# Efficient Regression Testing Based on Test History: An Industrial Evaluation

Ekelund, Edward Dunn; Engström, Emelie

*Published in:*
2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)

*DOI:*
10.1109/ICSM.2015.7332496

Published: 2015-01-01

# Efficient Regression Testing Based on Test History: An Industrial Evaluation

Edward Dunn Ekelund*, Emelie Engström†
* Axis Communications AB, Sweden
edward.ekelund@axis.com
† Dept. of Computer Science, Lund University, Sweden
emelie.engstrom@cs.lth.se

***Abstract*—Due to changes in the development practices at Axis Communications, towards continuous integration, faster regression testing feedback is needed. The current automated regression test suite takes approximately seven hours to run which prevents developers from integrating code changes several times a day as preferred. Therefore we want to implement a highly selective yet accurate regression testing strategy. Traditional code coverage based techniques are not applicable due to the size and complexity of the software under test. Instead we decided to select tests based on regression test history. We developed a tool, the Difference Engine, which parses and analyzes results from previous test runs and outputs regression test recommendations. The Difference Engine correlates code and test cases at package level and recommends test cases that are strongly correlated to recently changed packages. We evaluated the technique with respect to correctness, precision, recall and efficiency. Our results are promising. On average the tool manages to identify 80% of the relevant tests while recommending only 4% of the test cases in the full regression test suite.**

***Index Terms*—regression testing, industrial evaluation, continuous integration**

## I. Introduction

Regression testing is widely used in industry to ensure that introducing modifications to the software does not affect system functionality. A common approach is to have a dedicated regression test suite which is repeatedly run in its entirety [1]. This strategy may be easy to implement but is often unnecessarily expensive, especially if changes affect only a small part of the whole system. As software development practices changes towards shorter iterations and continuous integration, regression tests are run more frequently which in turn increase the demand for selective and minimized regression testing.

A vast amount of research has been spent on regression test selection (RTS), i.e. the design and evaluation of strategies which focus testing only on changed parts of the system [2]. However, only few empirical evaluations of RTS techniques are carried out in a real industrial context [3]. Many proposed techniques are very meticulous in finding exactly which function affects which test, and must therefore perform a deep and often time consuming static analysis to map changes to tests. At a company with a large code base such fine-grained correlations between tests and functions in the code are not necessarily needed, or even possible to obtain. The software under test, SUT, may be too large and complex, involving interactions between several different components. Such interactions are difficult to quantify by static analysis alone. Furthermore, the value of a static analysis is short-lived in an active, ever changing code base.

In this paper we report our experiences of implementing and evaluating a tool that enables continuous analysis of the history of changed packages and their test outcomes, to support regression test selection on code package level. The research was conducted at Axis Communications which has a big regression test suite run daily. The full regression test suite takes approximately seven hours to run which delays the integration of new code to the platform severely. It is a huge waste of time and makes development difficult and costly. In our case correlations at package level is sufficient for the task of giving us fast regression feed-back several times a day. With a cheaper analysis we expect it to be performed more often and thus better keep up with the transformations of an active and volatile codebase.

The remainder of the paper is structured as follows: Section II and III presents related work and the background of the study. Section IV and V outlines the evaluation framework and explains the process followed in this research. Section VI and VII describe the Difference Engine and outlines our results. In Section VIII and IX we discuss our results and future work. Section X concludes the paper.

## II. Related Work

Much of the early RTS research is based on employing static code analysis on a functional level to perform the test case selection [4], [5], [6]. Since such analysis would be very costly in our context[1] – with the probable effect of not being updated often enough to keep compatible with the changing SUT – we decided using historical test data for test selection would be a more effective approach.

Some studies, by e.g. Kim and Porter [7] show improvements on the problem of test selection from costly static analysis by including a historical context as well as taking into consideration the fact that the tests need to be continuously re-evaluated and re-prioritized. They achieved lower predicted costs without reducing effectiveness by running only carefully selected subsets of the test suite. However, in their work, Kim

[1]The code base contains more than 35 MLOC.

and Porter only analyze a couple of smaller programs on a function-based level involving only a relatively low amount of tests, which is not generalizeable to our large test suite and need for streamlining the regression testing process.

Wikstrand et al. studied the usage of a fixed-file cache for RTS [8], [9]. The idea is to correlate tests to source code based on information from closed error reports. A drawback of this approach is the dependability on human based input. Furthermore it requires a sufficient existing infrastructure to aggregate this data from the change management process with data from the testing process, which is not in place at Axis. We do however explore the idea of keeping a cache of correlations but we use data from one database with automatically generated test data.

Rees et al. propose employing Bayesian graphical models (BGMs) to optimize the failure probabilities for a given number of tests [10]. Although it is an interesting approach it was not fully automated and results were inconclusive regarding the benefit of creating BMGs in all contexts. Thus we chose not to add this extra intelligence to our algorithm at this stage.

Huang et al. proposed and evaluated cost cognizant history based test case prioritization [11]. They employ a genetic algorithm to analyze historical data on regression test runs and assign priorities to test cases. Also this algorithm adds intelligence to the test selection compared to the one implemented in this case study. We decided to keep the algorithm as simple as possible to make introspection of results, maintenance and possible future extension of the algorithm as easy as possible.

## III. Background

The main research question for this study was:

RQ: How to shorten lead times for regression testing to enable fast feedback in a continuous integration context?

To explore this we iteratively built a tool for analyzing the history of regression tests and correlate changed code with failing tests on package level. Tests are then assigned risk levels or weights which may be presented for a test coordinator to assess manually. Alternatively, the regression test system could be configured to automatically select tests with a risk level above some specified threshold.

The SUT in our case contains several hundred packages encompassing several millions of lines of code. The test history at Axis resides in a vast MySQL database, called DBDiffer. In this database various regression test data is kept, and among this data one can extract which packages were changed before the regression test suite was run, and which tests failed and succeeded. DBDiffer contains everything we need in order to find the correlation between code packages and tests, but the data exists in several tables which are loosely linked to each other through MySQL-id:s.

## IV. Metrics

We evaluated the RTS strategies in accordance with the framework by Rothermel and Harrold [12]. The goal of the test selection, apart from saving time, is to be sure that the tests that are run are relevant tests. A relevant test is in our case a test that will flip[2] due to the changed code. The trivial method of finding all possible flips, i.e. achieving 100% recall[3], is to run all tests, but then precision would be low. Instead, precision would be high if the only tests we suggest were fault revealing tests. Both precision and recall are important for test selection, and thus we calculate the F-measure based on the ratio of the two as a singular measurement. In other words, the F-measure is a way to compress two dimensions of selection quality, recall and precision, into one dimension, to make direct comparisons between different selection strategies easier.

*Precision* is defined as the fraction of selected tests that are relevant to a particular build. The precision score ranges between 0.0 and 1.0. A precision of 1.0 means that 100% of the selected tests are relevant tests.

$$precision = \frac{|\{relevant\ tests\} \cap \{selected\ tests\}|}{|\{selected\ tests\}|} \quad (1)$$

*recall* is defined as the fraction of relevant tests that are included in the presented selected tests. Also the recall score ranges between 0.0 and 1.0. A recall of 1.0 means that 100% of the relevant tests are included in the set of selected tests.

$$recall = \frac{|\{relevant\ tests\} \cap \{selected\ tests\}|}{|\{relevant\ tests\}|} \quad (2)$$

*Relevant* is defined in this context as a test that has flipped.

*Selected* is defined in this context as a test suggested by the Difference Engine.

*The F-measure* of precision and recall is defined as

$$F - measure = 2 * \frac{precision * recall}{precision + recall} \quad (3)$$

Just like with precision and recall, the F-measure lies in the range [0.0, 1.0]. 1.0 is a perfect score, meaning that both precision and recall are perfect as well.

*Efficiency* is approximately assessed by calculating the average execution time for a test case, multiply it by number of selected tests and comparing with running the full regression test suite.

## V. Research Procedure

For the sake of this study we developed three software modules, see Figure 1, the Difference Engine, providing the actual analysis of the correlations, and two auxiliaries: Seeker, which extracts the historical test data, and Simulatron, which generates simulated historical test data. Simulatron is used to ascertain that the analysis performed by the Difference Engine is plausible. All three programs were developed in parallel, and iteratively and the results of improving Seeker or Simulatron was used to improve the analysis of the Difference Engine. During development we evaluated the tool with respect to correctness, recall, precision and efficiency.

[2]A flipped test case is a test case that changes its verdict between two consecutive test runs.

[3]Rothermel and Harrold use the term inclusiveness instead of recall.

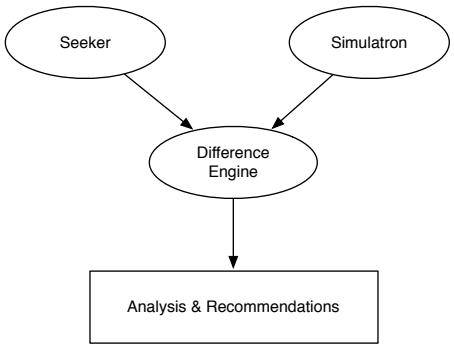


Fig. 1. Graph describing the program dependency chain

### A. Check for Correctness

A "correct" correlation in this context is a correlation between a code package and a test that we know must be true (i.e. if executed the test case would execute the code in the code package). This is basically what other RTS strategies achieve through static analysis of code and tests.

Since this information is not available, and would be prohibitively costly to extract through static analysis, in our context we assessed the correctness of the suggested correlations with the simulated data produced by Simulatron. With data generated by Simulatron we know whether the correlations found are valid since it is immediately apparent from the names of the code packages and test names.

We evaluated the test selection algorithm's ability to find correct correlations by determining whether the most highly weighted correlations were true correlations (i.e. that the correlations are not due to package changes and test flips that coincide by chance). We parsed the outcome of an analysis on simulated data and assigned and calculated the percentage of true correlations.

The following Simulatron parameters were used:

- 75% package noise and 30% test noise, for a variable number of builds. – This means that in 75% of the builds a random package will be changed, but its corresponding test wont flip, and in 30% of all builds a test will flip but its corresponding package will not be changed.
- 99% package noise and 99% test noise, for a variable number of builds.
- 500, 1000 and 10 000 builds.
- A pool of 100 packages to change.
- A pool of 100 tests to flip.

75% package noise and 30% test noise were chosen because a cursory inspection and analysis of the historical test data show that these numbers resembles the real data, approximately. Simulated data with 99% package noise and 99% test noise is more noisy than the real data, but is included to see how the Difference Engine performs with data that is noisier than the real data.

It should be noted that Simulatron generates much more dense data than the true historical data extracted with Seeker. Many of the builds that Seeker extracts are blank builds, meaning that either no packages have been changed or no tests have flipped since the preceding build. With data generated by Simulatron, however, all builds are guaranteed to have at least one package changed, and one test flipped.

### B. Optimizing with respect to Number of Analyzed Builds

With an increasing number of Builds, we can hypothesize that both precision and recall must increase. Thus we measured precision and recall for different sizes of build sets. We used Simulatron to create build sets of various sizes; 10, 100, 500, 1k, 2.5k and 10k builds. We then calculated the F-measure between precision and recall for the different sizes of build sets.

The Simulatron parameters will be the following:

- 75% package noise, meaning that 75% of the builds will contain changed packages with no corresponding test flips.
- 30% test noise, meaning that 30% of the builds will contain flipped tests with no corresponding package changes.
- A pool of 100 code packages to change.
- A pool of 100 test packages to flip.
- 10, 100, 500, 1000, 2500 and 10 000 builds.

### C. Evaluation with Real Data

Even if the Difference Engine performs well with simulated data in the two prior evaluation steps, the question is if these suggestions can be used to accurately predict the flips that actually occur on real data. We evaluated this using the historical builds, extracted by Seeker. To avoid the bias of overfitting our prediction model we split the historical data into a training set and a validation set. We split the data into 10 different parts, where 9 parts were used for training and 1 part for validation. Each of the ten parts were used as validation once, while the remaining comprise the training set. For each of the 10 validation sets we calculated recall and precision.

We compared three RTS strategies:

- **Retest all** – No test selection is performed; run all tests.
- **Wide selection** – Select all tests that have ever flipped, regardless of which packages have been changed. This can be useful if the narrow selection method returns very few suggested tests, or none.
- **Narrow selection** – Select tests based on which packages have been changed for a particular build. Some packages will have no correlated tests, which is when it might be useful to fall back to the wide test selection strategy.

## VI. Difference Engine, Seeker and Simulatron

All code was written in Python, which is a flexible and clear language with many readily available machine learning libraries and diagram generating utilities[4]. The code was written with the functional programming paradigm in mind, in order to avoid producing side effects and make unit testing easier.

[4]The library sklearn for example, provided a module for running the K-fold cross validation technique used for evaluation, and the library matplotlib has been used to generate the diagrams in this paper.

The core code modules have nearly 100% unit test coverage, and should therefore be easy to modify and improve should the need arise. Some object oriented features are utilized as well in order to gather functions into logical units and enable composability and enhance usability.

### A. Difference Engine

The Difference Engine takes a set of builds[5] as input, and gives a weighted correlation between code packages and test cases as output. Each build is extracted from the data generated by either Seeker or Simulatron.

Since the raw historical data in the database does not contain information of changed packages or flipped[6] test cases this must be calculated on the fly when performing the analysis by comparing consecutive builds. Correlations are calculated according to the following:

1: Compare each build with its previous build to identify changed packages and flipped test cases.
2: For each flipped test case in a build increase its correlation coefficient to each of the changed packages of that build.

Thus the strength of the correlation between a test and a code package depend on how many times they are found to co-change with respect to two consecutive builds.

The underlying algorithm used to find the correlations between code packages and tests can be expressed more succinctly in pseudo code:

```
For each <build_result> in <set>:
      For each <code_package> in <build_result>:
          If <code_package> is changed:
              For each <test> in <build_result>:
                  If <test> flipped:
                      correlation_coefficient += 1
```

### B. Seeker

The purpose of Seeker is to extract the relevant records from the database of historical regression test data, and organize them into a format parsable by the Difference Engine. Builds only exist conceptually. In order to find which modules where changed in a build, and which test cases flipped, packages and tests first have to be linked to the build in question. When parsing the database, Seeker needs to perform two actions:

1: Extract all code packages and their revisions and map them to their unique MySQL build identifier.
2: Extract the statuses and names from all test cases and map them to their unique MySQL build identifier.

The Seeker then joins all code packages and test cases based on their unique identifiers to create the builds. All subsequent builds are then added to an ordered list to form a build set, which is the intermediate data format passed on to the Difference Engine for analysis.

[5]In this paper we refer to a build as a set of packages and their revisions together with the corresponding regression tests and their verdicts

[6]A flipped test case is a test case that changes its verdict between two test runs

### C. Simulatron

Since the database containing the historical test results is gargantuan, around 100 gigabytes, and the code base contains several millions of lines of code, it is impossible to intuitively determine whether the correlations found by the Difference Engine are correct and not due to random noise or to errors in the correlation algorithm.

Simulatron was developed to enable assessment of the result produced by the Difference Engine. It creates a simulated regression test history where packages and tests are named so that it is immediately apparent whether they should be correlated or not. In their survey [2] Yoo and Harman argue that a realistic simulation may provide an efficient manner in which to assess RTS selection strategies.

Simulatron is used to generate:

- a variable number of code packages
- tests for each package
- associated test flips when changing a package
- noise, i.e. flipped tests with no corresponding package change, or package changes with no corresponding test flip
- a simulated regression test history containing the packages, tests and noise specified above.

Noise in this context is Simulatron adding random changes to packages, or random flips to tests, in order to simulate the noise present in the real historical data extracted from the database. Noise in the real data is not entirely random however, and may be due to a variety of reasons. For example, with real historical data, 2 packages are, in average, changed per build. This means that any potential test flips that occur will correlate both code packages to the tests, even if only changes in one package is responsible for the flip. In practice, this means that one of the code packages will be falsely correlated to the test, and we have "noise". Another possible source of noise is when something outside of the control of the regression test suite, for example a network error, causes tests to flip.

A great advantage of having Simulatron is that if the correlation algorithm used by the Difference Engine proves to be inaccurate or inadequate, a modified or entirely different algorithm can be quickly tested and evaluated easily with simulated data from Simulatron. Another advantage is that simulated test history can easily be generated by Simulatron to test different variations in input data, to see how such variations affect the performance of the selection algorithm.

## VII. Results and Analysis

### A. Noise

Figure 2 illustrate how the correlation weights indicate that a test outcome is dependent on changes to a code package. The example shows the results produced when running the Difference Engine analysis on simulated data generated by Simulatron. Here we see that `package3_test` has a correlation weight of 19 to package3. This means that a change to package3 that constitutes a new bug or a bug fix correlates very strongly to a change (flip) in the test outcome for this

*Input:*

```
$ ~/difference\_engine regression\_test\_data.db
```

*Output:*

... Analyzing data

*Results:*

```
code/package3.py
    package3_test: 19
    package2_test: 2
code/package94.py
    package94_test: 8
    package3_test:  3
    package17_test: 1
```

Fig. 2. Difference Engine example run on simulated data

test. Since we are using simulated data we can tell from the name that this test is in fact dependent on package3 and that this strong correlation is quite correct. If this analysis had been performed on real data, it would then be prudent to run package3_test every time a package3 is changed.

The correlation weight between package94 and the test for package3 is interesting, since it means that three times when this test flipped package94 was changed as well. The correlation is totally random however, and is just due to the noise that Simulatron adds when generating simulated regression test data. In reality package94 just happened to be changed three times at the same time as a bug was either fixed or added to package3. There are a lot of packages with correlation weights of 1, but most have been removed from the heavily pruned example output above. A weight of 1 is too low to be able to distinguish true correlations from random noise. That is, if two packages are changed and only one test flips, both packages will have their correlation weight increased by one, yet only one of the changes caused the flip. It isn't until we start to see a trend, that is, weights of 2 or more, that we can be more sure of the correlations between the code packages and test packages.

### B. Correctness

The results shown in Table I show that the correlation algorithm is very robust. Even when there is both package and test noise in 99% of the builds, the correct correlation is still weighted the highest in almost all cases. The more builds that are analyzed, the better the algorithm will perform in mapping the correct test to their corresponding package.

### C. Number of Builds

Early prototyping work during the iterative development of the code showed that one weakness of the Difference Engine is that it does not work well if the BuildSet it analyzes contains too few Builds, which is not very surprising. The minimal number of Builds necessary to provide a reasonable analysis is highly dependent on the input data, and which packages you

TABLE I
SHARE OF CORRECT CORRELATIONS IN SIMULATED DATA

| Nbr builds | Pkg noise % | Test noise % | correct correlations % |
|---|---|---|---|
| 500 | 75 | 30 | 96 |
| 500 | 99 | 99 | 88 |
| 1000 | 75 | 30 | 99 |
| 10k | 75 | 30 | 100 |
| 10k | 99 | 99 | 100 |

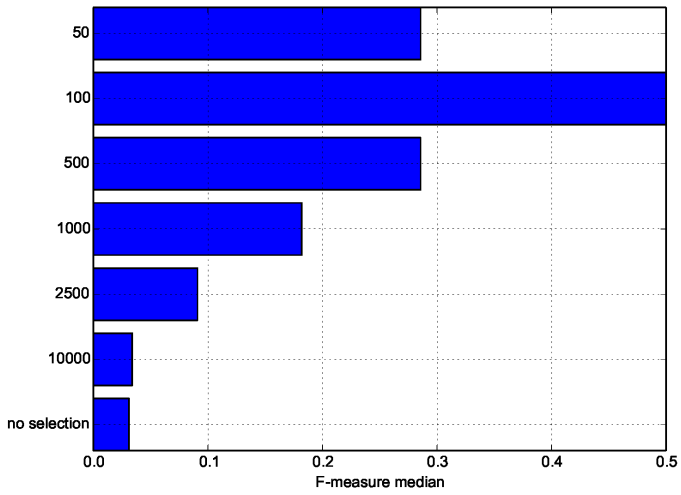


Fig. 3. F-measure for 50, 100, 500, 1k, 2.5k and 10k builds, using the narrow test selection strategy. The F-measure for no test selection is also shown, for reference.

are interested in finding test correlations for. For example, if you look for correlations for one specific package, you may be very lucky and the package will have been changed often in many recent builds, and therefore have many relevant and/or high correlations with tests, or very unlucky and the package will never have changed in the BuildSet you analyzed, and will therefore have no correlations. 10 builds are generally not enough to produce any meaningful test selection analysis. Most often there is not enough data to perform a narrow test selection.

In the diagram in Figure 3 the F-measure appears to reach a maximum at around 100 builds, before deteriorating. The Difference Engine did not perform as could be expected (i.e. that more builds will lead to better F-measure values), which in hindsight should be quite obvious; given enough builds, the accumulated noise quickly overwhelms the true correlation data.

The narrow selection continuously outperforms running all tests, until the amount of noise is too overwhelming. At that stage, at 10k analyzed builds, it performs as bad as running all tests, since it will basically recommend running all tests. This owes a lot to the fact that at the default settings, the Difference Engine performs no filtering at all, and even correlations of weight 1 are included in the selection of recommended tests.

A 1-weight correlation is just as likely to be random noise as a true correlation. Thus, as the number of builds increase, more noise is added to the set of recommended tests, until finally, there is virtually no difference between the set of selected tests and the set of all tests.

The solution to this problem is to introduce an intelligent way of culling the noise. Currently the Difference Engine supports a primitive filter for discarding all test recommendations beneath a specified threshold weight. Setting this threshold at 1, will keep the narrow test selection superior to running all tests longer, but it will eventually be overwhelmed as well.

Consequently, a future improvement to the Difference Engine would be to introduce a form of memory to the analysis. As more builds are added to the analyzed pool, the weights should be decreased for tests that have not flipped for some amount of time. The history based prioritization model proposed by Kim and Porter [7] could be applied for this purpose.

Finally, observing Figure 3, counterintuitively it seems that after a 100 builds, more analyzed builds give worse results. The reason is that the recall is perfect already at 500 builds, and cannot go any higher after that, while the precision continues dropping as more noise is added to the results of the analyzed builds. All in all, the results of this part of the evaluation provide compelling reasons for introducing a strategy for handling the accumulation of test noise.

### D. Effectiveness on Real Data

Figure 4 displays the recall of the different test selection strategies. As expected the average recall for running all tests is the highest, since running all tests guarantees that you will not inadvertently miss out on any relevant tests. The wide test selection places second, and the narrow test selection places last, also according to expectations, since they run fewer tests in turn. While the narrow selection might appear to have 20% worse recall than selecting all tests, this number is somewhat pessimistically skewed as the (arithmetic) mean is sensitive to extreme outliers. It is very likely that sometimes the narrow selection has a recall of 0 which will drag the average recall down a lot.

Figure 5 displays the precision of the different test selection strategies. The more specific the test selection strategy, the greater the precision. While the absolute values of the precision appear very low, the relative ratio between retest all and test selection is the important factor. Greater precision means less time wasted on running irrelevant tests, which was the goal of performing this study.

Also, it is important to keep in mind that the test selection is only a recommendation based on historical data. Just because a (correct) correlation has been found between a code package and a test, there is no guarantee that said test will flip just because that package is changed. Given the plausible assumption that the codebase is relatively stable, i.e. that new code most of the time does not introduce faults, most tests that are run will not flip. If a test is recommended, but it does not flip, its precision will inevitably suffer, no matter how accurate the RTS selection technique is. In relative terms the narrow test selection strategy significantly outperforms the retest all strategy, by about a factor of 14.

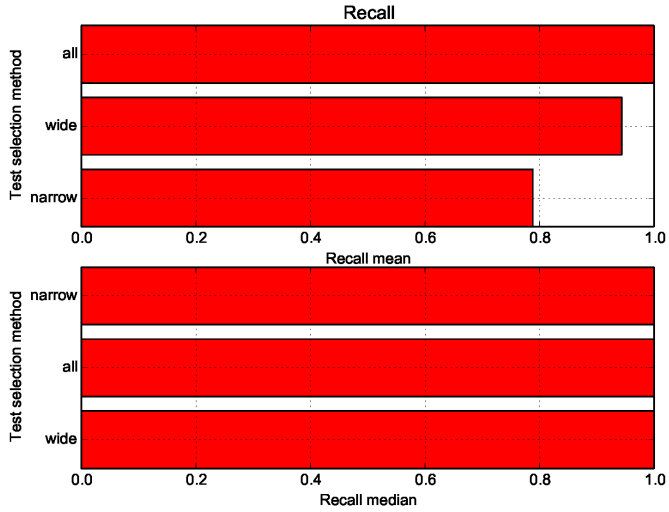


Fig. 4. Recall for narrow, wide and retest all strategy

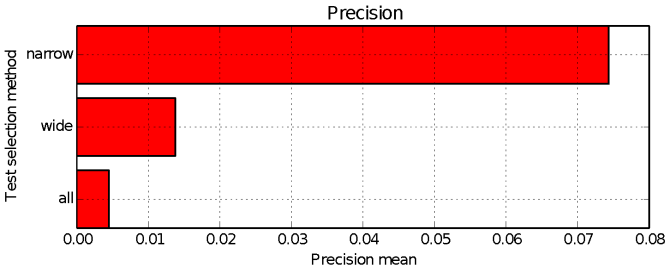


Fig. 5. Precision for narrow, wide and retest all strategy

### E. Efficiency

We use a simple cost model: If $analysis\_time + subset\_test\_run\_time < full\_test\_run\_time$ time is saved. Furthermore we consider all tests equal in duration. This is obviously not true, and will have to be revised at a later date in order to achieve a more correct analysis, but the work involved is considerable, and outside the scope of this study.

Extracting the data from the database with Seeker takes slightly less than $5$ $minutes$. Analyzing the data with the Difference Engine takes less than $15$ $seconds$. So we can say that $analysis\_time$ is approximately $5$ $minutes$. The $subset\_test\_run\_time$, or, the duration of running the selected tests, is calculated as the number of tests suggested by a selection model, multiplied by the average duration time of one test. All tests make up $1265$ $units$ and take about $7$ $hours$ to complete. The average duration of one test is therefore calculated to $760/1265$ which is $0.33$ $minutes$.

The time savings are presented in Table II. The duration is calculated by multiplying the number of tests with the average duration of one test ($0.33$ $minutes$). The total is calculated by

adding the time of analysis (5 $minutes$, worst case scenario) to the duration.

A lot of time is saved by running only the tests recommended by the Difference Engine. With the narrow test selection strategy, it is possible to perform 20 integrations in the same time frame as it takes to perform one integration running all tests. At that rate, it is doubtful whether the regression test runtime is a bottleneck any longer; more likely, the bottleneck will be other manual processes that are involved in an integration of new code into the platform.

Finally, an interesting observation due to how the wide RTS strategy is implemented, i.e. recommending all tests that have ever flipped, regardless of which package has been changed, we see in table II that over 60% of all the tests in the full regression test suite have never flipped. Therefore, running the full test suite in every integration regardless of how small the change is, is generally a huge, huge waste of time. Just culling the tests that have never shown any relevant results would result in reducing the time spent on regression testing by 60%. A full regression test run could still be performed nightly as a safety check.

### F. Validity

While the data produced by Simulatron is designed to be similar to the real data extracted by Seeker, there are still some differences. Simulatron generates much more dense data than Seeker, meaning that changes (i.e. changed code and flipped tests) are guaranteed to happen for every build. This is not the case with real historical regression test data where large portions of the data have either no changed packages or flipped tests and therefore cannot be used for analysis. However, the denseness of the simulated data is a conscious choice since it enables evaluation of long term trends not immediately apparent when analyzing real historical data. This also means that there is a risk of missing some vital characteristic that will only be visible when real data is used.

Due to the denser data some trends are significantly accelerated when analyzing the simulated data compared to when analyzing real historical data. This should be considered given the results presented, where accumulated noise will quickly deteriorate the F-measure as the number of builds increase. It will not happen as quickly when running the analysis on real data.

There are two apparent problems with using the F-measure to verify the efficacy of the test selection strategy proposed in this paper. The first is related to how precision is defined; in essence, precision is a value to measure whether a recommended test has flipped. A code package may be very strongly correlated to a test, but there is no guarantee that the test will flip just because that particular code package has been changed. In most cases the test will not flip, since most changes to the code do not cause any faults, and this means that the value of the precision will suffer greatly. The second problem is related to recall. With any form of non-safe test selection strategy there is always the risk that a relevant test will be missing from the pool of recommended tests. This means we cannot solely rely on the tests selected by the Difference Engine, and we will still have test runs that take the full 7 hours to complete.

A less apparent problem with the F-measure as an evaluation tool is that for most intents, great recall and decent precision are more beneficial than decent recall and great precision, yet it is not taken into account when calculating the F-measure.

## VIII. Discussion

RTS techniques may be beneficial if implemented properly. Our approach differ from much of the previous research on RTS in that we do not analyze the code but create correlation between tests and code solely based on test history. This makes our analysis less costly. Performing static analysis would not be practical in our context since the SUT are too large and complex. The algorithm implemented by The Difference Engine is very simple, but works fine and is fast even for large and complex systems. It is, on the other hand, highly dependent on having a regression test history to analyze and draw conclusions from. To provide a sufficient analysis we need a lot of historical data which cover all code packages and tests, and the data needs to contain enough failing tests to provide sufficient correlations. Ironically, the test selection algorithm thrives on an unstable code base that regularly introduces faults for as many different packages as possible, which is usually something you do not want when running regression tests.

Running only the recommended tests will starve the Difference Engine of historical data to analyze. Thus it is imperative that the full regression test suite is run periodically, for instance in regular nightly builds. Given this strategy, a test will only be missing from the pool of recommended tests for one day, then the full regression test run will pick it up during the night and it will be included in the set of recommended tests the next day. It is however important to remember that the (slight) risk of missing relevant tests will still be there for a full day, which has to be taken into account when verifying firmware using only the tests recommended by the Difference Engine.

In previous studies [3], [13] random test selection has been shown to be an effective RTS strategy. In our context selecting random tests would be an abysmal approach in comparison to the strategies enabled with the Difference Engine. Since the average share of relevant tests per test run is so small $(2/1265)$, a randomized choice would give poor values for the F-measure. If selection is narrow recall would be very low and if selection is widened to improve recall precision would suffer too much. A randomized RTS strategy also has the disadvantage that two consecutive test runs are likely to look entirely different. For various reasons engineers at Axis often want to compare the results of the newest regression test run with older versions. When the selection of tests is totally random such a comparison will not be possible since the selection of tests is likely to differ too much between different versions. The only benefit of random selection is that it is cheap computation-wise, as all it needs to do is randomly draw some tests from the test pool. However, the analysis performed

TABLE II
TIME SAVING CALCULATIONS

| Task | #Tests | Duration (min) | Total (min) | Time saved (min) | Time saved (%) |
|---|---|---|---|---|---|
| Retest all | 1265 | 420 | 420 | 0 | 0 |
| Wide selection | 470 | 152 | 157 | 263 | 63 |
| Narrow selection | 50 | 16 | 21 | 399 | 95 |

by the Difference Engine is also very fast; it finishes in mere seconds.

## IX. FUTURE WORK

While the results of this research are useful already, there are of course a lot of improvements that can be added at a later date. Although the software is already performant enough, it can be enhanced. Python has an abundant selection of libraries that are specifically tuned for the kind of computations used to analyze the historical test data. A lot of bespoke code can be replaced by existing, more efficient libraries. For example, there is a library called Pandas, featuring data structures optimized for the kind of data processing required by the Difference Engine to successfully analyze the raw data it has to work with. Currently the most time consuming process is the extraction of data from the MySQL database. This could be optimized further.

A lot can be learned by adding more metadata to the final analysis. For example, which code package causes most tests to flip? Which package causes the least amount of tests to flip? Which package is changed the most, and which is changed the least? Which tests fail or flip most often? What is the average amount of packages that change per build, and what is the average amount of tests that flip?

Given more powerful and flexible data structures than the ones currently implemented in the Difference Engine, these questions can be answered quite easily. Said answers can then be used to gain a better understanding of how the regression test suite performs, provide ideas to attain better performance from the test suite, and perhaps even help explain why some previously inexplicable things occur during regression test runs.

Some tests in the regression test suite have proven to be quite unstable and adding support for blacklisting such troublesome tests is important, which is another improvement that can be added to the Difference Engine. With proper metadata and analysis this process can even be automated.

Since the correlation calculations benefit from code errors causing tests to flip, increasing the occurrence of flips by intentionally introducing code errors through mutation testing would accelerate the rate by which the correlation database can accrue relevant data.

Initially we had plans for utilizing a more advanced analysis algorithm since we somewhat incorrectly assumed that the original algorithm would not produce as good results as it did. One idea was to realize part of this more advanced algorithm by employing Bayesian inference, which is a method of statistical inference that can be used to update the probability for a hypothesis as more evidence is acquired. We still believe this is a suitable and natural further improvement for evolving the algorithm.

The correlation weights given by the initial analysis of the current implementation of the Difference Engine can seamlessly be used as the prior probabilities for which code packages and tests are correlated. Bayesian inference is a tool that has been used effectively to improve machine learning and statistical methodologies over a broad field of sciences.

Finally, it is vital to find a way to automate the selection of tests and seamlessly integrate it into the developer workflow and code verification process to ensure that employing advanced RTS strategies is as easy as possible for its intended users.

## X. CONCLUSION

The desired outcome of this research is to minimize the time of the regression testing feedback loop during the day when developers are working on the code. During the night, nightly builds will ensure that the whole regression suite is run, to catch any potential regressions the minimized test selection might have missed, and to update the database with historical data to be used in subsequent analyses for finding correlations between tests and code packages.

We found that test selection based on historical data can show marked improvements in regression test execution time while still finding almost as many faults as running all tests would. While performing the narrow test selection might on average appear to yield 20% less recall than running all tests, it is important to remember that running all tests is a very expensive insurance policy, and that in most cases the narrow selection included all relevant tests.

However, we assume that any selection algorithm will miss out on important tests, and therefore we strive to provide a selection algorithm that can be continuously adjusted and calibrated. The test recommendations given by the Difference Engine are essentially self correcting, if one makes sure that the latest build is added to the pool of analyzed builds. In other words, one might miss out once on a flipping test, but the next time it will be in the list of recommended tests, if the analysis is performed continuously. This self correction comes at a price, however. As the recall is increased and as more tests are added to the list of recommended tests, the precision will inevitably drop further.

The cost saving potential of implementing intelligent test selection for the daily regression tests at Axis is considerable,

and the risks involved are only slight in comparison to the money that can be saved. The RTS strategy suggested show a potential of a 20-fold increase in integration productivity.

## ACKNOWLEDGEMENTS

This work was primarily funded by Axis Communications (www.axis.com), and partly funded by ELLIIT (The Linköping-Lund Initiative on IT and Mobile Communication, www.elliit.liu.se)

## REFERENCES

[1] E. Engström and P. Runeson, "A Qualitative Survey of Regression Testing Practices," in *Product-Focused Software Process Improvement*, ser. Lecture Notes in Computer Science, M. Ali Babar, M. Vierimaa, and M. Oivo, Eds. Springer Berlin / Heidelberg, 2010, vol. 6156, pp. 3–16.

[2] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[3] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, Jan. 2010.

[4] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 184–208, Apr. 2001. [Online]. Available: http://doi.acm.org/10.1145/367008.367020

[5] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173–210, Apr. 1997. [Online]. Available: http://doi.acm.org/10.1145/248233.248262

[6] A. Beszedes, T. Gergely, L. Schrettner, J. Jasz, L. Lango, and T. Gyimothy, "Code coverage-based regression test selection and prioritization in WebKit," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2012, pp. 46–55.

[7] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the ACM 24th International Conference on Software Engineering*, 2002, pp. 119–129.

[8] G. Wikstrand, R. Feldt, J. Gorantla, W. Zhe, and C. White, "Dynamic Regression Test Selection Based on a File Cache An Industrial Evaluation," in *International Conference on Software Testing Verification and Validation, 2009. ICST '09*, Apr. 2009, pp. 299–302.

[9] E. Engström, P. Runeson, and G. Wikstrand, "An Empirical Evaluation of Regression Testing Based on Fix-cache Recommendations," in *Proceedings of the 3rd International Conference on Software Testing Verification and Validation (ICST'10)*, 2010, pp. 75–78.

[10] K. Rees, F. Coolen, M. Goldstein, and D. Wooff, "Managing the uncertainties of software testing: a Bayesian approach," *Quality and Reliability Engineering International*, vol. 17, no. 3, pp. 191–203, May 2001. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/qre.411/abstract

[11] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang, "A history-based cost-cognizant test case prioritization technique in regression testing," *Journal of Systems and Software*, vol. 85, no. 3, pp. 626–637, Mar. 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121211002780

[12] G. Rothermel and M. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[13] J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test application frequency," *Software Testing, Verification and Reliability*, vol. 15, no. 4, pp. 257–279, Dec. 2005. [Online]. Available: http://doi.wiley.com/10.1002/stvr.326