2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation

Toolset and Program Repository for Code Coverage-Based Test Suite Analysis and Manipulation

Dávid Tengeri, Árpád Beszédes, Dávid Havas and Tibor Gyimóthy
Department of Software Engineering
University of Szeged, Szeged, Hungary
{dtengeri,beszedes,havasd,gyimothy}@inf.u-szeged.hu

***Abstract*—Code coverage is often used in academic and industrial practice of white-box software testing. Various test optimization methods, e.g. test selection and prioritization, rely on code coverage information, but other related fields benefit from it as well, such as fault localization. These methods require access to the fine details of coverage information and efficient ways of processing this data. The purpose of the (free) SoDA library and toolset is to provide an efficient set of data structures and algorithms which can be used to prepare, store and analyze in various ways data related to code coverage. The focus of SoDA is not on the calculation of coverage data (such as instrumentation and test execution) but on the analysis and manipulation of test suites based on such information. An important design goal of the library was to be usable on industrial-size programs and test suites. Furthermore, there is no limitation on programming language, analysis granularity and coverage criteria. In this paper, we demonstrate the purpose and benefits of the library, the associated toolset, which also includes a graphical user interface, as well as possible usage scenarios. SoDA also includes a repository of prepared programs, which are from small to large sizes and can be used for experimentation and as a benchmark for code coverage related research.**

Keywords*—Regression testing, test suite analysis, test suite optimization, code coverage, program repository.

I. Introduction

Using code coverage in software testing is based on simple concepts, but it often involves technical difficulties. For code coverage, one records which parts of the program code have been executed and which not, by observing the runtime behavior of the (dynamic) test cases. This concept is generally used in test case design, test progress monitoring and as test exit criteria, and is a basis for white-box testing approaches in industrial [1] as well as academic [2] practice. More advanced methods benefit from code coverage information as well, such as selective retesting, test case prioritization, automated test case generation and fault localization. There exists a large body of methods, algorithms and tools for code coverage measurement including various coverage criteria (statement, decision, condition, etc.) and support for different languages and platforms (both commercial and open source).

Working with code coverage in practice involves several technical difficulties, however. A notable one is the storage and analysis of the coverage information itself (in this study, we do not deal with other issues such as the calculation of coverage, instrumentation, test executions, etc). For systems consisting of millions of lines of code and maybe hundred thousand test cases, the amount of coverage data will be huge. Also, various coverage calculator tools produce data in different formats. This makes the application of code coverage approaches difficult for most resarchers and practitioners, especially for the implementation of more advanced methods. Finally, research prototypes and available benchmark programs typically focus only on small to medium size systems.

The aim of the *SoDA (Software Development Analysis framework)* library, toolset and program repository is to provide researchers and practitioners a framework with which various code coverage-based analyses can be performed in a unified environment, and which can be applied to large programs and test suites efficiently, and without limitations on programming language, granularity and coverage criteria.

We used this framework in some experiments in our testing research, and currently it includes support for some common algorithms such as test prioritization, reduction, fault localization and general test suite assessment by using coverage-related metrics. SoDA includes a set of efficient data structures implemented in C++, as well as various algorithm implementations for the mentioned tasks. We are also working on a graphical user interface, called *TAM (Test-suite Analysis and Manipulation)* which provides an easy to use access to most of the features. Furthermore, we included a carefully selected set of subject programs with associated test suites that have been prepared to serve as a benchmark in coverage-related research. It includes small, medium and large programs with appropriate measurement scripts, and we also provide some key measurement data for these programs.

Our long term goal with SoDA is to provide a general, extensible test suite analysis and manipulation framework to be used in industrial context, as well as a useful environment for coverage-based experimentation for researchers. Specifically, researchers could benefit from it thanks to the instantly available data from the program repository, the set of implemented algorithms, easy extensibility and the supporting GUI-based tool. On the other hand, developers and testers could find it useful for regression test suite maintenance including their assessment and optimizaton.

The paper is organized as follows. In Section II, we overview the main architecture and features of the SoDA framework, while more detailed description of the library and toolset will be given in Section III. The program repository with some measurements will be introduced in Section IV. We overview related work in Section V, and conclude in Section VI. Appendices at the end of the article provide information about the availability of SoDA and our planned tool demonstration session at the conference.

978-1-4799-6148-1/14 $31.00 © 2014 IEEE
DOI 10.1109/SCAM.2014.38

IEEE computer society

II. Overview

With the SoDA framework, we offer the following:

1) An efficient library for code coverage-related analyses with fundamental data structures and algorithms.
2) Appropriate adapters for accessing various coverage-related data produced by other tools. In GNU environment, most usual tools are already supported (e.g. parsing gcov coverage results, processing DejaGnu test reports), but the plugin architecture enables easy extension as well.
3) A set of algorithms for various applications including test selection, test prioritization and fault localization.
4) An implementation of a set of various metrics computable from the coverage data, which can be used for the assessment of test suites from quality aspects.
5) A graphical user interface based on the features of the library, whose overall goal is to analyse and manipulate test suites (the TAM application).
6) A subject program repository with tests and prepared analysis scripts.
7) Ready measurements on the programs from the repository, which can be used for research purposes.

The basic architecture of SoDA with the above-mentioned components is shown in Figure 1.

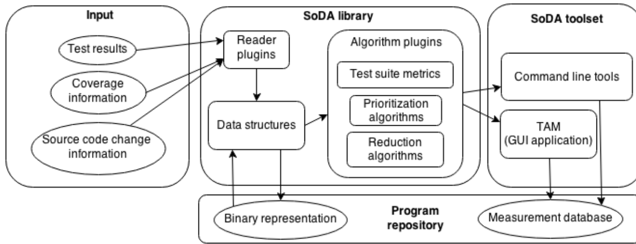

Fig. 1. Basic operation of the SoDA framework

A more detailed description of the library and the toolset will be given in the next section, together with a list of possible usage scenarios. We designed SoDA so that it would be useful for a broad range of users including researchers, professional developers and testers, QA assurance staff and (test) managers.

III. The SoDA Library and Toolset

The library and toolset parts of the SoDA framework can be further divided to data structures, algorithms, command-line tools and TAM, the graphical user interface, which will be separately introduced in the following. The *library* is implemented in C++ and contains the basic data structures for storing the coverage information, results of test executions and other aspects such as revision and program change information. The actual analysis algorithms are implemented also in the library in a *plugin*-oriented fashion, so that it can be easily extended with additional algorithms. Finally, the *toolset* part is where the actual user applications are located, which have two basic forms: command-line programs and the graphical user interface, TAM.

A. Data structures

SoDA stores the code coverage information in a special binary format. The format is designed so that it is able to store large amounts of data for large programs and test suites both in memory and on the disk. The library contains three main data structures: the coverage matrix, the results matrix and change information. The basic data structures are suitable for storing information of any kind of granularity of the source code and written in any kind of programming language, because we only define very general concepts such as code elements and test cases.

The *coverage matrix* is essentially a bit-matrix representing which test cases cover which code elements. Various matrix implementations are available for effectively working with sparse matrices, e.g [3]. However, in the case of code coverage matrices we cannot rely on any assumption about their sparsity (see the coverage densities of our subject programs in Table III), hence we designed general data structures for storing boolean values corresponding to the coverage. The library provides useful functions for easy access of coverage information of a test case or code element by using their names. There is also a possiblity for filtering the list of test cases or code elements, as well as some other utility functions.

The *results matrices* are used to store test execution results for the different revisions of the program. Each revision consist of one or more changes in the program, so a results matrix stores which test case was executed for each revision and what was the outcome of the test. Various utility functions are available for the results matrix as well.

SoDA includes a separate data structure to store *change information* about the subject programs, which is typically collected from version control systems such as SVN and Git. This information is useful for various test suite analysis tasks, e.g. test selection and test prioritization. The data structure for change sets is also based on a matrix representation, and is able to store the change information for multiple program elements and revisions.

B. Algorithms

The library includes algorithms usable in different testing activities, which can utilize code coverage information, such as test prioritization and selection, fault localization, etc. The plugin-structure of SoDA enables easy addition of new algorithms; either from the existing categories (like new prioritization strategies) or for completely new analyses.

The following are the main categories of algorithms currently implemented:

1) Test suite metrics: SoDA implements several metrics which can be computed for test suites to characterize them from different aspects, similarly to the traditional code metrics used for program code. We will present some of the availble metrics in the library, and for illustration, we will use an example coverage matrix from Figure 2.

The *code coverage* metric tells the percentage of how well the code elements are covered by the test cases, which is often used to guide test selection and test suite reduction activities. The value of this metric can be easily calculated based on the coverage matrix. Let T be the set of test cases and C be the set of code elements. Then,

$$\mathrm{cov}(T, C) = \frac{|\{c \in C \mid c \text{ covered by } T\}|}{|C|}.$$

$$\mathbf{C} = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \end{array} \begin{array}{c} \begin{array}{cccccc} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Fig. 2. An example coverage matrix (also called program spectrum). Rows represent the test cases and columns represent the code elements.

Another SoDA metric we used in some previous studies is the *partition metric*. We experimented with it in the context of fault localization [4]. It expresses the average portion of the program elements which are distinguishable from any given program element in terms of coverage. Code elements covered by the same test cases belong to the same partition. The partition metric of a particular program and its test cases is then derived using the structure of the partitions. Specifically, for a set T of test cases and a set C of code elements such a partitioning can be denoted with $\Pi \subseteq \mathcal{P}(P)$. We define $\pi_c \in \Pi$ for every $c \in C$, where

$$\pi_c = \{c' \in C \mid c' \text{ is covered by and only by the same test cases from } T \text{ as } c\}.$$

Having fault localization application in mind, $|\pi_c| - 1$ will be the number of code elements "similar" to c in the program, hence to localize c in π_c we would need at most $|\pi_c| - 1$ examinations [4]. Based on this observation, PART is formalized as follows:

$$\textsc{part}(T, C) = 1 - \frac{\sum_{c \in C}(|\pi_c| - 1)}{|P| \cdot (|P| - 1)}.$$

This metric takes a value from $[0, 1]$ similarly to the coverage metric, bigger meaning better.

SoDA includes derived metrics based on the former two as well, for example, to express the *optimality* of test suites. In this context, optimality deals with the number of test cases compared to some other attribute such as the number of program elements or another test suite metric. TPCE is the tests to code element ratio and it indicates how much test cases are written to test a code element on average:

$$\textsc{TpCE}(T, C) = \frac{|T|}{|C|}.$$

Next, efficiency metrics show what is the average contribution of one test case to some other test suite metric. For the coverage ($\textsc{Eff}_{\textsc{cov}}$) and partition metrics ($\textsc{Eff}_{\textsc{part}}$), efficiency is the following:

$$\textsc{Eff}_{\textsc{cov}}(T, C) = \frac{|\{c \in C \mid c \text{ covered by } T\}|}{|T|},$$

$$\textsc{Eff}_{\textsc{part}}(T, C) = \frac{\textsc{part}(T, C)\,|C|}{|T|}$$

Some example test suites with the corresponding partitions and the different test suite metric values of the example coverage matrix can be seen in tables II and I.

TABLE I. PARTITIONS OF THE EXAMPLE COVERAGE MATRIX

Test suite	Partitions
$\{t_1, t_2\}$	$\{\{p_1\}, \{p_2\}, \{p_3, p_4, p_5, p_6\}\}$
$\{t_3, t_4\}$	$\{\{p_1, p_2, p_3, p_4, p_6\}, \{p_5\}\}$
$\{t_5, t_6\}$	$\{\{p_1, p_2\}, \{p_3\}, \{p_4, p_5, p_6\}\}$
$\{t_7, t_8\}$	$\{\{p_1, p_2\}, \{p_3, p_4\}, \{p_5, p_6\}\}$
$\{t_1 \ldots t_8\}$	$\{\{p_1\}, \{p_2\}, \{p_3\}, \{p_4\}, \{p_5\}, \{p_6\}\}$

TABLE II. METRICS OF THE EXAMPLE COVERAGE MATRIX

Test suite	COV	PART	TPCE	$\textsc{Eff}_{\textsc{cov}}$	$\textsc{Eff}_{\textsc{part}}$
$\{t_1, t_2\}$	0.33	0.60	0.33	1.00	1.80
$\{t_3, t_4\}$	1.00	0.33	0.33	3.00	1.00
$\{t_5, t_6\}$	0.50	0.73	0.33	1.50	2.19
$\{t_7, t_8\}$	1.00	0.80	0.33	3.00	2.40
$\{t_1 \ldots t_8\}$	1.00	1.00	1.00	0.75	0.75

2) Test prioritization and selection: These methods enable selective and ranked retesting of regression tests to reduce the cost of test execution. In the SoDA library, we implemented several popular prioritization algorithms working on code coverage: *general coverage* and *additional coverage* [5], for instance, as well as some utility metholds like *random prioritization*.

3) Fault localization: Fault localization is a technique to help testers and developers in the process of testing, debugging and correcting a bug in the program by searching for the location of the fault in the system. A widely employed fault localization technique is the Spectrum-Based Fault Localization (SBFL) approach [6]. SBFL is based on the program spectrum, which is a signature of program behaviour and indicates which parts of the program were executed during a run [7]. The program spectrum is essentially the coverage matrix in our terminology. The technique observes the differences in execution profiles of the program for passed and failed test case executions and tries to find those code elements that may cause the tests to fail. SBFL uses a risk value to predict the relative risk of each code element containing the fault. The risk values can be computed in different ways by risk formulae. Several different methods defined such formulae in the literature, e.g. Tarantula, Ochiai and Jaccard [6]. In the SoDA library, the Tarantula and Ochiai SBFL methods have been implemented.

4) Test suite reduction: The goal of test suite reduction is to find a minimal subset of test cases that satisfy a requirement. Such a requirement can be to maximize the fault detection capability of the selected tests (often associated with code coverage). Another possible aspect is the fault localization capability, where the goal is to select test cases that help to localize a bug (this can be associated to the partition metric of test suites [4]). The problems are instances of the minimal hitting set problem which is NP-complete, so the reduction algorithms use some kind of heuristics to achieve the given requirements. Many reduction algorithms are based on test case prioritization by selecting the desired amount of test cases in the given order.

The SoDA library implements three test suite reduction algorithms. Two of them are fault detection driven-methods, namely the *Naive coverage-based* and *Additional coverage-based* methods [4]. They use the corresponding prioritization technique to determine the set of test cases that need to be included. The third algorithm is a fault localization-driven

method, whose goal is to keep the fault localization capability of the test suite as high as possible with a fewer number of test cases, and is called the *Partition-based* method [4].

C. Toolset

There are two types of user applications implemented on top of the SoDA library: command-line tools and a graphical user interface. Command-line tools provide access to all the implemented features of the library through a unified command-line interface. This way, the tools can be easily integrated into more complex analysis scripts and environments.

TAM (Test-suite Analysis and Manipulation) is a GUI application on top of the SoDA library that is designed to help researchers and developers investigating and manipulating test suites or creating assessment reports in an easy way. It uses the data structures and algorithms of the SoDA library and provides graphical interfaces for most of the functions. In addition, it includes various useful features such as configuration options for different measurements and support for investigating subparts of test suites graphically. Users can load multiple versions of a program into TAM and save their work into a workspace. To aid re-executing the measurements and performing various analyses on the same data, TAM saves the results into a database for later reuse. Figure 3 shows some examples of the TAM user interface.

Currently, TAM offers test suite analysis features but it is among our plans to implement automatic test suite manipulation functions as well.

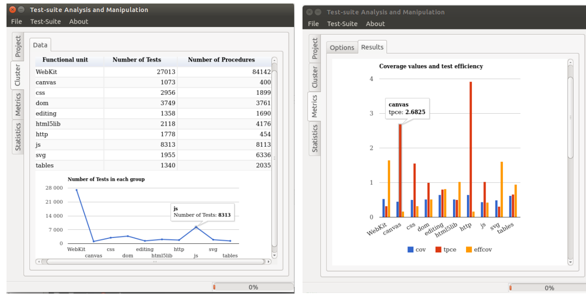

Fig. 3. TAM user interface.

D. Possible usage scenarios

The target users of SoDA (including TAM) are both researchers and professional developers and testers. Figure 4 shows the possible use cases of our system. Most features are accessible both from command line tools and TAM. Researchers can easily extend the library with new algorithms and compare the new methods with the available techniques implemented in SoDA. Developers can use test prioritization techniques to speed up their regression testing by selecting the relevant test cases that have to be executed. The list of possibly faulty elements provided by the fault localization methods could also be useful for developers. TAM provides an interface to easily find these elements and connects them with the source code. Test suite manipulation functions of TAM can make the testing process more efficient, which include investigating the functional units of the test suite separately and aiding the removal of duplicated test cases (in terms of coverage).

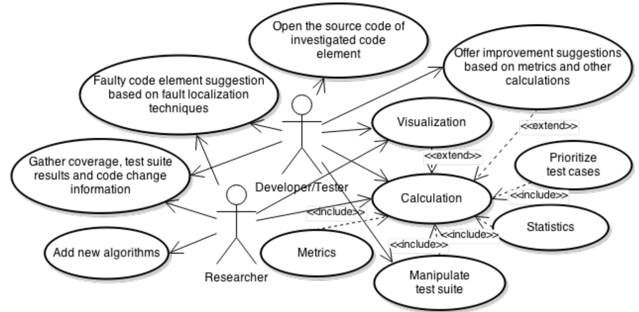

Fig. 4. Use cases of SoDA and TAM

1) Usage in previous studies: In a previous study [4], we used SoDA to empirically evaluate test suite reduction methods in terms of fault detection and localization. We investigated the effect of different test reduction methods on the performance of fault localization and detection techniques. We also provided new combined methods that incorporate both localization and detection aspects. In [8], we experimentally applied test selection methods to WebKit, a member of the SoDA program repository, to find out the technical difficulties and the expected benefits if this method is to be introduced into the actual build process. We presented results about the selection capabilities for a selected set of revisions of WebKit, and applied different test case prioritization strategies to further reduce the number of tests to execute. A preliminary version of SoDA was used in this study as well.

IV. The SoDA Program Repository

An important part of the SoDA framework is the *program repository*, a set of experimental programs with test suites and the corresponding measurement data produced by the library which can be used in code coverage-related research. The SoDA binary data files of the subject programs and the results of all measurements can be downloaded from the SoDA website [9]. Note that since SoDA is not tied to any specific analysis granularity, measurements on the subject programs can be performed on various levels such as statements and procedures. The readily available coverage data have been prepared at procedure level granularity. Also note that the SoDA binaries can be produced for programs written in any programming language by providing an input file in the required format or by extending the framework through plugins.

The repository includes various programs from small to large sizes in three groups (see Table III): programs from the SIR repository [10], three medium-size programs and two industrial-size open source systems, the GCC compiler and the WebKit web browser engine. All programs have associated test suites with pass/fail information, and we included several revisions with source code changes. In the first four columns of the table, we summarize the subject programs' basic statistics, while the 5th column presents *coverage density*, which is the percentage of the covered elements in the coverage matrix (ratio of positions where the value was 1).

The next five columns of Table III present metric values calculated by SoDA for these programs (see the definitions in Section III). The basic coverage and partition metrics were used in our previous research in relation to test reduction [4],

TABLE III. STATISTICS, TEST SUITE METRICS AND RESOURCE CONSUMPTION OF CALCULATION FOR THE SUBJECT PROGRAMS (* SEE TEXT)

	Statistics					Test suite metrics					Resource consumption	
Program	**Lines**	**Tests**	**Functions**	**Revs.**	**Cov. density**	**TPCE**	**COV**	**PART**	**EFF_{COV}**	**EFF_{PART}**	**Time (s)**	**Mem. (MB)**
printtokens	726	4 113	18	5	0.79	228.50	1.00	0.856	0.004	0.0037	0.03	2.51
printtokens2	570	4 098	19	10	0.89	215.68	1.00	0.608	0.004	0.0028	0.03	2.57
schedule	412	3 735	18	9	0.62	207.50	1.00	0.960	0.004	0.0046	0.03	2.30
schedule2	374	3 780	16	10	0.64	236.25	1.00	0.958	0.004	0.0041	0.03	2.47
space	6.2K	13 570	125	38	0.52	108.56	1.00	0.988	0.009	0.0091	0.33	8.22
tcas	173	2 239	9	37	0.45	248.78	1.00	0.889	0.004	0.0036	0.02	1.57
totinfo	576	1 036	7	20	0.70	148.00	1.00	0.952	0.006	0.0064	0.01	0.76
augeas	86.1K	273	784	278	0.40	0.35	1.00	0.913	2.872	2.6218	0.04	0.88
bison	34.3K	597	625	827	0.45	0.96	1.00	0.883	1.047	0.9239	0.06	1.29
dateutils	18.4K	522	349	883	0.18	1.50	1.00	0.990	0.669	0.6617	0.06	1.07
GCC	6.2M	128 230	20 372	2553	0.20	6.29	1.00	0.999	0.159	0.1587	575.35*	836.97
WebKit	4.5M	27 013	72 504	1907	0.08	0.37	0.65	0.869	1.758	2.3326	480.94*	784.06

while the others are part of our current research in the field of test suite quality. In particular, *TpCE* and the two efficiency metrics can reveal important information about the efficiency of test suites in terms of the number of tests required to achieve other attributes of the test suite.

Finally, we present data about runtime performance of SoDA for computing the above statistics in terms of execution time and RAM consumption (the measurements were run on Intel Core i5 CPU, 8GB of RAM with Ubuntu 14.04). The performance is clearly the function of the size of coverage information, which is in turn determined by the number of elements in the coverage matrix, in our case the product of the number of tests and procedures (functions and methods). The statistics are notable for the two big programs. The memory consumption is acceptable for these systems too, so we believe that it will be usable for other real size systems as well.

The execution time seems to be a bit long, however; but let us not forget that this includes the whole process starting from reading the coverage data from file, building up the internal data structures and performing the analyses. The average analysis times for GCC and WebKit were composed as follows: 11% was spent on loading the data from disk, 85% went to compute the partition metric and 4% to perform all the rest of the analyses. Computing the partition metric is computationally complex but it is not required in many applications, so we believe that the remaining computation times are accaptable; loading takes about a minute and computing most of the metrics is performed in seconds even for these large programs.

V. RELATED WORK

There are other frameworks and repositories related to testing activities as well. Yang *et al.* [11] prepared a survey of coverage-based testing tools. They compared the capabilities of 17 different tools including their own software, called eXVantage, and provided guidelines for developers to select the appropriate coverage testing tool. Sahid *et al.* [12] provided a study of the current test coverage research by investigating 47 papers between 2000-2010. Three of these studies proposed frameworks for test coverage measurement and analysis, but all of them have a different focus than that of SoDA, as we overview below.

Misurda *et al.* [13] provided a tool, called Jazz, that can be used to dynamically instrument code based on the requirements of coverage. They focused on the instrumentation process and defined their own language to guide the instrumentation. Bording *et al.* [14] proposed a framework called Couverture, which can produce the coverage of cross-compiled applications running in a virtualized environment. They focused on the creation of the coverage information, too. Sakamoto *et al.* [15] presented a framework that helps developing test coverage measurement tools, called Open Code Coverage Framework (OCCF). Their framework supports eight programming languages (C, C++, C#, Java, Javascript, Python, Ruby and Lua), and provides instrumentation support based on Abstract Syntax Trees to produce the coverage information. OCCF has been applied on fault localization and test minimization to demonstrate its capabilities. The main focus of these frameworks is on the creation of the coverage information, while with the SoDA library, we concentrate on the unified storage and processing of coverage data.

Campos *et al.* [16] developed an automated testing and fault localization tool to support testing and debugging, called GZOLTAR which is integrated into the Eclipse development environment as a plugin, and provides automatic instrumentation of Java code and calculation of coverage information. The main feature of the tool is spectrum-based fault localization based on the Ochiai algorithm. The SoDA library implements this algorithm as well, but it is more general and is not specific to any language or programming environment.

An infrastructure to support experimentations related to various regression testing techniques has been designed and constructed by Do *et al.* [10], who also made available their repository to other researchers. This repository, called Software Infrastructure Repository (SIR), has become very popular in the fields of regression testing and fault localization research because it includes the subject programs, the corresponding test suites and fault information as well. The drawback of this repository is that the included programs are typically very small, many test cases are artificial and most of the faults are seeded. In the SoDA framework, we included several programs from the SIR repository.

Another software repository was introduced by Shirabad *et al.* [17], namely the PROMISE Software Engineering Repository, whose purpose is to provide datasets that can be used in the area of predictive software models. Building such models is an important subfield of data mining and machine learning research. These datasets are publicly available to other researchers to support repeatable verifiable research in this area. We also provide the data publicly available that can be used as a benchmark for code coverage related research.

VI. Conclusions

SoDA, the framework presented in this article aims at providing useful features for researchers and practitioner developers and testers working with regression test suites of large applications. As the name suggests, however, SoDA is planned with a broader scope in mind; to include other features for the analysis of various software development artifacts, not only regression test suites and code coverage.

The current version of the framework provides solid foundations that proved in research projects. But, our experience is that even in research context one must pay attention to use efficient data structures and algorithms if the tool is to be applied to non-trivial programs. On the other hand, to reach the full potential of the framework in industrial practice, more development is needed. Therefore, we invite fellow researchers and practitioners to take part in the development of the framework.

We have a number of plans with SoDA, some of which are already listed on the website as topics for community contribution. Most notably, we envision the following extensions:

- additional metrics for the assessment of test suites,
- additional algorithms, e.g. test prioritization methods,
- more support for test suite manipulation,
- additional useful features in TAM.

In addition, we plan to establish a continuous measurement server for code coverage measurement and automatic test suite assesment of some open source projects' latest revisions, specifically for WebKit on a short term.

Appendix I

The current version of the SoDA library and toolset can be downloaded through the project's website [9] (the source code is hosted by Github at `https://github.com/sed-szeged`). All components of SoDA are released under the LGPL v3 license.

The SoDA homepage provides access to the program repository from Table III as well, including all related measurement data, which have been prepared by SoDA tools. This data is ready to be used in other research in connection with code coverage and regression test suite assessment, however the packages contain not just the results but the configuration files and scripts required to repeat the measurements as well.

Appendix II

We plan to demonstrate the features of the SoDA library and the TAM tool on the conference by performing the analysis of a small open source program. In addition, we will shortly overview the data we obtained for all the other programs from our repository including the large ones, GCC and WebKit.

The demonstration will include the following steps:

1) The code coverage information will be produced by executing some test cases with the example program.
2) The test results will be collected and the corresponding SoDA binaries will be created.
3) Some analyses (metric calculation and other algorithms) will be executed using command line tools.
4) The coverage and other results will be loaded into TAM to demonstrate them on the graphical user interface and perform additional analyses and identify suggestions for test suite optimization.
5) To emphasize the benefits of using the SoDA library, the analyzed program code and test cases will be investigated in parallel.

References

[1] L. Copeland, *A Practitioner's Guide to Software Test Design*. Artech House, 2004.

[2] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 85–103.

[3] "Blaze, an open-source, high-performance c++ math library," https://code.google.com/p/blaze-lib/, last visited: 2014-07-04.

[4] L. Vidács, Á. Beszédes, D. Tengeri, I. Siket, and T. Gyimóthy, "Test suite reduction for fault detection and localization: A combined approach," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 204–213.

[5] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.

[6] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *Search Based Software Engineering*. Springer, 2012, pp. 244–258.

[7] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," in *ACM SIGPLAN Notices*, vol. 33, no. 7. ACM, 1998, pp. 83–90.

[8] Á. Beszédes, T. Gergely, L. Schrettner, J. Jász, L. Langó, and T. Gyimóthy, "Code coverage-based regression test selection and prioritization in webkit," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 46–55.

[9] "Soda framework and repository," http://www.sed.inf.u-szeged.hu/soda, last visited: 2014-08-20.

[10] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[11] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, 2007.

[12] M. Shahid, S. Ibrahim, and M. N. Mahrin, "A study on test coverage in software testing," *Advanced Informatics School (AIS), Universiti Teknologi Malaysia, International Campus, Jalan Semarak, Kuala Lumpur, Malaysia*, 2011.

[13] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa, "Demand-driven structural testing with dynamic instrumentation," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 2005, pp. 156–165.

[14] M. Bordin, C. Comar, T. Gingold, J. Guitton, O. Hainque, T. Quinot, J. Delange, J. Hugues, and L. Pautet, "Couverture: an innovative open framework for coverage analysis of safety critical applications," *Ada User Journal*, vol. 30, no. 4, pp. 248–255, 2009.

[15] K. Sakamoto, K. Shimojo, R. Takasawa, H. Washizaki, and Y. Fukazawa, "Occf: A framework for developing test coverage measurement tools supporting multiple programming languages," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 422–430.

[16] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 378–381.

[17] J. S. Shirabad, S. Matwin, and T. C. Lethbridge, "Predictive software models," in *Software Technology and Engineering Practice, 2004. STEP 2004. The 12th International Workshop on*. IEEE, 2005, pp. 10–pp.