1,M_2 – models being compared; s_1,s_2 – K-length sequences, which belong to the models; $\delta(s_1,s_2)$ – equality relation between sequences. In other words, we consider two given models as equal ($\delta(M_1,M_2)$ = 1), when their sequence sets are equal, and different ($\delta(M_1,M_2)$ = 0) otherwise. Two sequences of interface functions are considered to be equal when their elements are equal pairwise (in terms of equality relation defined for them – see below).

To define the equality relation between two interface function calls we take into account parameters values, passed to the functions on their invocation. As pointed out before, we consider simple interactions between two modules, so we assume that function parameters are scalar. Scalar parameters may be classified as nominal or numerical. The definition area for nominal parameters is a finite set with no order relation and distance defined. The equality relation for nominal parameters is $\delta(x,y)$, where δ is Kronecker symbol. The definition area for numerical parameters is R for which both order relation and distance are defined. The equality relation for numerical parameters is also defined using Kronecker symbol. However, such an equality relation is not always convenient as it does not take into account information about the distance between elements. One of the ways to solve this problem is to use the discretisation method, which splits the definition area of numerical parameter into a finite number of intervals, and to define the equality relation between numerical elements as the equality relation between their intervals:

$$\delta(x, y) = \delta(\text{interval}(x), \text{interval}(y))$$

Now we can define the equality relation between two function calls. Let's assume that a function f has m scalar parameters, where first n parameters are nominal and others are numerical. We define the equality relation between two function calls as:

$$\delta(f_1, f_2) = \delta(\text{name}(f_1), \text{name}(f_2)) \times \\ \times \prod_{i=1}^{n} \delta(x_i, y_i) \times \prod_{i=n+1}^{m} \delta(\text{interval}(x_i), \text{interval}(y_i)) \tag{2}$$

where: f_1,f_2 – function calls being compared; name(f_i) – the name of a function f_i; x_i,y_i, i = $1..n$ – values of nominal parameters of functions f_1 and f_2; x_i,y_i, i = $n+1..m$ – values of numerical parameters of functions f_1 and f_2. We can see that the equality relation for function calls f_1 and f_2 is true if and only if 1) the functions have the same names, 2) the values of nominal parameters are the same, and 3) the values of numerical parameters belong to the same intervals.

Let's now return to the equation (1). $\delta(s_1, s_2)$ = 1, if and only if all elements of sequences s_1 and s_2 are equal pairwise. In other words $\delta(s_1, s_2)$ may be represented as:

$$\delta(s_1, s_2) = \prod_{k=1}^{K} \delta(f_{1k}, f_{2k}) \tag{3}$$

After substitution of equation (2) into equation (3) we will have:

$$\delta(s_1, s_2) = \prod_{k=1}^{K} (\delta(\text{name}(f_{k1}), \text{name}(f_{k2})) \times \\ \times \prod_{i=1}^{n} \delta(x_{ki}, y_{ki}) \times \prod_{i=n+1}^{m} \delta(\text{interval}(x_{ki}), \text{interval}(y_{ki}))) \tag{4}$$

Then, if we substitute equation (4) into (1), we will get:

$$\delta(M_1, M_2) = \prod_{s_1 \in M_1} \sum_{s_2 \in M_2} \prod_{k=1}^{K} [\delta(\text{name}(f_{k1}), \text{name}(f_{k2})) \times \\ \times \prod_{i=1}^{n} \delta(x_{ki}, y_{ki}) \times \prod_{i=n+1}^{m} \delta(\text{interval}(x_{ki}), \text{interval}(y_{ki}))] \times \\ \times \prod_{s_2 \in M_2} \sum_{s_1 \in M_1} \prod_{k=1}^{K} [\delta(\text{name}(f_{k2}), \text{name}(f_{k1})) \times \\ \times \prod_{i=1}^{n} \delta(y_{ki}, x_{ki}) \times \prod_{i=n+1}^{m} \delta(\text{interval}(y_{ki}), \text{interval}(x_{ki}))] \tag{5}$$

We will take this equation as an equality relation between two interaction behaviour models.

It is important to note that, although we consider only scalar parameters of interface functions, our method can be adapted to take into account any function parameters for which an equality relation can be defined. This follows from the way we used to construct the equality relation.

4 Experimental evaluation

An experiment was conducted to investigate the effectiveness of the method presented in the Section 3 and to compare it to the random test suite reduction method. We used two characteristics of the reduction method:

1. Level of test suite reduction. This parameter demonstrates the ability of a reduction method to reduce the size of a test suite. By reducing test-suite size, test-suite reduction techniques reduce the costs of executing, validating, and maintaining test suites, thereby reducing the costs of the whole regression testing [15].
2. Fault detection rate. A potential drawback of test-suite reduction is that removing test cases from a test suite may significantly alter the fault-detecting capabilities of that test suite [15]. Losses from the fall of fault detection rate may exceed the benefits from cost saving on regression testing, thus it is very important that the fault detection rate of the reduced test suite remain the same or nearly the same as of the original test suite.

4.1 Experiment conditions and subject applications

The experiment was conducted in Cygwin v1.5.24/Windows XP environment; invocations of interface functions were monitored using function call interception mechanism provided by the linker. The length K of interface function sequences was 6. GNU Assembler and the standard C library were used as modules to be integrated.

GNU Assembler [4] is a part of GNU Binutils package and freely available with its source code at [5]. GNU Assembler is a popular open-source project and constantly evolves in different directions: changes are made to its logic, new extensions are created to support new object file formats, architectures and platforms. In such a situation, a regression testing is necessary to ensure that the product works correctly after modifications. Regression testing of GNU Assembler is conducted using the test suite distributed together with the assembler.

The C standard library is a collection of header files and library routines used to implement common operations, such as input/output and string handling, in the C programming language. The library is standardized [1] and used by majority of programs written using C programming language.

We used the interaction between GNU Assembler and the standard C library, which was done through a subset of library functions responsible for input/output and defined in stdio.h header file, as an interaction to be tested. We used version 2.17 of GNU Assembler, which was configured as “i686-pc-cygwin”. The standard C library, was taken from Cygwin v1.5.24 package. GNU Assembler’s test suite, which contains 109 individual test cases, was used as a source of test cases. All materials used in the experiment are available in free access, so the experiment can be repeated independently.

In order to monitor interface function invocations, we used function call interception mechanism provided by *ld* linker which is also a part of the Binutils package. This mechanism allows building wrapper functions for functions with known signature and does not require access or instrumentation of a source code. Using this mechanism we developed software module, which implements wrapper functions for interface functions of the interaction being tested, and embedded this module into the assembler at linking stage.

To conduct the experiment, we seeded 38 integration errors to the parts of assembler source code, which are involved into interactions with the standard C library through its input/output routines.

In order to evaluate method’s characteristics we used the following indicators:

1. Size of the reduced test suite;
2. Percentage of test suite reduction, which was calculated as:

$$100 \times (1 - Size_{reduced}/Size_{original}),$$

where $Size_{original}$ is a size of the original test suite, $Size_{reduced}$ is a size of the reduced test suite;
3. Number of detected faults;
4. Fault detection rate (in percentage), which was calculated as:

$$100 \times (FaultsDetected_{reduced}/FaultsDetected_{original}),$$

where $FaultsDetected_{original}$ is a number of faults detected by the original test suite, $FaultsDetected_{reduced}$ is a number of faults detected by the reduced test suite.

Fault detection rate was calculated on a *per-test-suite*[15] basis: two different test suites are considered to be equivalent in discovering a particular fault, if each of them contains at least one test case which discovers this fault. This approach was also used in [13] and [15].

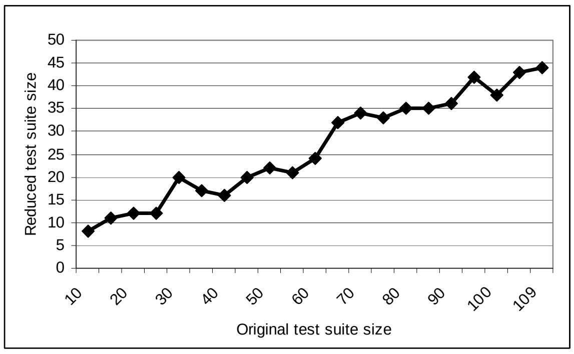
4.2 Conduction of the experiment

As the order, in which test cases go in a test suite, implicitly influences the work of the method being evaluated (as individual test cases are picked up from the original test suite in the order they are presented there, it is the first test case, which generates the new behaviour, goes into the reduced test suite), we randomly reordered the test pool to ensure the correctness of the experiment.

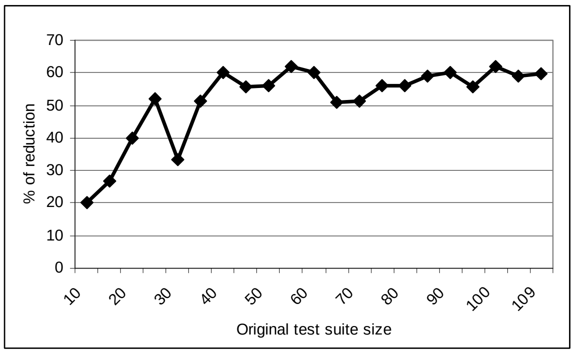
After that, from the original pool of test cases, we constructed 21 random original test suites of different size, from 10 to 105, with step 5. To construct the last, 21st, original test suite, we took all 109 test cases. Then, the assembler was executed on original test suites to build the models of interactions behaviour and, based on these models, original test suits were reduced using our method. After that, we calculated the size of the reduced test suites, the percentage of reduction, and fault detection rates. Finally, we compared fault detection rates of the reduced test suites with rates of the original test suites and randomly reduced test suites. Randomly reduced test suites were constructed out of the original test suites to have the same size as the reduced test suites.

4.3 Experiment results

Experiment results are illustrated at Pic. 2 – 5:

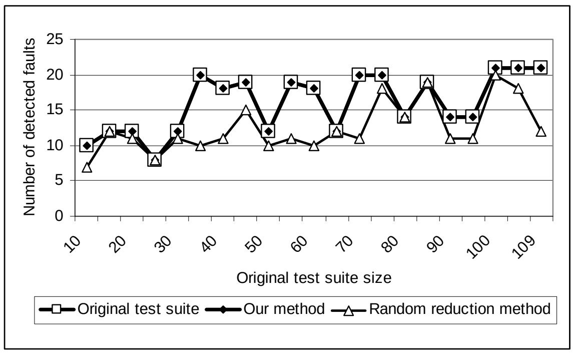


Pic. 2 Size of reduced test suites.

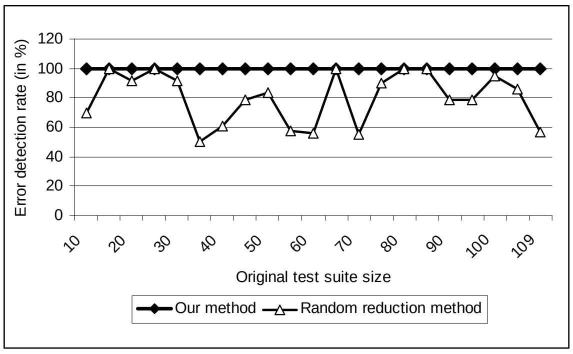


Pic. 3 Percentage of reduction.

The size of the reduced test suite was between 8 and 44 tests depending on the size of the corresponding original test suite. Reduction rate was between 20% and 62%.



Pic. 4 Number of detected faults.



Pic. 5 Fault detection rate.

Fault detection rate for the reduced test suites had not decreased and was the same as for the original test suites. Moreover, in most of cases our method demonstrated better results than random reduction method.

5. Conclusion

In this paper we presented the new test suite reduction method for regression testing of simple interactions between two software modules. The method is based on the modelling of the behaviour of interactions between software modules and used sequences of interface functions invoked during software execution. We considered simple interactions between modules, but the method can be adapted to take into account any function parameters for which an equality relation can be defined. The experimental evaluation demonstrated that our method outperforms the random test suite reduction method and can successfully reduce the size of a test suite while retaining its fault detection ability.

6. Acknowledgement

The author would like to thank A.K. Petrenko and his colleagues from The Institute for System Programming of the Russian Academy of Sciences for commenting on the earlier revisions of this work.

References

[1] American National Standards Institute. Programming Language C. ANSI standard X3.159-1989. ANSI, December 1989.
[2] Steve Cornett. Code Coverage Analysis, Bullseye Testing Technology, 2002. http://www.bullseye.com/coverage.html.
[3] Delamaro, M. E., Maldonado, J. C., and Mathur, A. P., Interface Mutation: an approach to integration testing, IEEE TSE, Vol. 27, No. 3, March 2001, pp228-247.
[4] Documentation for GNU Assembler 2.17, 2006. http://sourceware.org/binutils/docs-2.17/as/index.html.
[5] FSF/UNESCO Free Software Directory Website, GNU Binutils - Collection of binary utilities, 2006. http://directory.fsf.org/GNU/binutils.html.

[6] Harrold, M. J. and Soffa, M. L. Selecting and Using Data for Integration Testing. IEEE Softw. 8, 2 (Mar. 1991), pp.58-65, 1991.

[7] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology, 2(3):270-285, July 1993.

[8] S.A.Hofmeyr, S.Forrest, and A.Somayaji. Intrusion Detection using Sequences of System Calls. Journal of Computer Security, 6:151–180, 1998.

[9] Zhenyi Jin and Jeff Offutt. Coupling-based criteria for integration testing. The Journal of Software Testing, Verification, and Reliability, 8(3):133-154, September 1998.

[10] Kichigin D.Yu. About one test suite reduction method. Collection of papers of The Institute for System Programming of the Russian Academy of Sciences. ISP RAS, Moscow, 2007. //Кичигин Д.Ю. Об одном методе сокращения набора тестов. Сборник трудов ИСП РАН. М: ИСП РАН, 2007.

[11] Kim, Y. W. 2003. Efficient use of code coverage in large-scale software development. In Proceedings of the 2003 Conference of the Centre For Advanced Studies on Collaborative Research (Toronto, Ontario, Canada, October 06 - 09, 2003). IBM Centre for Advanced Studies Conference. IBM Press, 145-155.

[12] Linnenkugel, U. and Müllerburg, M. Test data selection criteria for (software) integration testing. In Proceedings of the First international Conference on Systems integration on Systems integration '90 (Morristown, New Jersey, United States). IEEE Press, Piscataway, NJ, 709-717, 1990.

[13] McMaster, S. and Memon, A. M. 2005. Call Stack Coverage for Test Suite Reduction. In Proceedings of the 21st IEEE international Conference on Software Maintenance (ICSM'05) - Volume 00 (September 25 - 30, 2005). ICSM. IEEE Computer Society, Washington, DC, 539-548.

[14] Piwowarski, P., Ohba, M., and Caruso, J. 1993. Coverage measurement experience during function test. In Proceedings of the 15th international Conference on Software Engineering (Baltimore, Maryland, United States, May 17 - 21, 1993). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 287-301.

[15] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. Journal of Software Testing, Verification, and Reliability, V. 12, no. 4, December, 2002.

[16] Atanas Rountev, Scott Kagan, and Jason Sawin. Coverage Criteria for Testing of Object Interactions in Sequence Diagrams, Fundamental Approaches to Software Engineering (FASE'05), LNCS 3442, pages 282-297, April 2005.

[17] H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, Vol.29, No.4, pp.366-427, December 1997.