MOTSD: A Multi-Objective Test Selection Tool using Test Suite Diagnosability

Daniel Correia, Rui Abreu
daniel.b.correia@tecnico.ulisboa.pt,rui@computer.org
Instituto Superior Técnico, University of Lisbon
Lisbon, Portugal

Pedro Santos, João Nadkarni
{pedro.santos,joao.nadkarni}@outsystems.com
OutSystems
Lisbon, Portugal

ABSTRACT

Performing regression testing on large software systems becomes unfeasible as it takes too long to run all the test cases every time a change is made. The main motivation of this work was to provide a faster and earlier feedback loop to the developers at OutSystems when a change is made. The developed tool, MOTSD, implements a multi-objective test selection approach in a C# code base using a test suite diagnosability metric and historical metrics as objectives and it is powered by a particle swarm optimization algorithm. We present implementation challenges, current experimental results and limitations of the tool when applied in an industrial context. Screencast demo link: https://www.youtube.com/watch?v=CYMfQTUu2BE

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*; Software maintenance tools;

KEYWORDS

test selection, multi-objective, diagnosability, feedback

ACM Reference Format:
Daniel Correia, Rui Abreu and Pedro Santos, João Nadkarni. 2019. MOTSD: A Multi-Objective Test Selection Tool using Test Suite Diagnosability. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3338906.3341187

1 INTRODUCTION AND MOTIVATION

Regression testing plays a critical role during the development of software artifacts, both in the quality of the produced software but also in the effective development costs. However, regression testing large software systems is problematic as it takes too long to run all the test cases every time a change is made. For example, Google uses test case prioritization techniques to tackle this problem in a large scale software project [5].

The problem of improving the regression testing process has been extensively studied and the developed approaches tend to fit into one of three categories [7]: test suite reduction, wherein the focus is on identifying redundant test cases and removing them from the test suite; test prioritization, where we try to find an optimal ordering of test execution that maximizes certain objectives (e.g. number of faults detected for a limited block of time); test selection, which consists in selecting an appropriate subset of the test suite in order to execute only the most relevant tests.

These types of techniques have been applied in industry [3, 5] to successfully reduce regression testing costs, both in terms of computational resources and developer time. However, reducing the diagnostic cost for the developers when a test fails (i.e. the cost of finding the root cause of a test failure and fixing the bug in the code) is typically not addressed by the evaluation metrics that guide these regression testing techniques.

This work targets the reduction of regression testing costs and diagnostic costs in the industrial context of OutSystems[1] development processes. The OutSystems platform is a highly complex, monolithic software product that has evolved over many years and current development efforts are supported by a CI pipeline using a large test suite. At OutSystems, the build time of the CI pipeline is a problem. This is caused not only by the large code base size (over 1 million lines of code) but also the high execution cost of the test suite. On average, a developer has to wait 40 minutes before a change passes through the first main stages of the CI pipeline, despite the usage of additional computational power and parallel computation strategies to speed up the testing process.

Hence, the main motivation of the tool developed for this problem was to achieve significant improvements in development speed and productivity by providing a faster and earlier feedback loop to the developers when a change is made. This would in turn motivate further improvements in OutSystems development processes such as pre-commit validations and migration to Git.

The core idea of the developed tool (illustrated in Figure 1) is that we can build a shorter feedback loop than the original CI pipeline by executing a subset of the test suite, relevant to the specific changes in a pre-commit stage, before passing them through the CI pipeline. Additionally, since a test suite diagnosability metric is used to guide the test selection process, the diagnostic cost should be reduced since the cost of fault localization is lower.

This tool demonstration paper focuses on the features and implementation details of the developed test selection tool called MOTSD. In particular, we highlight several problems encountered when implementing test selection techniques in an industrial context, both in the scalability of the coverage extraction processes but also in the theoretical assumptions of chosen techniques and metrics.

Finally, current experimental results over OutSystems' large code base are presented and discussed in terms of the tool's objectives: (1) the selected tests should correspond to the failing tests; (2) the tool

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia*, https://doi.org/10.1145/3338906.3341187.

[1] Introducing OutSystems blog post: https://www.outsystems.com/blog/posts/introducing-outsystems-11/ (accessed May 2019)

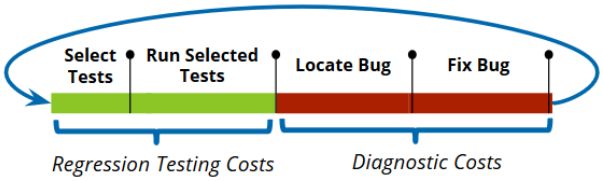

Figure 1: Core idea of MOTSD - early and shorter feedback loops in a pre-commit stage

should respond with a selection quickly enough to be integrated into a development process; (3) the size of the selected subset of tests should be small enough to provide a much faster feedback loop than the original process (i.e. the CI pipeline).

2 FEATURES

Summary. MOTSD is a test selection approach that was developed using a test suite diagnosability metric and historical metrics to guide the selection process. This is modelled as a multi-objective problem and a particle swarm optimization algorithm is used to find good solutions for this optimization problem. The activity matrix (i.e. coverage data) provided to MOTSD is filtered according to the changes introduced by the commit using file-level granularity.

Test Suite Diagnosability Metric. While classical test coverage metrics can be used to describe which parts of a system have been exercised, they provide no diagnostic help in case of regression and have limited applicability to multiple-faulted systems. On the other hand, test suite diagnosability metrics evaluate test suites based on the cost of diagnosing the cause of a fault if a test case from the given test suite fails. The DDU metric combines ideas from previous diagnosability metrics in order to capture the structural properties of the activity matrix w.r.t. its diagnosability [6].

Multi-Objective. Real-world problems are typically multicriteria wherein multiple conflicting goals (or objective functions) need to be satisfied and/or optimized simultaneously [8]. The described regression testing problem fits in this domain since there are several goals to be optimized, for example, minimize total execution time and maximize fault detection capability of the selected test suite. In multi-objective problems obtaining an optimal solution is impossible: instead, a set of solutions with the best trade-offs between each objective is provided. From these solutions, a decision maker can then choose a solution for the problem using a preference strategy.

Particle Swarm Optimization. Particle swarm optimization (PSO) is a global optimization technique from the field of swarm intelligence inspired by the social behavior of flocks of birds [2]. PSO is similar to genetic algorithms in the sense that the system is described by a population of solutions and new solutions are generated through perturbation of existing solutions. These perturbations are directed towards the current best solution according to the solutions encountered by each particle and by the whole swarm. PSO was chosen due to its simplicity, fast convergence rate and good adaptability to multi-objective settings [1].

3 IMPLEMENTATION

This section describes the implementation details of MOTSD regarding the challenges of extracting coverage data, the components of the test selection pipeline and the required integration steps with external systems (database and version control system). Figure 2 illustrates the architecture and information flow.

OutSystems Context. OutSystems' code base is built on top of a C# stack across one million lines of code and is accompanied by a few smaller modules implemented in Typescript. This means that even though most of the files changed during development will be C# files, there is a significant amount of changes to other types of files (e.g Typescript, XAML, XML, JSON, CSS) related to configuration mechanisms or interface development. The test suite used by OutSystems contains over 8500 tests implemented in NUnit[2]. These tests are split into three stages based on their complexity and execution overhead: (1) Development - 3000 *fast* tests with total execution time of 5 minutes; (2) Core - 5000 component and integration tests; (3) System - 500 interface and end-to-end tests which take a few hours to run.

Coverage Data Extraction. Regarding coverage data collection, we needed a coverage tool that could support C# code bases and report coverage results for each test. Thus, we used the open source tool OpenCover[3] which provides code coverage with a fine granularity (instruction level).

However, using OpenCover over OutSystems' large code base and test suite raised a significant challenge regarding execution time overhead and size of the generated XML coverage report. For example, most of the serialized XML attributes and elements are unnecessary since we only need to extract test-method coverage relations from the report. Furthermore, OpenCover is too fine grained to scale to the OutSystems code base: building the activity matrix using method calls granularity provides a reasonable trade-off between instrumentation overhead and coverage precision.

To answer these issues, we modified[4] OpenCover's XML serialization to minimize the size of the generated reports and changed the instrumentation logic such that only the first instruction was instrumented (i.e. the method call).

These changes provided significant benefits in terms of execution time and coverage report size overheads. For example, using only the 3000 tests from the Development stage (which take 5 minutes to run without instrumentation), we were able to reduce the total execution time by 56% - from 32 minutes with the original OpenCover to 14 minutes with the modified version. We also observed that the generated coverage report size was reduced from 2855 MB to 168 MB (i.e. a reduction of 94%).

Test Selection Pipeline. The core component of MOTSD is the test selection pipeline which, given a set of code changes from a commit, provides a subset of tests to be executed based on coverage data (represented by an activity matrix linking tests with methods) and historical metrics mined from build history.

The execution of the pipeline assumes that the extraction of both coverage data and historical metrics was done in a offline stage (e.g in a nightly build). This is important to reduce the runtime overhead

[2] https://nunit.org/ (accessed May 2019)
[3] https://github.com/OpenCover/opencover (accessed May 2019)
[4] Comparison of changes with original OpenCover: https://github.com/OpenCover/opencover/compare/4.6.519...danielcorreia96:oc_2016_merged (accessed May 2019)

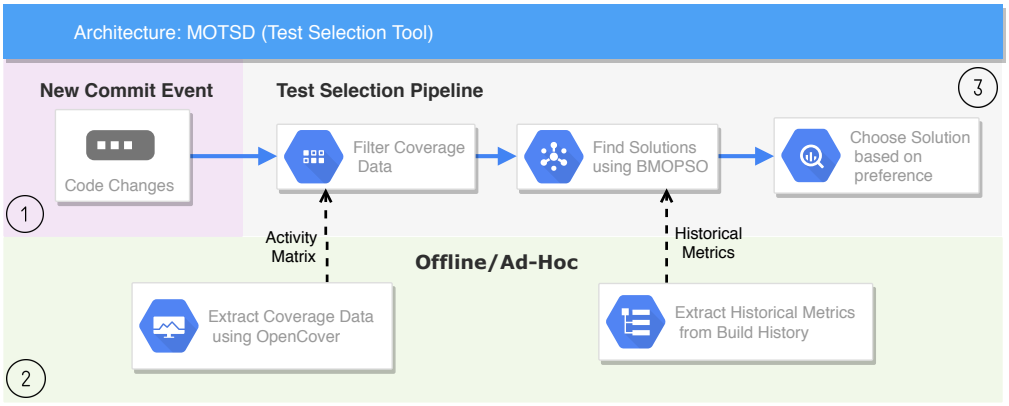

Figure 2: Architecture of MOTSD separated into three domains: (1) input event for the target commit, (2) offline stage for data extraction, (3) the multi-objective test selection pipeline.

of the pipeline by moving the overhead of database querying and coverage data extraction to an earlier ad-hoc stage.

The first step of the pipeline is initiated by the arrival of a new commit event. Based on the code changes of the commit, the pipeline filters the activity matrix using a file-level granularity such that the resulting activity matrix only contains methods from the changed files. To further reduce the size of the activity matrix, we apply a second filter that removes the tests with no coverage information in the filtered matrix. Since the OutSystems code base is version controlled using SVN, we used PySvn[5] to be able to obtain the set of code changes from the input commit event.

The second step takes the filtered activity matrix and, using a multi-objective algorithm, returns a list of subsets of tests that optimize the given metrics (DDU and historical metrics). Since this is a discrete optimization problem (i.e. a test is either selected or not selected), we solve the test selection problem using a Binary Multi-Objective Particle Swarm Optimization (BMOPSO) implementation backed by JMetalPy[6] APIs and based on the paper by Kennedy et al. in [2]. Currently, only the following objectives were implemented: (1) maximize DDU, (2) maximize method coverage, (3) minimize total number of tests, (4) minimize total estimated execution time, (5) maximize total test failures based on previous builds.

Finally, using the list of solutions found by the BMOPSO algorithm, we apply total ordering to sort the solutions by preference of the applied objectives and select the best solution. For example, assuming an execution of the pipeline using DDU and total number of test fails, one possible preference order would be to first sort the solutions by the highest DDU values and then sort by the highest total number of test fails. The final output is a list of tests to be executed before passing the commit to the CI pipeline.

4 RESULTS

This section describes preliminary results obtained through experiments done in the industrial context of OutSystems and presents a discussion on the limitations of MOTSD. To perform these experiments we decided to execute MOTSD in past versions of the OutSystems' code base. Specifically, we selected a 1-month period of development activities, generated an activity matrix at the start of the period and ran the tool for all the commits until the end of the period. To evaluate the results of the tool, we compared them with the results stored in the build history.

Table 1 shows the results obtained for 4 different 1-month periods of development using a test suite of around 8000 tests. For performance evaluation, we measured precision and recall since the test selection problem is similar to an Information Retrieval problem where a query (i.e commit/changelist) is provided to the system and a set of relevant documents (i.e. set of selected tests) is returned. For the purpose of this work, a relevant document corresponds to a test that failed in the past for a certain commit's build. Precision evaluates how many of the selected tests are relevant and recall evaluates how many of the relevant tests are selected [4].

Limitations. The results highlight several limitations when applied to an industrial context both in terms of how often it can produce results (on average, MOTSD executes on less than 50% of the commits) and how often it can actually find failing tests (see low Recall values in Table 1).

Regarding cases where the tool does not execute, we detected 3 error cases: (1) the commit changed no C# files (.cs extension) meaning they could not be mapped to the activity matrix; (2) the activity matrix did not have coverage data for the changed files/classes; (3) the commit only added or changed new files for which the tool had no information to work with.

The first error is caused by the fact that we only consider coverage data from C# code when performing test selection.

The second error stems from the assumption that the activity matrix provided covers most of the system's components. This is unrealistic since covering all possible code execution paths in a large scale system would require too many test cases and lead to a significant testing overhead. Additionally, some code may be untrackable due to external service calls. Finally, when we optimized OpenCover to reduce coverage extraction overhead, we removed the XML serialization of elements like getters, setters and anonymous types (which could increase the coverage data available).

[5] https://github.com/dsoprea/PySvn (accessed May 2019)
[6] https://github.com/jMetal/jMetalPy (accessed May 2019)

1-Month Period	# Commits	# Tool Executions		Tool Execution Errors			Tool Found Failing Tests?			Solution Size				Average Selection Time (s)	Average Feedback Time (s)
		Total	Total %	# No .cs Files	# No Cov. Data	# New Files	Yes, at least one	Precision	Recall	Average	Min	Max	Std		
July 2018	469	237	51%	71	146	15	32%	1%	22%	904	2	2104	744	12	801
October 2018	336	116	35%	67	133	20	31%	0%	21%	1210	5	2595	865	14	1211
February 2019	463	111	24%	126	196	30	36%	1%	23%	1447	8	2737	994	19	1211
March 2019	579	174	30%	203	153	49	46%	1%	26%	1639	3	2536	951	17	1140

Table 1: Results obtained for a set of experiments on the OutSystems code base in 2018-2019. The objectives used for optimization were (1) Maximize DDU and (2) Maximize Total Test Failures.

The third error reveals a lack of adaptability to unknown data. In theory, this issue could be minimized by frequently updating the activity matrix. However, this is unfeasible since generating the activity matrix for such a large code base takes too long due to the overhead of executing the tests with instrumentation.

Performance and Objectives The experimental results provide some insights onto the objectives defined for the tool. For example, in terms of ability to select failing tests, MOTSD has a much lower performance than would be desirable (max. 26% recall) and we are currently investigating if there are cases where the tool should have been able to select more failing tests. In addition, if we consider that finding at least one of the failing tests is enough for the selection to be successful, then the results are a bit more positive (reaching a max of 46%).

The size of returned subsets of tests and the average time to perform a selection show that MOTSD is not only fast enough to be integrated into an existing development workflow (on average, returns a selection in less than 10 seconds), but also that the number of selected tests is much smaller than the original test suite, which supports our motivation of a shorter feedback loop.

This is supported even further by the newly observed feedback time, i.e. the time to actually execute the selected tests and receive feedback on them. On average, MOTSD was able to provide a feedback time of 1200 seconds (20 minutes). This is significantly better than the original feedback time - 90 minutes. For the sake of simplicity, the feedback time was evaluated assuming a sequential execution of tests on a single process (i.e. any parallel computation or commit aggregation strategies were ignored).

Regarding the large range of solution size values (min, max and std), these are caused by the presence of very heterogeneous commits in the code base: some commits change a lot of files leading to larger solution sizes, while other commits change only one or two files resulting in smaller selections.

Tool and Data Availability. The source code of MOTSD is available in GitHub[7] with a more detailed explanation of the dependencies and configuration steps required. Keep in mind that it is in an experimental stage and some implementation assumptions/limitations still exist. Regarding experimental data, we are unable to make it available since all of the experimental data consists of OutSystems property: starting at the source code from which coverage data was extracted, to the historical metrics that were mined from build history, and finally to the commit history used to perform experiments to evaluate the tool.

Future Work. This work revealed several concerns regarding applicability to cross-language code changes (i.e. non C# files) and the scalability bottleneck from coverage data extraction. To answer these concerns, one possible improvement would be to use some other source of data than coverage information. For example, the activity matrix could be modelled using a dependency graph built by analyzing the version history and linking files that are changed in the same commit[8]. This graph could be further enriched with build history information by linking changed files with tests that failed in the respective commit.

A second challenge that was raised during this work was the evaluation of the tool, for which we still have many open questions. For example, how should the tool be evaluated on non-failing commits? Additionally, how do we protect the evaluation results against the "innocent commits problem", i.e. an unrelated commit may be linked to failing tests that were broken in previous commits.

Finally, to the best of our knowledge, there are no benchmarks in the literature on which to evaluate any test selection tool. While we understand that developing such a benchmark has several difficult challenges (e.g. multiple languages, CI tooling and version control systems), we feel that the lack of a benchmark limited our ability to compare the effectiveness of MOTSD to existing approaches.

5 CONCLUSIONS

This paper presents a multi-objective test selection tool developed for the industrial context of OutSystems. The developed tool uses a recently proposed test suite diagnosability metric, DDU, instead of a classical code coverage metric in order to tackle the diagnostic cost of the selected subset of tests. Several challenges were revealed when implementing this type of tool in a large scale industrial project both in terms of the code coverage tools overhead and the limitations of the proposed test selection approach. The experimental results obtained show that, despite the high number of industrial constraints and the underlying monolithic organization of the test suite, this approach was still able to return fairly accurate selections which in turn would lead to much faster feedback times.

ACKNOWLEDGMENTS

This work was supported by OutSystems under a master thesis collaboration project.

[7] https://github.com/danielcorreia96/MOTSD (accessed May 2019)

[8] For example, this could be done using hercules https://github.com/src-d/hercules

REFERENCES

[1] Carlos A Coello Coello, Gary B Lamont, and David A Van Veldhuizen. 2007. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Vol. 5. Springer.

[2] J. Kennedy and R. C. Eberhart. 1997. A Discrete Binary Version of the Particle Swarm Algorithm. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, Vol. 5. 4104–4108 vol.5. https://doi.org/10.1109/ICSMC.1997.637339

[3] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2018. Predictive Test Selection. *CoRR* abs/1810.05286 (2018). arXiv:1810.05286 http://arxiv.org/abs/1810.05286

[4] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. Evaluation in information retrieval. In *Introduction to Information Retrieval*. Cambridge University Press, Chapter 8, 151–175.

[5] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. IEEE Press, Piscataway, NJ, USA, 233–242. https://doi.org/10.1109/ICSE-SEIP.2017.16

[6] Alexandre Perez, Rui Abreu, and Arie van Deursen. 2017. A Test-suite Diagnosability Metric for Spectrum-based Fault Localization Approaches. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 654–664. https://doi.org/10.1109/ICSE.2017.66

[7] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[8] Aimin Zhou, Bo-Yang Qu, Hui Li, Shi-Zheng Zhao, Ponnuthurai Nagaratnam Suganthan, and Qingfu Zhang. 2011. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation* 1, 1 (2011), 32 – 49. https://doi.org/10.1016/j.swevo.2011.03.001