Towards Tool-Support for Test Case Selection in Manual Regression Testing

Georg Buchgeher, Christian Ernstbrunner, Rudolf Ramler
Software Competence Center Hagenberg (SCCH)
Hagenberg, Austria
{firstname.lastname}@scch.at

Michael Lusser
OMICRON electronics GmbH
Klaus, Austria
michael.lusser@omicron.at

***Abstract*—Manual regression testing can be a time-intensive and costly activity. Required efforts can be reduced by selecting only the tests for re-testing that verify actually modified system parts. However, if testers are not familiar with the system implementation the selection of relevant test cases may become difficult. In this paper we report on our experiences with the development of a tool-based approach supporting the selection of manual regression tests. The presented approach is developed together with the software testing department of an international company. Test cases are selected by analyzing different kinds of information resources, i.e., code coverage information and data provided by versioning systems. Experience shows that code coverage information can assist in selecting candidate test cases for regression testing. However, we also encountered some principal challenges in implementing test case selection in practice: Relying solely on code coverage often leads to a large set of test cases, available versioning systems lack the necessary details to map code changes to relevant structural elements, and collecting and keeping coverage data for manual regressing testing up-to-date involves additional costs and effort.**

Keywords—regression testing; test suite reduction; test case selection; manual testing.

I. Introduction

Regression testing is the process of re-testing a software system after modifications to the system have been made [1] in order to ensure that the new version of the system has retained the capabilities of the old version, and that no new defects have been introduced. If regression testing discovers defects, it is likely that the modified code "broke" the system. Regression testing is a time consuming activity [2]. Therefore it is ideally performed via automated tests that are executed as part of software build processes. Although automation of regression testing is highly desirable, it cannot always be achieved for large software systems containing a legacy of components and technologies of varying testability. Typical examples impeding automation are system tests that require direct interactions with the user or the hardware [2], or tests that require manual inspections of the results (e.g., if all elements of the user interface are properly arranged and the graphical representations are clear and "attractive") [3].

The re-execution of manual tests requires a considerable amount of human resources. Thus, manual regression testing is usually a time intensive and costly activity. These costs can be reduced by re-testing only a subset of the available test cases. Ideally only those tests are re-executed, which check the parts of the system that have been modified or that are affected by modifications. However, selecting relevant tests is difficult when development and testing is performed by different teams. In this case, testers are often not sufficiently familiar with the system's implementation to spot all relationships and indirect dependencies between the modifications made in development and the test cases that should be selected for re-testing.

In this paper we report on our experiences with the development of an approach that provides automated assistance for the selection of manual regression tests. The selection of test cases can be supported by analyzing different kinds of data produced as part of the development and test process – i.e., code coverage data recorded when manually executing the test cases, data about modifications provided by versioning systems, and data collected about the dependencies in the system's implementation. The presented approach supporting the selection of regression test cases has been developed together with the software testing department of a company in the electrical engineering domain producing electronic measuring and diagnostic equipment. Thereby we followed an action research methodology [4]. Action research is characterized by a close interaction between academia and industry with continuous feedback loops and observations [5]. In this way we were able to get first hand experiences and fast feedback in order to validate and continuously refine our approach.

The remainder of this paper is organized as follows: Section II describes the industrial context in which we developed our approach. In Section III we give an overview of the conceptual underpinnings. Section IV describes the developed tool support. In Section V we report on the collected experiences with the developed approach as well as the tool support. Section VI discusses related works. Section VII concludes with a summary and an outlook on current and future work.

II. Industrial Context

In this section we describe the context in which the presented approach has been developed. In particular, we provide a brief overview of the domain and the system for which the presented approach has been developed, and we describe central problems that we seek to address with our work.

OMICRON electronics GmbH is a producer of measurement and diagnostic solutions for the electrical power industry. OMICRON started to develop a software suite for electrical hardware diagnostics in 1995. This software suite has a modu-

978-0-7695-4993-4/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSTW.2013.16

lar architecture (see Figure 1): The *hardware (HW) abstraction layer* contains device drivers for measuring hardware and translates software statements to machine instructions. The *framework* uses the HW abstraction layer to communicate with electrical measurement devices and provides common components (for persistency, reporting, etc.) and data models for diagnostic modules. Different diagnostic *modules* support specific manual checks and functionalities such as protection relay checking, transducer checking, and power quality analyzer checking. Each module may operate either as stand-alone application or as part of a suite, i.e., the *center* (execution environment). If two modules support similar diagnostic functionality their commonalities are extracted to a single package of *shared code*. The software suite including all modules consists of more than 2.5 million lines of code (MLOC). More than half of the system is written in unmanaged C++; C# (0.5 MLOC), Visual Basic (0.4 MLOC) and C are the other major programming languages used in the system.

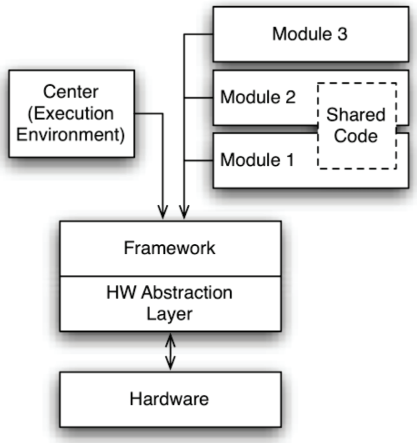


Figure 1: Overview of the system architecture under test.

The software system is primarily tested manually. Automated tests are only available for a small part of the system, mainly in form of acceptance tests for hardware related parts such as the Hardware Abstraction Layer shown in Figure 1. This has two main reasons: First, the diagnostic modules are user interface centered and, thus, automatic testing only works in combination with user interface test automation [6]. When starting with development and testing of the software suite in the late 90's, the available tools that supported user interface test automation (e.g., via capture and replay) were not popular because the produced tests were often fragile, inflexible and the configuration of the tools as well as the test environment was very time-consuming. The best solution was therefore a basis of manual tests combined with an automated test suite for selected parts of the software system [7]. Second, the strong dependencies on hardware and measurement equipment restrict the conclusiveness of automated tests as the results are not always exactly reproducible. Test results may vary for the same test with the same setup and therefore need interpretation of an expert. At best, such tests are semi-automated. They are automatically executed, the results are written to a results file and the tester validates the results for correctness.

In more than 15 years of implementation and maintenance work, the software system grew tremendously in size as well as in complexity. Furthermore, many different developers, architects, product owners and testers were involved in the evolution of the system building up individual pools of knowledge about the various parts and aspects of the system and how they work together and how they depend on each other. Thus, today, the effort required for manual regression testing has pushed the time and budget limits to a level where simply adding more personnel to the test team is not a viable option.

The main challenge that has to be addressed for effective and efficient regression testing is the management of the knowledge about dependencies in the software system. With the continuous growth of the system also the amount of dependencies between the different parts of the system increased – on the one hand, between the system's modules, but on the other hand also within individual modules and across parts implemented in other system layers. Technically the architecture of the system (see Figure 1) splits the different functional parts of the system in separate modules while common aspects are combined in the framework or in form of shared code. Thus, dependencies between modules have been created and changes may affect a potentially large number of modules. Furthermore, implicit dependencies between modules exist that can only be discovered with detailed knowledge about the modules' functionalities and implementation.

Over the years personnel turnover has further intensified the challenge of managing knowledge about dependencies. When developers leave the development team, the rationale for certain design and implementation decisions is lost. Without this knowledge making changes to the system may lead to unintended side effects and, thus, defects. For example, making changes in shared code may lead to defects in other modules using this shared code. Regression testing is the key measure for detecting defects introduced by such modifications. However, new testers joining the test team require a significant amount of time for getting familiar with the system. For new testers, selecting the right test cases for a particular modification is a difficult task due to the lack of knowledge about how to match the system implementation and the available test cases. But also for experienced testers the selection of test cases can be challenging due to the many potential dependencies that may lead to unintended side effects. Testing for such side effects includes not only the test cases that exercise the modification but should involve all those test cases that check potentially affected system parts, i.e., those parts that have not been modified but explicitly or implicitly depend on a modified system part. Therefore, valuable support for testers in the selection of relevant test cases can be provided by disclosing such dependencies.

III. APPROACH

Given the challenges described in the previous section it was our aim to develop an approach that assists testers in the selection of test cases for manual regression testing. Ideally relevant test cases can be proposed automatically for a given set of system modifications. In this section we provide an overview of the conceptual underpinnings of our approach.

In order to identify test cases that check a set of system modifications we use two central kinds of information: (1) information about system modifications and (2) information which test cases actually check these modifications (e.g., a modified file or a method). In our approach we obtain this information as follows:

- *Information about modified system parts*: This information is obtained by analyzing data provided by versioning systems. Versioning systems like Subversion or the Microsoft Team Foundation Server (TFS) keep track of all system modifications by storing each modification (check-in) as a separate revision. Versioning systems typically provide an API for accessing check-in information like the modification time, the modified files and which specific lines of code have been modified.
- *Code coverage information:* In order to identify test cases that have the potential to verify a certain part (i.e., a file or method) of a system we use data provided by code coverage tools. Code coverage tools like NCover measuree which parts of a system (i.e., which lines of code, statements and branches) are actually executed when a test case is run. This way untested execution paths in the system implementation can be identified. We use the code coverage data for determining the parts of the system that are actually exercised by a test case.

Information about modified system parts and code coverage are then combined as depicted in Figure 2. As shown in the figure, for each revision contained in the versioning system we determine which files of the system have been modified. For these files we search for corresponding test cases that cover the modified files. In Figure 2, for example, for testing the modifications made by *Revision 101,* the tests *Test 1* and *Test 2* need to be executed to verify the modified *File A*.

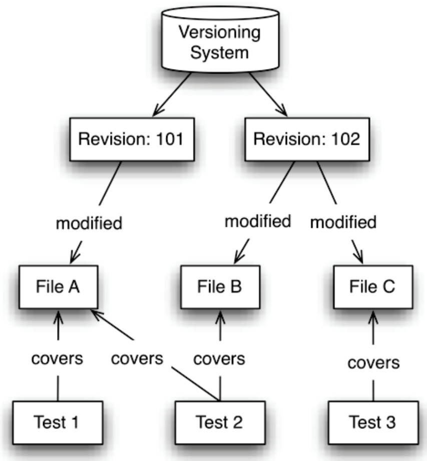


Figure 2: Examplary use of modification and coverage information for test case selection.

IV. TOOL SUPPORT

In order to make our approach practically applicable we have developed a corresponding prototype (Sherlock). In this section we provide some details on the technical realization and how we support the test selection process.

A. Technical Realization

An overview of the technical realization of our approach is depicted in Figure 3. As shown in the figure, all relevant data is stored in a graph database. A graph database permits fast queries of graph-based structures [8]. In our case the database contains revisions, files and test cases represented as nodes. Information about which files are modified by a revision and which files are covered by a test case are stored as relationships.

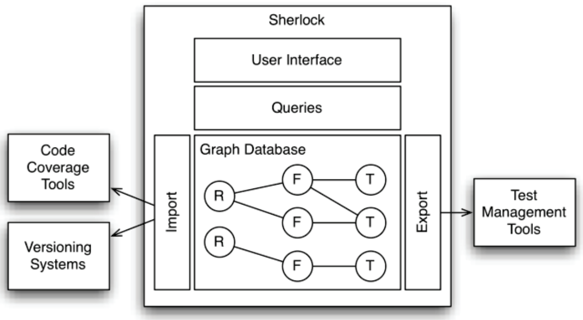


Figure 3: Overview of the Sherlock tool prototype.

The graph database is filled by importing information from code coverage tools and versioning systems. Code coverage tools typically export coverage information as XML documents, which can easily be parsed. XML documents contain information which files, methods and lines of the code have been executed by a test case. We have developed import operations for different code coverage tools because the analyzed system has been implemented with a heterogeneous set of programming languages and technologies (see Section II). Versioning systems typically provide an API for accessing information about system modifications. This information typically includes modified files and source line changes relatively to the previous version. However, not all necessary details about structural elements like modified classes, methods and fields are provided by the existing tools.

Once the available information has been imported in the graph database, the tester can use Sherlock for searching for test cases related to a given set of system modifications. It is possible to search for test cases relating to single revisions as well as all revisions within a specified date range. Searching for tests for a single revision is important in case of verifying single bug fixes, while checking all revisions in a given date range is required, for example, in verifying periodic maintenance releases.

The main focus of Sherlock lies in the identification and selection of relevant test cases. Once relevant test cases have been selected, this selection can be exported to the test management tool that is used to control the actual testing process.

B. Test Case Selection

In Sherlock the test case selection process is an interactive process that contains at least two steps. In the first step the tester needs to define the scope for the selection process. In the second step the actual test cases are selected.

First, as part of the scoping process, the tester define the date range in which system modifications took place are and the particular parts of the system that should be tested. Restricting the system parts is important for focused testing of single modules. Based on the date range and the selected system parts, relevant modifications and related test cases are queried.

Figure 4 depicts a screenshot of the user interface for the test selection as implemented in Sherlock. The left part of the user interface provides an overview of the context for the selection process. As shown in the figure, the context provides information on the considered check-ins, the modified files and methods, and all the available test cases that provide coverage for the modified resources.

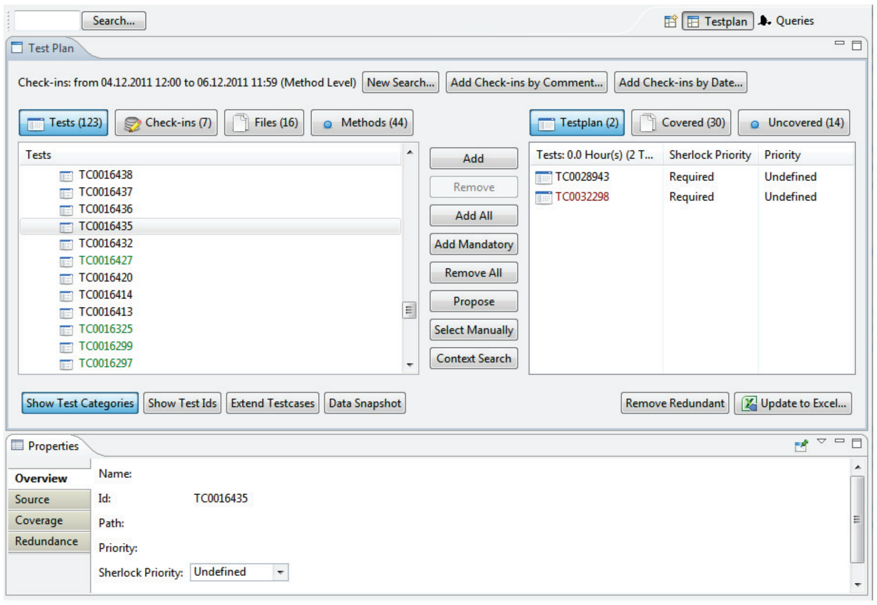
The second step comprises the actual selection of tests that will be considered for regression testing. Proposed test cases are listed in the right part of the user interface (see Figure 4). We assist the tester in different ways. For instance, in the left part of the user interface, test cases that increase the coverage are displayed in green. In the right part of the user interface, test cases that can be removed without decreasing the coverage rate are displayed in red. Furthermore it can be analyzed which check-ins, files and methods are actually covered by the selected test cases and which parts are still uncovered. This information is important for identifying currently untested modifications.

V. Experiences and Preliminary Results

We have been working together with the testing department of OMICRON electronics GmbH for about 12 months to jointly develop the presented approach. During this time the results were discussed on a regular basis and the approach has been continuously refined to improve the support for test case selection. The testing department has collected coverage data for 392 test cases, which provide comprehensive coverage for two modules of the software system described in Section II. These test cases served as a basis for assessing and discussing our approach. While we were able to continuously improve our approach and obtain successful first results, we also encountered some fundamental questions and challenges in our work that we consider generally relevant for implementing test case selection in practice. In the following we discuss these questions and challenges as well as our solutions or the current workarounds.

Analyzing modifications at files vs. method level. Initially we have been analyzing the system at the granularity level of files. For proposing test cases this means that a test case was selected if a file was modified by a check-in and if a test case provided some coverage for this particular file. The main reason for working at this level of abstraction was the circumstance that the used versioning systems provide data about modified files and modified lines of code, but not about structural elements such as methods or fields. Early results showed that working on file level leads to proposing unacceptable large test sets. In many cases, over 200 test cases have been identified for a single modification. As a consequence we started exploring if the number of test cases can be reduced by working at the abstraction level of methods. Table 1 provides a comparison of the number of identified test cases when working at the abstraction level of files versus methods. As shown in the table, for some modifications the number of identified test cases could be significantly reduced. For example, for modification #1 the number of potentially relevant tests could be reduced from 160 to 14 suggestions, and for the modifications #11 and #12 the number of suggested tests could be reduced from 384 to 35 while still covering all modifications.

However, as also shown in the table, for several modifications the abstraction level of methods has not led to a significant reduction in the number of proposed test cases. For example, for the modifications #5 and #6 the number of proposed tests was only reduced from 222 to 200. We found that more "central" files are always covered by an extensive number of



Check-ins: from 04.12.2011 12:00 to 06.12.2011 11:59 (Method Level)

Tests
TC0016438
TC0016437
TC0016436
TC0016435
TC0016432
TC0016427
TC0016420
TC0016414
TC0016413
TC0016325
TC0016299
TC0016297

Tests: 0.0 Hour(s) (2 T...	Sherlock Priority	Priority
TC0028943	Required	Undefined
TC0032298	Required	Undefined

Name:
Id: TC0016435
Path:
Priority:
Sherlock Priority: Undefined

Figure 4: Sherlock user interface for test case selection and test planning.

test cases. If the modifications affect these files, it is not possible to reduce the number of suggested test cases by changing the granularity level of the analysis.

TABLE 1: NUMBER OF PROPOSED TESTS AT FILE- VS. METHOD-LEVEL

Modification #	*Affected Files*	*Proposed Tests*	*Affected Methods*	*Propoesed Tests*
1	1	160	1	14
2	1	48	1	48
3	1	48	1	48
4	1	384	1	35
5	1	222	1	200
6	1	222	1	200
7	1	222	1	222
8	1	175	2	175
9	1	222	2	222
10	1	384	11	175
11	2	384	1	35
12	2	384	1	35

Ignoring coverage from startup. In case of the studied modules, we observed that a major part of the implementation is already executed at system startup, when the modules are initialized. So a large share of the module's implementation is covered even without executing any of the actual functionality. Thus, despite designing test cases that address different functional aspects of a module, the coverage footprint of these test cases is largely identical. Furthermore, due to the large coverage footprints, a high number of test cases are suggested for almost any modification of the module. One possible solution to deal with extensive startup coverage is to consider how often a modified system part is executed by a test. Tests that execute the modified system part more than once, i.e., not only as part of the startup sequence, should be considered more relevant. We have not yet evaluated this heuristic, because the currently used code coverage tools do not collect data about how often the parts of the system are executed.

Relating changes to structural elements. Another challenge we encountered is the detection of modified system parts based on information from versioning systems. While it is no problem to identify modified files, the detection of modified methods is a difficult task. Versioning tools only provide information on the modified lines of code, but not which methods these lines belong to. When we analyzed the feasibility of working at the abstraction level of methods we identified the modified methods by manually comparing revisions of modified files. However, we have yet to develop a way to automatically determine modified methods using a diff that takes the structural information into account. A challenge closely related to the detection of modified methods is to filter not relevant system modifications. Not each modification requires testing. For instance, formatting operations, removed blanks, added source code comments, or renamed variables are changes that can be excluded from re-testing. In order to detect such changes it would be necessary to analyze modifications not at the file level but at the level of the abstract syntax tree. Since the tested system is a heterogeneous system, diff algorithms further would need to consider multiple programming languages.

Keeping coverage data up-to-date. A further issue tied to the use of coverage information is the fact that this information tends to get outdated over time. For automated tests, coverage data can be collected while executing all test cases at no or little additional costs. However, updating code coverage information for manual test cases requires the manual re-execution of each test case. The involved effort for collecting or updating code coverage information would thus be similar to the effort required for manual testing. The problem of outdated code coverage data is especially critical during the system development process, when files and methods are changed frequently and new files and methods are added to the system. Collecting code coverage data during these periods of heavy development activity may therefore result in highly inaccurate coverage data. During the system maintenance phase – when only smaller and more localized changes are made – code coverage information is more likely to remain current over a longer period of time. In response we plan to compute a "reliability factor" for coverage data to issue a warning when test cases are selected based on potentially outdated coverage information.

Selecting defect exposing tests. Code coverage reveals only what parts of the code a test case has executed but not what it has actually checked. This limitation of coverage measures becomes obvious when two or more test cases have an identical coverage footprint as they execute the same functionality but verify different aspects of the functionality. As a result, although several test cases may exercise the modified part, only one may check the aspect that is related to the specific modification and, thus, only this test case is able to expose a defect. The problem is less likely to be encountered in manual testing than in automated test execution since human testers usually observe different aspects of the system's behavior on the fly and can interactively explore the system when noticing irregularities. As a pragmatic solution, checking the results of a test case from different viewpoints and avoiding similar tests that only differ in checkpoints has been added as guideline for designing tests. Nevertheless the missing information about what actually is checked is still considered an issue in selecting relevant test cases out of a number of potential candidates for testing a modification. Making such decisions automatically may not be possible without further information. In our experiments and the discussions we found that human testers are able to make accurate decisions when selecting one test case out of many test cases with a similar or equivalent coverage footprint if the test objective is evident from the test case name.

VI. RELATED WORK

Numerous approaches exist for the selection of regression test cases. A comprehensive overview can be found in the literature reviews published by Yoo and Harman [9] or by Engström, Runeson and Skoglund [10]. Yoo and Harman [9] classify existing approaches into three categories: test case minimization, test case selection and test case prioritization. *Test case minimization* techniques aim to reduce test suites by eliminating redundancies [11]. These approaches are not applicable in our context due to the high redundancy of most test cases we analyzed. *Test case prioritization* approaches deal with the ordering of test cases to maximize desirable properties like the early fault detection [12]. We do not yet consider any test case prioritization approaches in our tool support, but we plan to integrate such techniques as part of our future work. Most of our current work relates to the category of *test case selection* approaches, which deal with the selection of a subset of tests that actually check system modifications [9]. Techniques in this category that are closely related to our approach are the graph-walk technique [13] and the modification-based technique [14]. In contrast to the scientific studies investigated in the literature reviews, which focus on a refinement of the mentioned techniques under optimal experimental conditions, our work is mainly concerned with the challenges one encounters when implementing these techniques in a real-world setting (see Section V). Thus, while we aim to draw from the conclusions of previous studies, we find ourselves often confronted with practical limitations in terms of quality, granularity or availability of necessary data that impose challenges requesting different solutions than those proposed in previous studies.

Related practical experiences have been reported by Juergens et al. [2] who presented an industrial case study on the identification of manual regression tests. Similar to our experiences, they identified the problem that frequently too many test cases are selected. To overcome this problem they proposed a semi-automated approach named *trace-guided test case selection*. In this approach the tester performs a manual prioritization of the automatically identified test cases. Trace-guided test case selection is compatible to our two-step selection approach.

Despite the existing body of research we are not aware of any tools adopted in industrial practice for the selection of manual regression tests. Available tools for regression testing usually focus on automation of test case execution and continuous integration. For example, at the level of unit and integration testing Microsoft's Visual Studio 2010 provides built-in support for test impact analysis (TIA) [15]. Visual studio automatically detects modified methods, searches corresponding unit tests and automatically executes these tests. The concept implemented in TIA is similar to our approach. However, TIA is only applicable for automated testing, whereas our approach has been developed for manual testing. The time and cost intensive work that remains without adequate tool support involves test case specification, configuration of the test execution environment, etc. Available test management tools that tend to address these tasks lack support for systematic test case selection or prioritization.

VII. Conclusion and Future Work

In this paper we have described our experiences with the development of a tool-based approach supporting the selection of manual regression tests. Regression test cases are selected by analyzing data provided by code coverage tools and versioning systems. Our experiences showed that code coverage can assist the process of manually selecting test cases by providing information on covered and uncovered files and methods. However, the selection of test cases based solely on code coverage often leads to a large set of candidate test cases without enough additional information to select the most appropriate test cases to verify a modification. Furthermore, measures to keep the coverage data up-to-date and to exclude irrelevant information need to be developed.

Although we have not yet achieved our goal of providing automated support for selecting a small sets of relevant test cases for manual regression testing, the proposed approach has already shown is usefulness in assisting testers during the manual selection process by identifying all potential test cases for a set of modifications and by indicating gaps where modifications are not adequately covered by any of the available test cases. Our current and future work aims at improving the process of selecting test cases and the prioritization of the proposed tests based on additional information such as how often a particular method is executed by a test case. A major obstacle in implementing our approach is the circumstance that many of the existing tools such as the versioning system and code coverage tools do not provide the required information at the right level of detail or quality.

References

[1] Paul Ammann and Jeff Offutt. 2008. Introduction to Software Testing (1 ed.). Cambridge University Press, New York, NY, USA.

[2] Elmar Juergens, Benjamin Hummel, Florian Deissenboeck, Martin Feilkas, Christian Schlogel, and Andreas Wubbeke. 2011. Regression Test Selection of Manual System Tests in Practice. In Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering (CSMR '11). IEEE Computer Society, 309-312.

[3] Josef Pichler and Rudolf Ramler. 2008. How to Test the Intangible Properties of Graphical User Interfaces? First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, IEEE Computer Society, 494-497.

[4] David E. Avison, Francis Lau, Michael D. Myers, and Peter Axel Nielsen. 1999. Action research. Commun. ACM 42, 1 (January 1999), 94-97.

[5] Ned Kock. 2006. Information Systems Action Research: An Applied View of Emerging Concepts and Methods (Series in Information Systems). Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[6] Lee White, Husain Almezen, and Nasser Alzeidi. 2001. User-Based Testing of GUI Sequences and Their Interactions. 12th International Symposium on Software Reliability Engineering (ISSRE '01). IEEE Computer Society, 54-63.

[7] Rudolf Ramler, Klaus Wolfmaier. 2006. Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost. Proc. of the Workshop on Automation of Software Test (AST '06) at 28th Int. Conf. on Software Engineering (ICSE '06). ACM, 85-91.

[8] Robert Dabrowski, Krysztof Stencel, and Grzegorz Timoszuk. 2011. Software is a directed multigraph. In Proceedings of the 5th European conference on Software architecture. Springer-Verlag, 360 -369.

[9] Shin Yoo and Mark Harmann. 2012. Regression Testing Minimisation, Selection, and Prioritisation: A Survey. Software Test. Verif. Reliab, vol. 22, iss. 2, pp. 67-120.

[10] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A systematic review on regression test selection techniques. Inf. Softw. Technol. 52, 1 (January 2010), 14-30.

[11] A. Jefferson Offutt, Jie Pan, and Jeffrey M. Voas. 1995. Procedures for reducing the size of coverage-based test sets. In Proceedings of the 12th International Conference on Testing Computer Software, ACM, 1995; 111–123.

[12] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing test cases for regression testing. ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '00), ACM, 102-112.

[13] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. 6, 2 (April 1997), 173-210.

[14] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. 1994. TestTube: a system for selective regression testing. In Proceedings of the 16th international conference on Software engineering (ICSE '94). IEEE Computer Society, 211-220.

[15] Daryush Laqab and Tim Varcak. 2010. Streamline Testing Process with Test Impact Analysis. MSDN Library, Microsoft Visual Studio 2010. (http://msdn.microsoft.com/en-us/library/ff576128.aspx)