Applying Regression Test Selection on the example of Business Central

Diego Joshua Martinez Pineda (diem@itu.dk)
Supervisor: Mahsa Varshosaz (mahv@itu.dk)

December 14, 2021

Abstract

In this research project, we address the problem of implementing a test selection algorithm on the Business Central project. For the scope of this project, a single test selection algorithm was implemented, which is based on test coverage information. The findings of this work should serve as a foundation to explore the different selection and prioritizations techniques in this system. Much of the work presented here went into understanding the context and setting of the system, sufficiently to execute the algorithm in actual revision examples. The insights acquired provide a basis for future evaluation settings to increase the validity of the results here presented and of future algorithms to be evaluated.

1 Introduction

When working with large software systems, regression testing is a valuable technique that increases the confidence of the product stakeholders that future revisions, new features, and evolution of the system will not break previous functionality.

For this reason, it is common for products to have the execution of the regression test suite as part of the main pipeline of the product. However, when systems are sufficiently large, it becomes very expensive to run the full suite of tests.

A test selection algorithm aims at choosing a subset of a full test suite to run, that is not as expensive as running all tests cases. However, this selection must preserve the capabilities of fault detection that the full test suite has. This means that careful selection must be made so that the confidence in fault detection is not lost because of not selecting some test cases.

Business Central is an Enterprise Resource Planning software (ERP) software system from Microsoft, it is a large software system, with different sets of tests for the diverse aspects of the product.

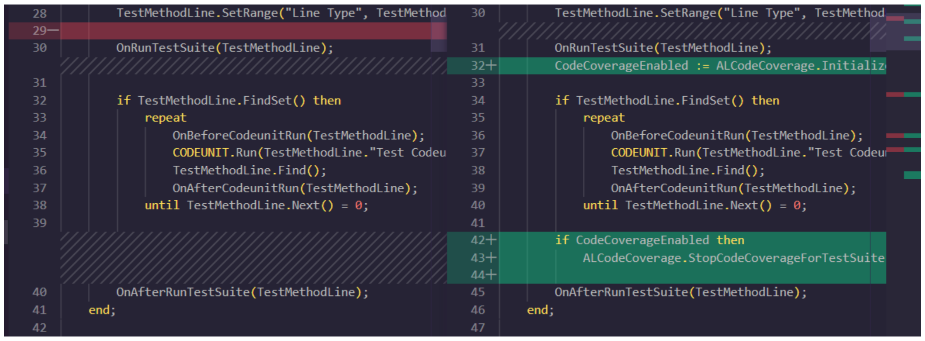
For the test selection explored in this project, the test suite of the application code (business logic) is explored. The business logic of the system is written on

AL, a DSL similar to Pascal that has primitives for interacting with the runtime of Business Central.

We would like to explore different test selection techniques for this system, to reduce the time and computing power required to detect faults in the current CI pipeline.

The idea for the method explored in this research project is to use the line coverage information that each test case produces. By looking at which lines were modified on a given change, and given that we know which tests were covering those particular lines then we would select those tests.

For example, see figure 1. Here we can see a `diff` of an AL code file on a given revision that added lines in between what previously were lines 30-31, and 39-40. Also what was previously line 29 was removed. Therefore, provided that we know which tests were hitting those lines, then we could propose them as tests to be selected.



```
28          TestMethodLine.SetRange("Line Type", TestMethod
29-
30          OnRunTestSuite(TestMethodLine);

31

32          if TestMethodLine.FindSet() then
33              repeat
34                  OnBeforeCodeunitRun(TestMethodLine);
35                  CODEUNIT.Run(TestMethodLine."Test Codeu
36                  TestMethodLine.Find();
37                  OnAfterCodeunitRun(TestMethodLine);
38              until TestMethodLine.Next() = 0;
39



40          OnAfterRunTestSuite(TestMethodLine);
41      end;
42
```

```
30          TestMethodLine.SetRange("Line Type", TestMethod

31          OnRunTestSuite(TestMethodLine);
32+         CodeCoverageEnabled := ALCodeCoverage.Initializ
33

34          if TestMethodLine.FindSet() then
35              repeat
36                  OnBeforeCodeunitRun(TestMethodLine);
37                  CODEUNIT.Run(TestMethodLine."Test Codeu
38                  TestMethodLine.Find();
39                  OnAfterCodeunitRun(TestMethodLine);
40              until TestMethodLine.Next() = 0;
41
42+         if CodeCoverageEnabled then
43+             ALCodeCoverage.StopCodeCoverageForTestSuite
44+
45          OnAfterRunTestSuite(TestMethodLine);
46      end;
47
```

Figure 1: A revision change on AL code.

Intuitively, this seems like a good heuristic to select tests. However, we do not claim it is sufficient, further examples can show that this heuristic fails in some cases. For example, when a part of the code is dependent on the side-effects it produces; then changing the environment through the side-effects of a revision, can cause a seemingly unrelated set of test cases to fail.

This method is meant to provide a foundation for further explorations on selection methods using different criteria besides this straightforward heuristic. We describe the different ways we can extend this method on section 6.

2 Background

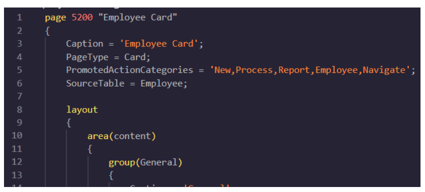
Before explaining details of the work done, we present for reference and clarity some of the minimal background knowledge required to understand how code and tests in AL are defined.

As mentioned in the introduction, AL is a DSL for Business Central and extensions for this system. The code is organized in several *AL objects*, which do not resemble objects in the traditional OOP sense, but objects from the system runtime. *AL objects* can be from different types, for example:

- *Tables.* Corresponding to underlying SQL Server tables.
- *Pages.* Corresponding to interactive pages the user can see.
- *Codeunits.* Modules with *procedures*, that is code that can be reused throughout the codebase

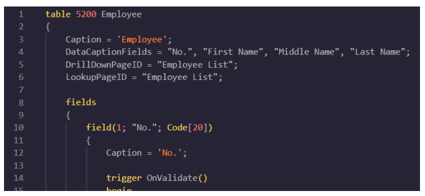
Each AL file in the codebase has as part of its definition, its type of object, and a numeric ID. Such a pair must be unique for a given codebase.

To make things more concrete, figure 2 shows an example of AL objects of types *page* and *table*. They are identified by both the type of object and ID, notice in this example, they share the same id: 5200, but they differ on the type of object.



```
page 5200 "Employee Card"
{
    Caption = 'Employee Card';
    PageType = Card;
    PromotedActionCategories = 'New,Process,Report,Employee,Navigate';
    SourceTable = Employee;

    layout
    {
        area(content)
        {
            group(General)
            {
```

(a) AL Page

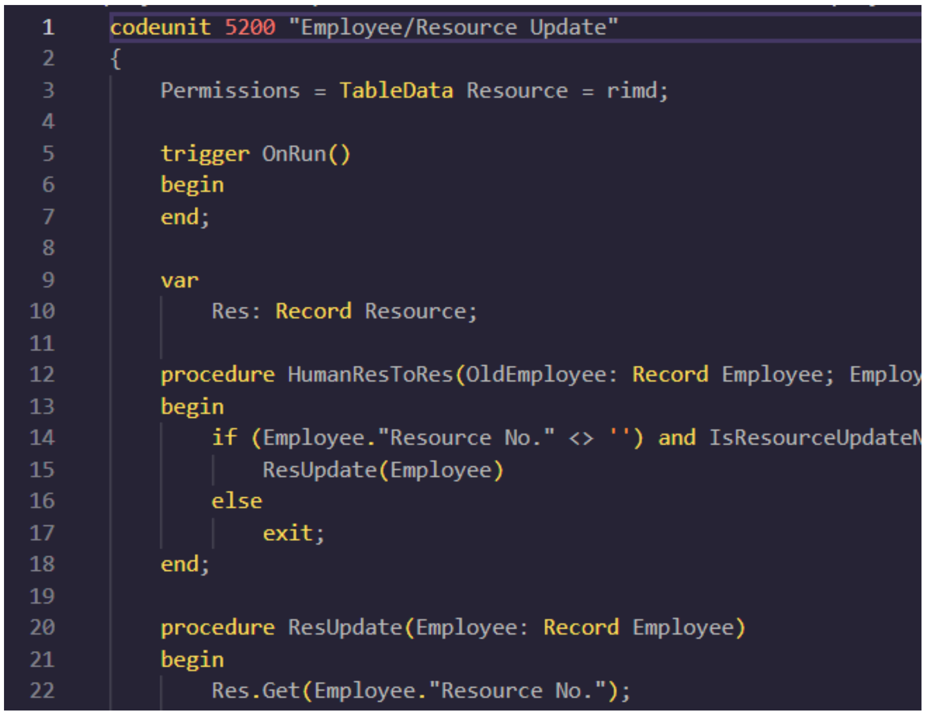


```
table 5200 Employee
{
    Caption = 'Employee';
    DataCaptionFields = "No.", "First Name", "Middle Name", "Last Name";
    DrillDownPageID = "Employee List";
    LookupPageID = "Employee List";

    fields
    {
        field(1; "No."; Code[20])
        {
            Caption = 'No.';

            trigger OnValidate()
```

(b) AL Table

Figure 2: Examples of *AL Objects*.

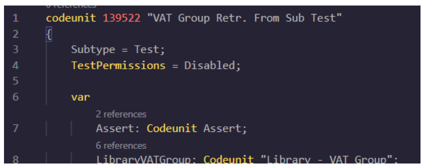
A mechanism for code reuse in this language is, as listed above, *codeunits*, which is similar to the concept of "modules" in other languages. They consist of several *procedures*. See an example in figure 3. Procedures defined on *codeunit*s can be called from any other object in the codebase.



```
codeunit 5200 "Employee/Resource Update"
{
    Permissions = TableData Resource = rimd;

    trigger OnRun()
    begin
    end;

    var
        Res: Record Resource;

    procedure HumanResToRes(OldEmployee: Record Employee; Employ
    begin
        if (Employee."Resource No." <> '') and IsResourceUpdateN
            ResUpdate(Employee)
        else
            exit;
    end;

    procedure ResUpdate(Employee: Record Employee)
    begin
        Res.Get(Employee."Resource No.");
```

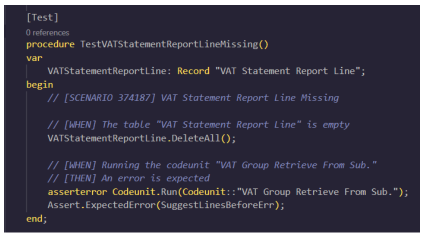
Figure 3: Example of an *AL Codeunit* with procedures `HumanResToRes` and `ResUpdate`.

Finally, we explain how test scenarios for AL code are defined. Unit and scenario tests for application code are written in AL themselves. They correspond to *codeunits* that are marked as tests. Procedures within such a *test codeunit* may correspond to scenarios to be tested. See figure 4 for a typical example on how *codeunits* are marked as tests and a scenario example. In the example, we see that the keyword `Subtype` is used to define the *codeunit* as a *test codeunit*. We can see a test scenario as a procedure on this *codeunit* called `TestVATStatementReportLineMissing`. A typical test would describe in comments the scenario being tested as shown.



```
codeunit 139522 "VAT Group Retr. From Sub Test"
{
    Subtype = Test;
    TestPermissions = Disabled;

    var
        2 references
        Assert: Codeunit Assert;
        6 references
        LibraryVATGroup: Codeunit "Library - VAT Group";
```

(a) AL Test Codeunit Definition



```
[Test]
0 references
procedure TestVATStatementReportLineMissing()
var
    VATStatementReportLine: Record "VAT Statement Report Line";
begin
    // [SCENARIO 374187] VAT Statement Report Line Missing

    // [WHEN] The table "VAT Statement Report Line" is empty
    VATStatementReportLine.DeleteAll();

    // [WHEN] Running the codeunit "VAT Group Retrieve From Sub."
    // [THEN] An error is expected
    asserterror Codeunit.Run(Codeunit::"VAT Group Retrieve From Sub.");
    Assert.ExpectedError(SuggestLinesBeforeErr);
end;
```

(b) AL Test Procedure - Scenario

Figure 4: Examples of definition of AL Test Scenarios.

3 Test Selection Method

3.1 Coverage information of test runs on AL

AL provides primitives to dynamically record line coverage information when executing arbitrary sections of code. This line coverage information has the following components, relevant for the selection method:

- Object ID
- Object type
- Line number
- Number of hits

As explained in the previous section, both the *Object ID* and *Object Type* identify a file in the codebase. The line number is for such a file.

It is relevant to stress that this information does not provide a file path on the codebase directly, but only a mean for identification.

Currently on the system, this primitive is used throughout the execution of the test suite. These measures test coverage information, providing the set of objects and lines covered by the suite, and it is used to impose coverage requirements for new additions to the codebase.

This means, that the information obtained by recording coverage throughout the complete execution of the test suite, cannot be used to distinguish coverage information between different test scenarios. The coverage information currently available on the pipeline of the system does not have information on which lines were hit by a particular test, but it only determines that it was hit by some test on the test suite.

However, previously in section 1, we described the method as selecting those test cases which covered a particular set of lines. With the current test coverage information, it is not possible to make that distinction.

For this reason, we needed changes on the test runner, to allow for individual tracking of coverage information per test case. However, these changes are currently not on the main pipeline of the product[1]. The coverage information it provides cannot be obtained from the CI pipeline of the system.

Another relevant technical consideration of how this information was dealt with, is how the output of the coverage information of each test case was stored. The modified test runner outputs a set of CSV files, one corresponding for each test procedure, containing the described coverage information together with the contents of the covered line.

As part of the work for this project, a different organization of this information was proposed and implemented. To allow for efficient querying of information in different ways, we proposed an SQLite database. A parsing-loading program was developed for this purpose. This provides a better organization for dealing with the information since the database is a single file. The reason for proposing a file database instead of a regular database is that coverage information can also be used for different purposes and easy distribution was desired.

Figure 5 shows the ER diagram in crow's foot notation of the DB schema proposed. AL Objects are modeled with their lines in a one-to-many relationship. These in turn are joined through the `coverage data` pivot table with the test procedures.

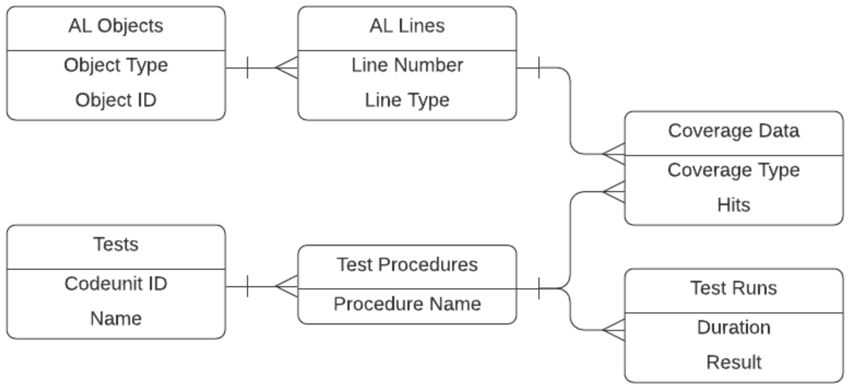


Figure 5: ER diagram for the proposed coverage database schema

3.2 Representing revision differences

The codebase with the business logic is composed of several AL applications.

When working on a new revision, some of the files on these applications are changed, removed, or added. The canonical way of treating with differences

[1]These changes were not work of the author, but from Nikola Kukrika nikolak@microsoft.com

between file revisions is with line `diffs`, see for example figure 1.

The output information of such `diff` is the set of file paths added, removed, or changed, together with the lines where these changes occurred.

However, in the previous section, we explained that the coverage information we can extract has as components the Object Id and Object Type where line coverage was recorded. While these properties correspond uniquely to a file in the codebase, directly checking the file path where code was changed does not lead to getting the Object Id and Object Type that the file has.

Furthermore, we are considering changes on the whole codebase, that may imply changes on tests as well. For example, adding new test procedures. For these kinds of changes, we would not like to have the same interpretation on tests that should be selected. Whenever test procedures are added, they should be at least selected. This would not be possible by only restricting criteria for test case selection to coverage information since these tests did not exist previously.

These two situations give way to thinking that more information than just the file path and lines changed is required from a given revision change. For this selection method, a lightweight parsing was performed on the given changed files to extract which Object Id is changes, and whether or not such object is a *test codeunit*.

Notice that the choice of what information to extract from a given revision change was just to address these two situations. We could however extract more information, enhancing our knowledge of changes caused by the revision. This could be even more granular, for example, to detect changes on specific procedures, addition of procedures, definition of new columns on a table. In general, semantical information related to changes. This could be a direction for future work, as input for more complex selection criteria. This is further explained in section 6.

For completeness, table 1 shows the information extracted for identifying a set of revision changes.

Type of change	Parameters
`AddObject`	None
`DeleteObject`	`removedObjectId`
`AddLinesObject`	`objectId`, `afterLine`
`RemoveLinesObject`	`objectId`, `firstLineRemoved`, `nLinesRemoved`
`AddTestObject`	`testCodeunitId`
`DeleteTestObject`	`testCodeunitId`
`AddLinesTestObject`	`testCodeunitId`
`RemoveLinesTestObject`	`testCodeunitId`

Table 1: Information extracted from revision changes

As part of this project, a program to obtain such representation was implemented. For reference, this consisted in taking the `diff` from the change, visiting each file on the appropriate revision, and parsing the file to obtain the required information.

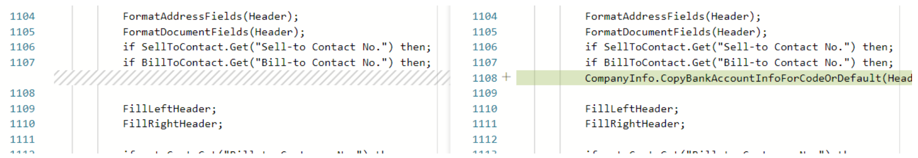
3.3 Test selection

Given that coverage information per test procedure is available, and revision changes can be understood in relation to test coverage information. We can specify the heuristic explained in section 1 by describing what tests should the algorithm select for each kind of revision change we observe. This is the main core procedure proposed for selecting tests on this project.

3.3.1 Adding lines to an AL object

Whenever lines are added to an AL object, they are added after a given line. The selected tests then are those that hit that line and the next.

For further illustration see figure 6, where we can see such a change. A new line was added between what previously was line 1107 and 1108. From the coverage information, we can obtain the set of test procedures that on the previous revision were hitting those lines. This method selects those tests.



```
1104        FormatAddressFields(Header);
1105        FormatDocumentFields(Header);
1106        if SellToContact.Get("Sell-to Contact No.") then;
1107        if BillToContact.Get("Bill-to Contact No.") then;

1108
1109        FillLeftHeader;
1110        FillRightHeader;
1111
```

```
1104        FormatAddressFields(Header);
1105        FormatDocumentFields(Header);
1106        if SellToContact.Get("Sell-to Contact No.") then;
1107        if BillToContact.Get("Bill-to Contact No.") then;
1108 +      CompanyInfo.CopyBankAccountInfoForCodeOrDefault(Head
1109
1110        FillLeftHeader;
1111        FillRightHeader;
1112
```

Figure 6: Example of a change that added a line between what previously was line 1107 and 1108.

3.3.2 Removing lines from an AL object

For removed lines, any test that was hitting what were previously those lines are selected.

3.3.3 Adding an AL object

We can not infer any test run from a new object, since coverage information would not include it.

3.3.4 Removing an AL object

We can understand this case as being the same as removing all the lines of the object. Therefore any test hitting that object should be selected. It is worth noticing that this is not a traditional revision change, but it is taken into consideration for completeness.

3.3.5 Adding lines to an AL test codeunit

The current representation for the revision change does not distinguish if the lines added are because of adding new test cases on a given test codeunit, or because of changing a test procedure. Both cases could be treated differently if the model for the revision change allowed for that.

Instead, we only keep the information about which test codeunit lines were added. As a conservative approach, in this method, we select all test procedures inside this test codeunit.

3.3.6 Removing lines from an AL test codeunit

Similar to adding lines to an AL test codeunit, as a conservative measure, all tests in this test procedure are selected.

3.3.7 Adding an AL test codeunit

For this case, we select all the test procedures in the new test codeunit.

3.3.8 Removing an AL test codeunit

When the change is removing a test codeunit, we do not select any test case. However, the Object Id of the removed test codeunit is kept, to filter our selection by removing any references to such tests.

3.4 The full algorithm

We give an overview of how different parts of the selection method explained in the previous section compose together.

A codebase at a given point in time can be enriched when running the test suite with the information of the lines covered by each test case. This is achieved through the modifications on the *Test Runner* application explained in section 3.1. This information is stored in a coverage database.

When a revision is given, we know the set of file paths and lines changed. We can leverage from parsing such files on each revision to gather more information about what this revision change implies. The information extracted is to allow for different test selection criteria for each type of change, as it is listed in section 3.2, table 1. In general, consider this step as extracting an expressive representation of the revision changes.

With this information, the test selection criteria described in section 3.3 is then applied, resulting to a set of selected test cases.

Figure 7 shows a diagram to further illustrate the above description.

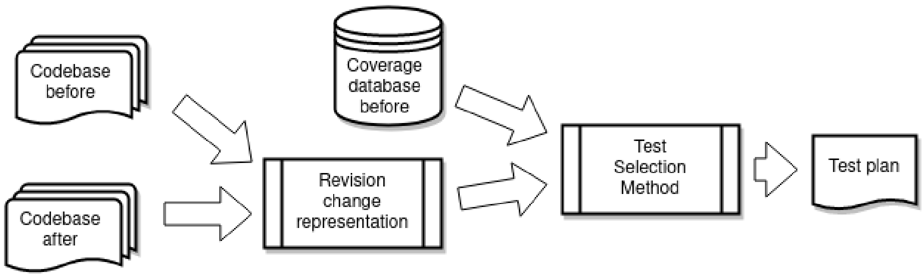


Figure 7: Diagram of the procedure for test selection.

4 Evaluation

4.1 Metrics

In the literature for the problem of test selection, several metrics have been proposed to quantify the different aspects addressed by a selection algorithm.

For example, counting how many methods were selected, relates to the expected performance improvement when executing this reduced test suite. After all, choosing all tests is a valid selection algorithm, although not ideal for our goal.

Another example is measuring how effective the algorithm was to select the faulty test cases. If our selection is very small it would be efficient, but if such selection is not capturing the faulty tests, then it is damaging to the product quality.

We select metrics based on a survey done by Pan, Bagherzadeh, et. al. in [1] on this topic. One of the research questions they dealt with, was the metrics used to evaluate the Test Selection and Prioritization (TSP) problem. Additionally, other common metrics were used.

4.1.1 Test selection execution time

This metric measures the total time of running the selected subset of tests, against the total time that it takes to run all of them.

Ideally, we would like this value to be as small as possible.

4.1.2 Test selection time to first failure

This metric measures the time taken for test execution until the first failure is detected, according to the proposed plan; with respect to the total execution time of all tests. This is a metric for evaluating a test prioritization technique, since a selection may have different orderings, and each order would change the value of this metric.

In such context, we take the selection to be the prioritized set and take one of the orderings arbitrarily. This is a relevant metric to implement since in further work we would like to also evaluate prioritization techniques.

Ideally, we would like to detect failures as quickly as possible, that is, the time taken to run the test that fails the quickest.

4.1.3 Inclusiveness

This metric refers to the ratio of the failing test cases included out of the total number of failing test cases. Therefore, for a test selection algorithm that includes all the failing test cases, we would aim to have a value close to 1 for this metric.

4.1.4 Selection size

For this metric, the ratio of the selected tests against the total amount of tests is taken. Ideally, we aim at as low number as possible while detecting every failing test case.

4.1.5 Normalized APFD

Average of the Percentage of Faults Detected is another commonly used metric to measure the effectiveness of Test Prioritization algorithms. First introduced by Rothermel et. al in [3].

This metric measures the area under the curve of the plot of tests run against faults detected. The normalized version takes into account possibly not detecting faults, which is the case for a selection algorithm. For completeness, the expression for computing it is given in (1).

$$nAPFD = p - \frac{\sum_{i=1}^{m} TF_i}{nm} + \frac{p}{2n} \tag{1}$$

Where p is the ratio of detected over total faults; n is the total number of test cases; m is the total number of faults, and TF_i is the ranking of the first test case that revealed the fault of the i-th failure.

This value measures the proportion of area covered by the described plot, therefore we would like this area to be as close to 1 as possible.

4.2 Selecting real revision examples

In this section, we describe the process and difficulties of providing examples of real revision changes that made a test fail. We also outline the required changes for future work. We present the metrics obtained from running the evaluation in a single failing revision example. While we are aware of the limitations to argue given these results, the main outcome was the engineering knowledge of the process to build a proper evaluation dataset.

The build system used for Business Central is referred to as DME. Roughly described, when a developer submits a test job, its branch gets built and tests get to run against it. After some time, the developer receives information on whether or not the changes can be integrated into the main branch for the product release being worked on.

A test job is composed of several interdepending tasks, and building the target system for which the tests are run is not a straightforward process. As an example, the codebase at a given point in time is a representation of several AL applications, each of which can be localized (country-specific logic organized within *layers*). This means that running tests involves localizing the codebase for each country, building each application, building the appropriate databases, and executing different test solutions on it.

Trying to match this with the simple context we had so far analyzed, increased the complexity for properly evaluating. However, the DME system is capable of being queried for further integrations, and in principle most of the process to get the evaluation point is possible, and to a certain degree can be automated.

For this project, a program was made to query and extract information of this system, to collect failing test jobs on real revisions.

The main complication that we faced was getting a reproducible revision example, together with the long time that these heavy operations take, as they are usually performed on clusters of machines for this purpose.

Nevertheless, we present the case of job 2412636, a case where we were able to extract the line coverage information and get a prediction with the test selection method proposed. In this test job, changes to an extension `INTaxEngine` were made. The corresponding test extension where failures were found is `TaxEngine-JsonExchange`. After obtaining the codebase for such revisions, the database and views were constructed for the failing version. In this case, the localized version corresponds to IN. To this setup, the modification to the test runner was applied, and the test suite was run again on both revisions, collecting the coverage information for the initial revision.

Afterward, the test selection algorithm was executed. Figure 8 shows the output of the selection algorithm and figure 9 the failures that occurred in the build system. We can observe that for this case, the inclusiveness results in 0. No selected test was failing. When looking at the stack trace of the error, we can see that the failure was due to not having permissions on new tables after their modifications. These types of faults are hard for our current algorithm to propose since they depend on the context that the revision introduced. This is not corresponding directly to statements or lines on the codebase, and therefore only using line coverage information does not suffice. In the next section, when evaluating against injected faults on the system, a class for the proposed changes made is context-faults that resemble faults like the ones detected for this job.

```
136802 TestInitializeTaxRates
136802 TestUpdateTaxRates
136802 TestUpdateFromAndToRangeAttributeOnTaxRates
136802 TestUpdateLinkdAttributeOnTaxRates
136802 TestUpdateFromAndToRangeAttributeForError
```

Figure 8: Output of the Selection algorithm for job 2412636.

```
137700 TestImportTaxTypes
137700 TestExportTaxTypes
137700 TestExportNodes
137700 TestUseCaseExecution
137700 TestUseCasePosting
137700 TestTaxInformationFactbox
137700 TestUseCasePostingExecution
```

Figure 9: Failing tests on job 2412636.

The program built for querying the build system provided different failing test jobs together with the log files associated with them. We could use them as starting point to collect the required information and attempt to reproduce them. However, we faced difficulties in streamlining this process. We list some of them.

The failing job could be dismissed if the job was submitted as a private deployment for the developer, since developers can request jobs from the pipeline without publishing their changes to the remote repository, making it inaccessible for us to obtain.

If the pull request associated with the test job was closed, typically the branch gets deleted as well, making it impossible to get the commit where the failing job was executed.

Also when obtaining the database and binaries associated with the build version, typically only the latest are provided to the developers since those are the ones required for their work. But to reproduce some of the job errors, it required in some cases to change the version of the binaries used. Overall, this made the process prone to failure and it needed a large investment of time.

In future work, we aim to integrate the collection of evaluation information on the CI pipeline, so that reproducibility errors are not an issue. With these changes, we could use the existing build system and resources to produce the required information at the time of failure. To do this, changes would be required on the build system, which themselves live under the main application repository in the form of Infrastructure-as-Code. We expand more on this in section 6.

4.3 Injecting faults on the codebase

Given the information challenges concerning reproducing real examples of code changes and the corresponding test failures within this project's timeline, we use fault injection to build a set of examples for evaluation. Modifications were purposefully made to make part of the tests fail. We did this however to state that we are capable of computing and gather some insight on the results of this selection, although we are aware that such might be skewed.

We used two test extensions from the real application: `Bank` (Test cases 1-4) with 454 test procedures and `VATGroupManagement` (Test cases 5-8) with 75 test procedures.

We can identify some classes on the mutations made, based on the type of modifications. However, we do not argue they are representatives of the actual test failures that the system usually experiences.

Depending on how lines are mapped to statements we classify the kinds of bugs introduced:

- Context-fault. Lines changed for this kind of bugs did not correspond to statements on the execution, but in the side-effects they produced. Test cases #3, #4, #7, and #8 belong to this class.
- Statement-fault. Lines changed correspond to statements on a sequential procedure and therefore are easily captured by the heuristic we used.

Test case	Type	$t_{Full}(\%)$	$t_{FF}(\%)$	i	$size(\%)$	$nAPFD$
#1	Statement	60.028	0.044	24.615	34.549	20.004
#2	Statement	0.058	0.027	100	0.796	99.867
#3	Context	71.006	NA	0	44.562	0
#4	Context	84.198	NA	0	49.337	0
#5	Statement	35.402	17.968	100	21.212	87.121
#6	Statement	33.216	33.216	100	21.212	79.545
#7	Context	0	NA	0	0	0
#8	Context	37.058	NA	0	24.242	0

Table 2: Evaluation of injected faults.

In the previous table, t_{Full} refers to the percentage of time taken to execute the complete selection, t_{FF} is the percentage of time until the first failure was detected, i is the inclusiveness, $size$ and $nAPFD$ are given as a percentage.

The results in table 2 show some of the insight we had previously discussed. When changing lines from a sequential procedure, the coverage information is useful and inclusiveness is significant. However, when changes related to the context were made, the inclusiveness was zero.

In some cases, no test had coverage information for the lines changed, and the selection was empty.

It is also relevant to note that the size of some of these selections was considerably large. For example, in cases #3 and #4, nearly half of the test cases were selected, but none of those were the faulty ones. Observing the time of fully executing for case #4 shows it was 84.198%. A possible explanation for the large size is that some test cases have a large density of lines covered. This is the case of tests in the `Bank` extension, where the test case that covers most lines covers 10285 lines, and the average of lines covered is 4191.90. In contrast to tests in the `VATGroupManagement` extension with maximum lines covered 1778 and an average of 365.64.

5 Related work

In this section we expand on the comparison with methods found in literature, to also outline possible directions for future work. For this, two surveys were used: the work from Yoo and Harman in [2] for general minimization, selection, and prioritization techniques, to put in perspective the current approach with the academic context. And the work from Pan, Bagherzadeh, et. al in [1] where they focus on techniques using Machine Learning, to understand what inputs are required to understand how applicable these techniques can be in future work.

The implemented technique here explored follows quite precisely the formulation for identifying fault-revealing test cases given by Rothermel and Harrold in [3]. Here they define a modification-traversing test as such that results on different execution traces for two given revisions of a codebase. Such tests can happen when either they execute new or modified code in the new revision, or if they formerly executed code deleted in the new revision.

By using the *Controlled-Regression-Testing Assumption* they can prove that modification-traversing selected tests cases are a superset of the possible fault-revealing tests cases. Such is an strong assumption, and in our case it does not hold, as we gave insight for in section 1. There are faults that test cases reveal that are context-aware, they do not correspond to the execution traces of the test cases. For instance, the real evaluation example described in section 4.

The algorithm they propose *SelectTests* compares Control Flow Graphs (CFG) generated from each test case, and when different, select all the tests that reached such nodes. We can understand our approach as a loose approximation of this, we do not rely on generating a CFG for the execution of the new revision of test cases, but we directly interpret line operations on files as a difference and select the corresponding cases. We differ on how we decide whether or not execution traces of a test are equivalent.

Work and terminology introduced by Rothermel and Harrold were foundational for the area and it has been used as a framework to communicate results in surveys and other papers. That is the case of another similar approach by Volkolos and Frankl, like in this project, they also take `diff` information as input for their selection; in the same fashion, they characterize each line operation, and associate a set of tests depending on the execution traces from previous revisions. The main difference with what here was presented is the preprocessing of revisions into a normal form to have a better correspondence between lines and execution traces. This is a direction that could be interesting to explore to increase the precision of selected test cases.

The similarity between the CFG traversing approach of Rothermel and Harrold with the textual difference approach from Volkolos and Frankl was hinted at when we compared how similar our approach was with the CFG approach. This was also observed by Yoo and Harman on their survey, noticing that the textual approach operated on a different representation of the system, but the behavior was in essence very similar.

Another related approach that could hint a direction on an extension of the technique here proposed is the work by Chen et. al. [8] where the CFG is extended by changing the definition of how a program can be partitioned. Changing this representation allows for more entities to be considered. Extending the representation of AL files, and giving a way to relate coverage information with different representations could allow more information to be captured, increasing the inclusiveness of our approach.

A very similar approach was also found in [4] by Beszédes et. al., here they describe the implementation of a coverage-based selection technique on the C++ components of the WebKit project. They follow a similar approach in the technical perspective, by implementing a procedure-level coverage database instead of a line coverage database. They identify changed modules from a revision change and execute the related test cases according to their coverage database. This is an analogous process to the one we follow, only taking a coarser perspective by grouping the different changes on the procedures it affected.

On the other hand, a more recent survey focuses on the techniques using Machine Learning algorithms [1], as they have been the subject of much interest in recent years. Rather than comparing them with the proposed selection algorithm, we present some of the applicable techniques that were found, for the focus of future work.

Conveniently, the survey exhibits a classification of the methods by artifacts and information provided by the papers. This can be very useful if we want to compare different selection criteria with the tools provided by the authors, adapting the collected information to their required inputs. We elaborate on the techniques found.

The technique proposed by Mahdieh, et. al. in [5] uses a neural network after extracting features from the codebase. These features consist on different metrics for the faulty revision, for instance, complexity metrics, coupling metrics, documentation metrics, or inheritance metrics defined for Java programs. It also uses the code coverage grouping by procedure as the approach from Beszédes et. al. described above. Such features are then the input of a neural network binary classifier that predicts whether or not a test case will be a fault-revealing one. In this technique, code coverage is used for prioritization.

The family of techniques presented by Cruciani et. al. in [6] and [7] take a different approach, by noticing that TSP is a big data problem and proposing techniques based on similarity. The main benefit of these approaches is that the core procedure in which they are based does not require any information but the test cases themselves. They use a vector space representation for each test and use clustering algorithms to determine which tests cases to use. This makes them highly scalable. The input of the tool provided is this vector space representation of the test cases. We could adapt generating such representation from AL test cases, as the features required rely on textual information and randomized projections.

6 Future work

Throughout this report, we have hinted at possible directions for future work. In this section, we summarize and expand on them.

6.1 Different revision representations

Among the different techniques that were shown to be similar in section 5, a missing step on the current implemented technique was getting more information about the change made. For example, in [4] they identify the procedures where the changes were made by using different source code representation tools. Also in [3], they depend on using a CFG representation of the tests, to match such information and identify in which node of the graph were changes made.

A future direction can be extracting more semantical information from a given change. AL compiles into C# code, which is used by the runtime server of the application to execute the required business logic. C# is a widely studied language with several source code representation tools that we can leverage.

This is also relevant to explore Machine Learning techniques since a common step for most of these techniques is feature extraction. This consists of extracting more information, from the source code, coverage information, and historical information about the test cases.

6.2 Applicable techniques from the literature

In section 5, we went over the different existing approaches to regression testing that followed similar ideas to the one proposed here. Some of the insight collected from such implementation can aid our algorithm and could be a way for extending it, like preprocessing files into a normal form or having different representations for AL files.

We also listed some applicable ML techniques, that use the information we have available, or for which some changes are required. Those are the family of FAST techniques from Cruciani et. a.l. and the neural network approach of Mahdieh, et. al. we aim is to compare and extend the technique proposed in this project with their provided artifacts.

6.3 Infrastructure for integrating evaluation on CI/CD pipeline

To increase our confidence in the validity of the results, we need a more robust evaluation method. For this project, we obtained manually the revision information of existing failing jobs and re-executed the test job failures collecting the coverage information to have the required inputs for our selection algorithm. This process was shown to be hard to reproduce, as explained in section 4.2.

A better approach to this problem would be not to reproduce the error as it happened while collecting the required information. Instead, to collect that information from the job itself. This would require changes in the infrastructure pipeline.

The pipeline and plan executed by the build system are kept in the same repository as Infrastructure-as-Code. Modifications to it would result in executing different tasks, like the collection of the coverage information. The scope of future work here should focus on making modifications to these task execution models for evaluation data collection.

6.4 Efficiency of producing coverage database

A relevant engineering concern not addressed when introducing the proposed coverage database is the efficiency of producing the coverage database. While having a database allows for more efficient and flexible querying of information instead of the existing CSV schema of processing coverage; it comes currently with a toll in performance when populating the coverage database.

There are several optimizations possible regarding transaction management in SQLite and other configuration options for the engine like using prepared statements. This work item becomes increasingly important if looking to integrate the evaluation on the CI/CD pipeline since it should not be considerably more expensive to obtain such a database than the current process for collecting coverage information.

6.5 Other evaluation metrics

Related to the previous item, other metrics would give better insights to understand how likely would it be to adopt the techniques on the real system. As observed in [6], test selection is a process with several phases, following the literature these phases are: an analysis phase, an execution phase, and a collection phase. The current metrics only deal with the execution phase of the algorithm, however, the cost of collecting the coverage information currently is significant. Only measuring one of the phases skews the evaluation of how adoptable is this approach.

7 Conclusion

Throughout this project, we built some parts of the required infrastructure to run test selection techniques on real revision examples of Business Central. We also implemented a basic selection technique. There still exists a difficulty in streamlining the evaluation of real revision examples, however, we have given an outline of the required changes to increase the validity of our evaluation in future work. This would increase the dataset, and provide the required information for real test failures in the build system without the need of reproducing the failures.

We obtained however insights on the current limitations of the approach with the limited evaluation dataset built from injecting faults into the real system. One insight is the difficulty of selecting faulty test cases when the changes are on the side-effects rather than on sequential statement execution. Also, the large selection proposed by this approach when test cases cover a large number of lines.

We also located the approach in its context and qualitatively compared them with other approaches. Such comparisons led to ideas that could be used to extend the selection algorithm in future work. After this, we also evaluated techniques using Machine Learning that have artifacts available and require input information that we could obtain. [2]

References

[1] Rong Pan and Mojtaba Bagherzadeh and Taher Ahmed Ghaleb and Lionel C. Briand, *Test Case Selection and Prioritization Using Machine Learning: A Systematic Literature Review*, ArXiv, 2021, abs/2106.13891

[2] Shin Yoo and Mark Harman, *Regression Testing Minimisation, Selection and Prioritisation - A Survey*, 2009

[3] Rothermel, G. and Untch, R.H. and Chengyun Chu and Harrold, M.J., *Prioritizing test cases for regression testing*, IEEE Transactions on Software Engineering, 2001, volume 27 number 10 pages 929-948,10.1109/32.962562

[4] Beszédes, Árpád and Gergely, Tamás and Schrettner, Lajos and Jász, Judit and Langó, László and Gyimóthy, Tibor, *Code coverage-based regression test selection and prioritization in WebKit*, 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012, 10.1109/ICSM.2012.6405252

[5] Mostafa Mahdieh and Seyed-Hassan Mirian-Hosseinabadi and Khashayar Etemadi and Ali Nosrati and Sajad Jalali, *Incorporating fault-proneness estimations into coverage-based test case prioritization methods*, Information and Software Technology, 2020, https://doi.org/10.1016/j.infsof.2020.106269

[2] Artifacts for this project can be found in https://github.com/mynjj/research-project-itu

[6] Cruciani, Emilio and Miranda, Breno and Verdecchia, Roberto and Bertolino, Antonia, *Scalable Approaches for Test Suite Reduction*, IEEE Press, Proceedings of the 41st International Conference on Software Engineering, 2019, pages 419-429, 10.1109/ICSE.2019.00055

[7] Miranda, Breno and Cruciani, Emilio and Verdecchia, Roberto and Bertolino, Antonia, *FAST Approaches to Scalable Similarity-Based Test Case Prioritization*, Proceedings of the 40th International Conference on Software Engineering, 2018, pages 222-232, 10.1145/3180155.3180210

[8] Chen YF, Rosenblum D, Vo KP. *Testtube: A system for selective regression testing*, Proveedings of the 16th International Conference on Software Engineering (ICSE 1994), ACM PressM, 1994; 211–220.