Applying Multi-Objective Test Selection for Continuous Integration at OutSystems

Daniel Correia
daniel.b.correia@tecnico.ulisboa.pt
Instituto Superior Técnico, University of Lisbon
Lisbon, Portugal

ABSTRACT
Performing exhaustive regression testing whenever a change occurs in large software systems tends to be unfeasible as it takes too long to run all the test cases. The main motivation of this work is to provide a shorter and earlier feedback loop to developers when a change is made, instead of having to wait for the currently slower feedback from a Continuous Integration pipeline execution.

The developed technique and tool, MOTSD, implements a multi-objective test selection approach for C# codebases. MOTSD uses a particle swarm optimization algorithm to search for relevant selections and uses a test suite diagnosability metric and historical metrics as objectives. When compared with random selections, MOTSD is able to match in terms of ability to select failing tests while also providing, in at least 25% of the commits, selections that were, on average, 93% smaller and 76% faster to execute.

CCS CONCEPTS
• Software and its engineering ▶ Software testing and debugging; Software maintenance tools;

KEYWORDS
test selection, multi-objective, diagnosability, continuous integration

ACM Reference Format:

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 far from trivial as it usually takes a prohibitively amount of time 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 [11]:

- **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. Unfortunately, reducing the diagnostic cost for the developers when tests fail (i.e., the cost of finding the root cause of test failures and fixing the bug in the code) has not been 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 OutSystems1 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.

---

The main goal of applying test selection techniques is to reduce the time required to retest a modified software artifact by selecting a subset of the existing test suite to be executed instead of executing the whole test suite [8].

When compared with random selections, MOTSD is able to match in terms of ability to select failing tests while also providing, in at least 25% of the commits, selections that were, on average, 93% smaller and 76% faster to execute. This paper makes the following contributions:

1. A multi-objective test selection approach, MOTSD, is proposed using a test suite diagnosability metric (DDU [6]) and historical metrics as objectives.
2. A toolset implementation is provided on GitHub.
3. MOTSD is evaluated on the industrial context of OutSystems in comparison with random selections.
4. Several integration use cases and future research work for test selection problems are proposed.

### 3 RELATED WORK

In this section some of the existing regression test selection techniques are discussed, giving particular attention to approaches that use PSO or multi-objective mechanisms. The following list is however not exhaustive.

Souza et al. in [1] proposed a constrained PSO algorithm for test case selection based on requirements coverage in which the execution effort is considered as a constraint during the search process. In this algorithm, each particle is represented by a binary vector $x = (x_1, ..., x_n)$ where $x_j \in 0, 1$ indicates the presence (1) or absence (0) of a test $j$ in the selected test suite. The results of this work indicate that the performance of an algorithm with these features (constraints, local search hybridization) is mostly dependent on the characteristics of the test suite used and the constraints imposed on the problem.

Yoo et al. [10] introduced the concept of Pareto efficiency in the context of test case selection. In fact, this paper presented the first multi-objective formulation of the test case selection problem. The experimental results and case studies revealed the benefits of developing Pareto-based approaches for test case selection problems.
whether the approach answers (or not) each integration concern.

In a perfect world, this would not be a problem since the system would be modularized according to functional properties of the product and the resulting test suite would be split into specific test targets (which would only need to be executed if their modules changed). Thus, for any change a developer would make, it would be trivial to know which tests needed to be run.

Evidently, the industrial context of OutSystems and existing monolithic organization of the code base, is not a perfect world environment. Hence, the developed solution should be able to provide insights into the underlying structural properties of the system, without requiring manual functional separation of the product into proper modules.

4 INDUSTRIAL CONTEXT

Motivation. The main goal of this work is the reduction of regression testing and diagnostic costs at OutSystems. In particular, this work targets the feedback loop that developers have when making changes in the OutSystems’ code base.

A key concern of developers is that, after making a set of changes, they would like to know which tests they need to run. This is a real problem due to the large code base (and regression test suite) and the inherent complexity of a monolithic system.

In a perfect world, this would not be a problem since the system would be modularized according to functional properties of the product and the resulting test suite would be split into specific test targets (which would only need to be executed if their modules changed). Thus, for any change a developer would make, it would be trivial to know which tests needed to be run.

Evidently, the industrial context of OutSystems and existing monolithic organization of the code base, is not a perfect world environment. Hence, the developed solution should be able to provide insights into the underlying structural properties of the system, without requiring manual functional separation of the product into proper modules.

4.1 OutSystems

Code Base. 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.

Test Suite. The test suite used by OutSystems contains over 8500 tests implemented in NUnit3. These tests are split into three stages based on their complexity and execution overhead: (Development) 3000 fast tests with total execution time of 5 minutes; (Core) 5000 component and integration tests; (System) 500 interface and end-to-end tests which take a few hours to run.

Trunk-Based Development. An additional concern of the proposed solution is that it should be possible to integrate it into existing development processes. In the case of OutSystems, trunk-based development is used, which means that all developers commit to the same branch of code.

This is supported by an automated pipeline that continuously checks out a version of the code and runs regression testing stages on it. On average, this means that a developer at OutSystems has to wait 40 minutes before his/her change passes through the first main stages of testing (slower tests like interface and system tests are not included in this group).

4.2 Integration Use Cases

Integrating MOTSD into existing development processes was not implemented since it would require too much domain-specific knowledge of the OutSystems build system and processes. Hence, some integration use cases are presented to illustrate how MOTSD can be applied in real software development scenarios.

Selected Tests Execution (Pre-Commit). The first use case matches the main focus of this work: providing a shorter feedback loop to developers by executing a specific subset of tests in an earlier stage. Thus, this use case considers the execution of selected tests in an earlier stage (e.g., pre-commit), before the commit is passed to existing CI pipeline.

However, there are several ways this use case can be implemented in practice, with different trade-offs in terms of additional infrastructure costs, performance constraints and developer experience. Concretely, there are two tasks that need to be completed: (1) selection of the tests using a set of changes as input; (2) execution of the selected subset of tests. Table 1 presents a summary of the possible integration approaches.

Commit Blaming (CI). Since OutSystems uses trunk-based development, every developer commits changes to the same common branch of the code base. Additionally, when the CI pipeline executes a new iteration of a test stage, it aggregates all the new commits in order to validate the most recent version of the code.

In practice, this means that when tests break, all the associated commits will be marked as “guilty”. This is a significant problem since all developers linked with these commits will be notified with equal weight of blame, even though in most scenarios only a few of the commits are actually responsible for breaking the tests.

Thus, this use case considers the usage of the proposed test selection pipeline to provide insights into which commit should be blamed for each test breakage. This approach has the advantage of having zero overhead of test execution, since the selection results are sufficient to link the commits with the each test case.

Test Case Prioritization (CI). Finally, the results of the test selection solution can be used to optimize the execution order of tests in the CI pipeline. Although this does not tackle the main goals of this work, it is still a valid use case since the returned selections of tests are expected to be the most relevant to be executed as soon as possible (which fits the idea of test case prioritization).

---

Table 1: Summary of the integration approaches for the “Selected Tests Execution” use case. The ✓ and × are used to describe whether the approach answers (or not) each integration concern.

<table>
<thead>
<tr>
<th>Integration Approach</th>
<th>Deployment</th>
<th>Maintenance</th>
<th>Infrastructure Costs</th>
<th>Developer Experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Server-side</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Server Selection + Client Execution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Client Selection + Server Execution</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Full Client-side</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

1https://nunit.org/ (accessed May 2019)
5 IMPLEMENTATION

This section describes the implementation details of MOTSD regarding the challenges of extracting coverage data and the components of the test selection pipeline. Figure 2 illustrates the architecture and information flow.

Coverage Data Extraction. The open source tool OpenCover was used to extract coverage data since it supports C# code bases and reports coverage results for each test.

However, using OpenCover was challenging due to the execution time overhead and the size of the XML coverage report. For example, most of the serialized XML attributes and elements are unnecessary since only the test-method coverage relations need to be extracted from the report. Furthermore, OpenCover’s instrumentation is too fine-grained to scale to large code bases.

To address these issues, OpenCover was modified to minimize the size of the coverage reports and the instrumentation logic was changed to target only the first instruction (i.e., the method call).

These changes provided significant benefits: (1) the total execution time was reduced by 56%, from 32 minutes to 14 minutes with the modified version and (2) the coverage report was reduced from 2855 MB to 168 MB (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 an offline stage (e.g., in a nightly build). This is important to reduce the runtime overhead 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, a second filter is applied that removes the tests with no coverage information in the filtered matrix. Since the OutSystems code base is version controlled using SVN, PySvn was used to obtain the set of code changes from the input commit event.

The second step takes the filtered activity matrix and returns a list of subsets of tests that optimize the given objectives (using DDU and historical data). Since this is a discrete optimization problem (i.e., a test is either selected or not selected), the test selection problem is solved using a Binary Multi-Objective Particle Swarm Optimization (BMOPSO) implementation backed by JMetalPy APIs and based on the work by Kennedy et al. in [2].

In terms of objective functions, MOTSD implements five objectives which can be split into 3 categories: (1) Structural objectives that consider the coverage information extracted from the system; (2) Historical objectives that evaluate the candidate selections according to metrics obtained from previous builds; (3) Basic objectives which are independent of the coverage or historical data available. Table 2 describes the objective functions implemented in MOTSD for each of these categories.

Finally, using the list of solutions found by the BMOPSO algorithm, total ordering is applied 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.

6 EVALUATION

Dataset. The experimental results were obtained using OutSystems’ code base, test suite and historical data. Together, these artifacts make up the dataset that MOTSD was evaluated on. Although none of these components can be made available publicly, the dataset explanation provided in Section 4.1 should suffice to understand how the characteristics of the available data may influence the presented results.

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.

**Methodology.** Regarding the evaluation methodology used, the idea is straightforward: (1) run the tool for a set of commits, (2) compare the selected tests with known past test execution outcomes for each commit and (3) measure the evaluation metrics chosen for this problem.

Specifically, MOTSD was evaluated in four different 1-month long periods. For each period, coverage and historical data was collected only once at the start and this data was used to run MOTSD for all commits during the 1-month period.

The choice of 1-month was intentional since using longer periods would likely result in much worse results in the later commits of the period. In contrast, using shorter periods like 1-week long would not be feasible with the available hardware for experimentation.

Lastly, all experiments were conducted on a single machine with 16GB of RAM and a 4-core Intel i7 2.60GHz CPU.

**Evaluation Metrics.** The evaluation of the obtained results was done according to a set of metrics that fit into 3 categories: (1) frequency of error cases; (2) ability to detect failing tests; (3) additional metrics related to tool execution performance.

MOTSD encountered 3 types of errors when executing over each commit considered in the evaluation period. These error categories were obtained by manually diagnosing the tool crash reports and understanding their common root cause.

- **No .cs Files**: this error happens when a commit does not change any C# code file. Since MOTSD relies on an activity matrix built using only C# code coverage, this type of change results in an empty matrix after filtering.
- **No Coverage Data**: assuming at least one C# code file was changed, it may still happen that the activity matrix contains no coverage information for the changed files. This can be caused by several factors such as non-perfect test coverage over the system or external service calls.
- **New Files**: it may happen that a commit only adds or modifies new files during the evaluation period. This happens because the coverage data is extracted once at the beginning of the period and never updated.

The second category evaluates a simple question in the context of red commits (i.e., commits associated with failing tests): was the tool able to find the failing tests? With that in mind, the following metrics were considered: "Yes, at least one", Macro-Recall and Micro-Recall.

"Yes, at least one" refers to how often MOTSD was able to find at least one of the failing tests. This metric evaluates the effectiveness of MOTSD optimistically since, ideally, the tool should select all failing tests and not only one of them. However, in practice, having one test failing is good enough for the developer to be forced to fix the test before re-committing. In addition, when multiple tests fail at the same time, it is fairly common that a fix that targets only one of the tests, also fixes the remaining tests.

The next 2 metrics, Macro-Recall and Micro-Recall, come from the IR field since the test selection problem is similar to a document retrieval problem from IR, wherein the system returns a set of relevant documents for a given input query. Following the analogy, MOTSD is a system that returns a set of relevant tests (documents) for a given commit (query).

Manning et al. [4] present two measures for IR system effectiveness: Precision and Recall. Hence, for each commit $c$, these measures are calculated as presented in (1).

$$Precision(c) = \frac{\#\text{ failing tests selected}}{\#\text{ selected tests}}$$
$$Recall(c) = \frac{\#\text{ failing tests selected}}{\#\text{ failing tests}}$$

For the purpose of this work, Precision was not considered since it does not provide any relevant insights into the effectiveness of MOTSD. During experimental evaluations, the Precision values were very low (around 1%). This happened because usually only a few tests fail, even though MOTSD may be correct in classifying a larger subset of tests as relevant for a set of code changes.

Regarding Recall, the results of each commit were aggregated using two different strategies described in (2): macro-average (Macro-Recall) and micro-average (Micro-Recall).

$$Macro-Recall = \frac{\sum_c Recall(c)}{\#\text{ commits}}$$
$$Micro-Recall = \frac{\sum_c \# \text{ failing tests selected for } c}{\sum_c \# \text{ failing tests for } c}$$

<table>
<thead>
<tr>
<th>Category</th>
<th>Objective Function</th>
<th>Label</th>
<th>How to calculate?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural</td>
<td>DDU (Max)</td>
<td>D</td>
<td>DDU of the activity matrix with the selected tests</td>
</tr>
<tr>
<td></td>
<td>Normalized Coverage (Max)</td>
<td>N</td>
<td>Add activity contributions per method, normalize to 1s/0s and calculate coverage.</td>
</tr>
<tr>
<td>Historical</td>
<td>Test Fails (Max)</td>
<td>F</td>
<td>Sum the number of test fails in previous builds.</td>
</tr>
<tr>
<td></td>
<td>Execution Time (Min)</td>
<td>E</td>
<td>Sum the execution time of each selected test.</td>
</tr>
<tr>
<td>Basic</td>
<td># Tests (Min)</td>
<td>T</td>
<td>Number of tests in the candidate selection.</td>
</tr>
</tbody>
</table>

Table 2: Objective functions implemented in MOTSD.
Looking at both Macro-Recall and Micro-Recall is important since at OutSystems there is a significant portion of commits that either (1) break hundreds of tests at the same time or (2) only break a single test. In the first scenario, even if MOTSD is able to select a large number of failing tests the recall value for that commit will still be very low. The second case also happens frequently, for example with flaky tests.

Note that for the remaining green commits (i.e., commits without failing tests) this group of metrics is not applicable and only the additional metrics presented in the next section were calculated.

One possible improvement would be to look at repairing commits, i.e., a green commit after a red commit. The idea here would be to see if MOTSD was still able to select the relevant tests in a repaired scenario. In other words, if a test was failing and the next commit fixes that test, then MOTSD should return a selection containing that test.

Additional Metrics. Finally, a few additional metrics were measured to study other characteristics of MOTSD:

1. Solution Size: size of the returned selection of tests
2. Computing Time: how long it took MOTSD to run the selection pipeline and return a selection of tests
3. Feedback Time: how long it would take to execute the selected tests and provide feedback to the developer.

Keep in mind that the Feedback Time metric was approximately calculated since executing the tests for every commit would require too many computational resources. Ergo, since the execution time of each test is stored in the OutSystems database, both the Original Feedback Time (time to execute all the tests in the original system) and the New Feedback Time (time to execute each selection of tests) were calculated using this historical information.

7 RESULTS

This section contains the experimental results obtained in this work. Specifically, the following research questions are studied:

1. How does MOTSD compare to random selections?
2. What is the performance of MOTSD for different combinations of objectives?
3. What is influence of innocent commits in the results?

The experiments presented hereafter were executed over the following 1-month periods at OutSystems: July 2018, October 2018, February 2019 and March 2019.

For the sake of readability, the results from each 1-month period were aggregated and averaged by adding the evaluation metrics values of each period and dividing by the number of periods.

Table 3 presents the results for all experiments regarding the number of commits, number of tool executions, Error Cases and Original Feedback Time.

<table>
<thead>
<tr>
<th># Commits</th>
<th># Tool Executions (%)</th>
<th># No .cs Files</th>
<th># No Cov. Data</th>
<th># New Files</th>
<th>Original Feedback Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1579</td>
<td>40%</td>
<td>439</td>
<td>398</td>
<td>111</td>
<td>101</td>
</tr>
</tbody>
</table>

Table 3: Number of commits, tool executions and error cases.

From Table 3, there is a significant amount of commits that are discarded due to error cases - only 40% of the commits are able to be used in our experiments.

The most frequent error cases are changes to only other file types (No .cs files) and lack of coverage data (No Cov. Data): given the OutSystems industrial context, this is expected since there are several changes to configuration or non-C# files, making it unreasonable to expect full test coverage over the system.

Baselines. To the best of our knowledge, none of techniques presented in section 3 are publicly available, which makes it difficult to evaluate MOTSD against existing work. Additionally, there is no set of benchmarks that fits the requirements of our work: large C# code base, high testing costs and extensive build history data.

Given these evaluation limitations, a decision was made to use random selections as baselines to compare with the results provided by MOTSD. This approach was chosen since it is not only easier to implement and understand the results, but it also provides a challenging target baseline for MOTSD to be compared with.

Random Selection. On the subject of random selections, there are two variations that need to be considered: either use a fixed random selection of tests (Fixed) or generate a new random selection for each commit (Dynamic).

In addition, since random selection does not require coverage data, none of the error cases described in the previous section will ever occur in a normal execution. Ergo, for the sake of fairness, random selection was only applied for commits where MOTSD would also be able to execute.

The results presented in Table 4 were obtained by running, for each 1-month period and each random selection variant, 10 iterations using 4 different solution sizes. For ease of understanding and analysis, the results are colored using a green-to-red gradient to highlight the better and worse values obtained: better results have a green color and worse results have a red color.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>% of Test Suite</th>
<th>Tool Found Failing Tests?</th>
<th>New Feedback Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Fixed (RF)</td>
<td>10%</td>
<td>25%</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td>15%</td>
<td>29%</td>
<td>12%</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>33%</td>
<td>15%</td>
</tr>
<tr>
<td></td>
<td>25%</td>
<td>40%</td>
<td>22%</td>
</tr>
<tr>
<td>Random Dynamic (RD)</td>
<td>10%</td>
<td>23%</td>
<td>8%</td>
</tr>
<tr>
<td></td>
<td>15%</td>
<td>30%</td>
<td>13%</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>34%</td>
<td>16%</td>
</tr>
<tr>
<td></td>
<td>25%</td>
<td>39%</td>
<td>21%</td>
</tr>
</tbody>
</table>

Table 4: Results for random selection experiments.

As would be expected, using a higher portion of the test suite leads to higher Recall values - random selections with 25% of the test suite provides the highest Recall values. Following a similar logic, the Feedback Time results also evolves as expected, i.e., a smaller random selection using 10% of the test suite provides the best Feedback Time values.

However, one should note that the Feedback Time presented is fixed for every commit regardless of the size of the code change. This is important because, from a developer point of view, if only a small number of files were changed then the returned selection should also be in the same order of magnitude.
Finally, the Computing Time values are not shown since computing a random selection takes less than 1 second. In addition, a shorthand notation will be used to refer to these results: e.g., RF_25 refers to Random Fixed with 25% of the test suite.

Objectives Combinations. Moving on to different combinations of objectives, the following questions are studied:

A. How is the performance of MOTSD impacted by different combinations of objectives?
B. Is there an optimal combination of objectives?
C. Does DDU improve the performance of MOTSD when used instead of coverage?

For the purpose of this work, only combinations of 2 were studied due to computational resources limitations and the exponential growth of possible combinations. Additionally, any combination of objectives had one and only one structural objective, in order to study the performance of DDU and normalized coverage separately.

The objectives combinations are encoded using the label letters presented in Table 2. For example, an objective combination where the first objective is maximize DDU and the second objective is minimize execution time, is encoded as DE.

Table 5 presents the results pertaining Recall and additional metrics (e.g., New Feedback Time) for combinations of 2 objectives. For comparison with baselines, the 2 best results from random selection are used (RF_25 and RD_25).

Looking at the detection of failing tests, MOTSD performs similarly to random selections: slightly better in terms of Macro-Recall (24% vs 22%) and Micro-Recall (31% vs 25%) and slightly worse in finding at least one of the failing tests (36% vs 40%).

Regarding Computing Time, an average of 6-12 seconds is good enough for integration purposes, although it is not negligible like in random selections.

Finally, despite having an average Solution Size much lower than random selections (1300 vs 2084), the average New Feedback Time is slightly higher by a few minutes. However, the distribution of these values (see Table 6) presents a high variation due to differently sized commits.

Table 6 describes the distribution of Solution Size and New Feedback Time for best combinations of 2 objectives.

Regarding the performance of MOTSD with DDU instead of Normalized Coverage, Table 5 only reveals a slight difference in terms of Micro-Recall where DDU-based configurations tend to be worse. This happens because Normalized Coverage does not take into consideration the test diversity: hence, the selections obtained are more likely to contain tests that have identical activity patterns and tend to fail at the same time. On the other hand, when optimizing for the DDU metric, selections with duplicate or similar tests are less likely to be returned as solutions.

Finally, the results presented indicate that (1) maximizing Test Fails is associated with higher Recall values and (2) minimizing Execution Time leads to lower Feedback Time values, regardless of the structural objective used. Hence, it is likely that a combination
of 3 objectives using Execution Time (Min) and Test Fails (Max) will provide the best results across both evaluation metrics.

Innocent Commits. At OutSystems, developers commit changes to the same branch of code over which the CI pipeline executes test suites to check the quality of the changes. Since these test executions occur sequentially for each commit that arrives, it may happen that a commit reports failing tests that were already failing due to a previous commit.

In this case, this commit should be tagged as an innocent commit for the purpose of evaluating MOTSD fairly since the code change was not related with the tests that failed. In order to identify which of the red commits (i.e., reporting failing tests) are innocent, there are 3 strategies that could be used:

1. **Previous Red**: if the previous commit was red, then the following commits are innocent.
2. **Equal**: if the previous commit had the same set of failing tests, then the current commit is innocent.
3. **Superset**: if the previous commit’s set of failing tests is a superset of the current commit’s, then the current commit is innocent.

Table 7 illustrates each of these strategies in the context of a sequence of revisions and associated sets of failing tests. For each strategy, each revision is classified as innocent (Yes) or not (No). In addition, correct and incorrect classifications are marked with green and red colors, respectively.

![Table 7: Example of the different innocent commit classification strategies.](image)

Evidently, the superset strategy is the only one that correctly identifies all the innocent commits in Table 7 example (which is representative of the most common cases). Ergo, the previous experiments were repeated, while discarding commits that were classified as innocent when calculating Recall values.

This change meant that the number of valid red commits considered was much lower than in normal mode used in the previous section. Table 8 shows that more than 50% of the red commits are discarded when applying this innocent commit filter.

![Table 8: Number of valid red commits when filtering innocent commits or not.](image)

Table 9 presents the results obtained for random selections and combinations of 2 objectives using this innocent commit filter.

Despite the significantly lower amount of commits, the performance of random selection did not change much: the only metric that improved was finding at least one of the failing tests which increased by 8%. Every other metric (Recall and Feedback Time) remained roughly the same.

Regarding the combinations of 2 objectives, in terms of ability to detect failing tests, the performance of MOTSD increased on all metrics: (1) finding at least one: +10%; (2) Macro-Recall: +5%; (3) Micro-Recall: +10%.

The remaining metrics remained the same which indicates that reducing the number of commits did not impact the distribution of the commits considered for evaluation.

In addition, it is worth noting that the best combinations from previous sections (FD, FN and NF) were now able to perform much better than random selections in terms of Recall: +7% Macro-Recall and +13% Micro-Recall.

This significant improvement, by filtering out innocent red commits, highlights the advantage of using a structural-based selection from MOTSD instead of random selections.

8 FUTURE WORK

**Alternative Data Source.** This work revealed concerns regarding applicability to cross-language code changes and the scalability bottleneck of coverage data. One possible improvement would be to use some other source of data instead of code coverage.

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\(^9\). This graph could be further enriched with build history data by linking changed files with tests that failed in the respective commit.

**User Study and Diagnostic Cost.** MOTSD was only evaluated over past commits at OutSystems to study its ability to detect failing tests and estimate how much better the feedback time would be.

However, this evaluation study did not look at the developer experience when using such a tool, which is a critical part in increasing its adoption and effective use.

Hence, it would have been interesting to do a user study where MOTSD would be integrated in a production environment with real developers. In this context, it would be possible to gather both user feedback to measure tool adoption by the developers and calculate diagnostic cost improvements.

**Simulate 0-100% Recall.** Looking forward to future developments, it would be interesting to know how much more can the Feedback Time be reduced by increasing the Recall. In other words, assuming a perfect test selection with 100% Recall, what would be the feedback time in this case?

Using historical data, this could be simulated easily by querying the database and extracting the set of failing tests for each commit. However, what should be the actual size of the selection, i.e., how high should the Precision be? In a utopic scenario, the perfect selection tool would have both 100% Recall and 100% Precision wherein only the failing tests would be selected.

**Machine Learning Approach.** This work revealed the multiple challenges of applying existing coverage-based techniques in an industry context where:

\(^9\)For example, this could be done using hercules https://github.com/src-d/hercules
Obtaining test coverage is expensive and difficult which raises performance concerns.

- Code bases tend to be multi-language, eventually leading to missing data when selecting tests.
- Existing test coverage is not perfect and does not cover all execution paths as is assumed in the literature.
- The test selection tool needs to be resilient to new data due to the high frequency of code changes.

One possible alternative to the current approach would be to use machine learning since, in industry, there is a wealth of historical data which can be exploited to predict which tests will fail for a given commit.

Concretely, this would be done using a classification model that, given a commit and a test, predicts if the test will fail or not. These predictions could then be used to build the selection of tests that should be executed for a set of code changes.

This type of approach has been implemented recently in industry contexts much larger than OutSystems, which further motivates its potential application. For example, Facebook [3] has successfully built a machine learning model using three types of features:

1. **Change-Level**: change frequency of modified files, number of changed files, number of distinct authors.
2. **Test-Level**: test failure rate, associated project name.
3. **Cross-Features**: distance in build dependency graph, lexical distance between file paths.

Another example of applying machine learning to this problem comes from Microsoft’s FastLane [7] system, albeit it is not only a test selection solution. FastLane tackles the same problem but applies multiple different techniques in an end-to-end solution:

1. First, a commit risk prediction model is used to predict if any tests need to be ran for a given commit;
2. Then, FastLane measures test correlations based on previous execution outcomes to know if a test needs to be ran;
3. Finally, while the test is running, a second prediction model is used to stop the tests based on current runtime metrics (e.g., if a test is taking too long to complete).

### Test Selection Benchmark

Finally, to the best of the author’s knowledge, there are no benchmarks on which to evaluate any test selection tool.

While it is clear that developing such a benchmark has several technical challenges (e.g., multiple languages, CI tooling and version control systems), the lack of a benchmark makes it difficult to understand how MOTSD compares with existing approaches.

The test selection benchmark should be available as open science to support easier reproducibility and replication of test selection approaches. In addition, it should support two main features: (1) the ability to extract a test selection dataset from a GitHub project; (2) automate the evaluation process of a new test selection approach for a given dataset.

### 9 CONCLUSIONS

This paper presents a multi-objective test selection approach developed for the industrial context of OutSystems. The developed tool, MOTSD, 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.
REFERENCES

A Constrained Particle Swarm Optimization Approach for Test Case Selection.
In SEKE.

the Particle Swarm Algorithm. In 1997 IEEE International conference on systems,
man, and cybernetics. Computational cybernetics and simulation, Vol. 5. IEEE,
4104–4108.

Predictive Test Selection. In Proceedings of the 41st International Conference on
Software Engineering: Software Engineering in Practice (ICSE-SEIP ’19). IEEE Press,
Piscataway, NJ, USA, 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018

[4] Christopher Manning, Prabhatbhar Ramhavan, and Hinrich Schütze. 2010. Intro-
duction to information retrieval. Natural Language Engineering 16, 1 (2010), 100–103.

[5] Atif Memon, Zehao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siem-
borski, and John Mecco. 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

ability 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] Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Mad-
dila, and Nachiappan Nagappan. 2019. FastLane: Test Minimization for Rapidly
Deployed Large-scale Online Services. In Proceedings of the 41st International
Conference on Software Engineering (ICSE ’19). IEEE Press, Piscataway, NJ, USA,
408–418. https://doi.org/10.1109/ICSE.2019.00054

https://doi.org/10.1109/32.536955

[9] Luciano S. de Souza, Pericles B. C. de Miranda, Ricardo B. C. Prudencio, and
Flavia de A. Barros. 2011. A Multi-objective Particle Swarm Optimization for
Test Case Selection Based on Functional Requirements Coverage and Execution
Artificial Intelligence (ICTAI ’11). IEEE Computer Society, Washington, DC, USA,
245–252. https://doi.org/10.1109/ICTAI.2011.45

selection. In Proceedings of the 2007 international symposium on Software testing
and analysis. ACM, 140–150.

https://doi.org/10.1002/stv.439

[12] 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