FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Data intensive applications verification

Daniel Arménio Silva Mendonça

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rui Abreu Maranhão

July 22, 2017

Data intensive applications verification

Daniel Arménio Silva Mendonça

Mestrado Integrado em Engenharia Informática e Computação

July 22, 2017

Abstract

The main challenges for a Data-Intensive Application are the data quantity, the data complexity and the speed at which the data changes. For these applications, there are two predominant data categories, Master Data and Transactional Data. Master Data represents the data that is not frequently modified and that is required for the system to operate. Transactional Data refers to the data that is produced in the system operations. These applications require big investments in the development of tests and long test runs, due to their large size and the time that it requires to generate the tests data and to perform the tests.

Data-Intensive applications tests requires the pre-existence of master data. In the beginning of a test, the data that will be used is expected to be in a specific state, otherwise the test integrity cannot be assured. One of the possibilities to cope with these requirements is to generate the master data in the test set-up, which is the solution that Critical Manufacturing is currently using to test its Manufacturing Execution System (MES). A MES is used in manufacturing to track and document the transformation of materials, schedule tasks, control the flows and steps of the production and manage it in general.

In this project, it was developed a decision support tool to identify sets of tests with resemblances in their set-ups through static code analysis. By identifying the tests' local variable declarations of classes that are known to generate the master data, and then analyse their creation parameters and assignments to their members it is possible to estimate a resemblance value. Tests in the same set may eventually share the same Master Data, instead of generating new data for every time it runs. Also through static code analysis, a tool was developed to transform conventional tests into data-driven tests that allows to run different tests with the same code. The tools were developed for the test-suite of Critical Manufacturing's MES.

Resumo

Os principais desafios para uma aplicação de elevado volume de dados são, a quantidade de dados, a complexidade dos dados e a velocidade a que os dados podem mudar. Para essas aplicações, existem duas categorias de dados predominantes, dados mestre e dados de transação. Os dados mestre representam a informação que não é modificada com frequência e que é necessária para que o sistema funcione. Os dados de transação representam a informação utilizada e produzida pelas operações realizadas no sistema. A verificação deste tipo de aplicações requer um grande investimento e longos períodos de tempo para a execução dos testes.

Os testes para uma aplicação de grande volume de dados requerem que já existam no sistema os dados mestre. No início de um teste, os dados que vão ser utilizados devem estar no estado esperado pelo teste, caso contrário, a integridade do teste está comprometida. Uma das possibilidades de assegurar que os dados do teste estão no estado esperado é gerar esses dados no próprio teste, é esta a atual solução adotada pela Critical Manufacturing para testar o seu Sistema de Execução de Manufatura (MES).

Neste projeto, foi desenvolvida uma ferramenta de apoio à decisão que permite identificar conjuntos de testes com semelhanças na sua configuração através da análise estática de código. Testes que pertencem ao mesmo conjunto podem eventualmente compartilhar os mesmos dados mestre, evitando assim a geração de novos dados sempre que um teste é executado. Também através da análise estática de código foi desenvolvida uma ferramenta que permite a conversão de testes escritos da forma convencional em testes data-driven. As ferramentas foram desenvolvidas para a test-suite dos Sistema de Execução de Manufatura da Critical Manufacturing.

Contents

List of Figures

LIST OF FIGURES

Abbreviations

API	Application Programming Interface
CMF	Critical Manufacturing
ERP	Enterprise Resource Planning
FEUP	Faculdade de Engenharia da Universidade do Porto
LINQ	Language Integrated Query
LOC	Lines of code
MES	Manufacturing Execution System
MEF	Managed Extensibility Framework
MS	Microsoft
SDK	Software development kit
VS	Microsoft Visual Studio
VSIX	Visual Studio Extensibility
WIP	Work In Progress
WWW	*World Wide Web*

Chapter 1

Introduction

1.1 Context

Critical Manufacturing, founded in 2009, is a software house company that focus on the development of automation software for high tech industries as semiconductores, solar and electronics. Critical Manufacturing main product is a Manufacturing Execution System (MES), mainly developed in .NET technology and with Microsoft technology.

A Data-Intensive application centres its focus on it is data at a given time, and the result of most of the operations is bound to that data. Properties or types of new generated data commonly are the result of an operation over already existing data, the same is for data changes, that may be related with the previous state of the data that the operation used or was used over. This relation influences the subsequent steps of the execution process thus possibly limiting the following operations over that data. In a typical Data Intensive Application there are two main types of data, Transactional Data and Master Data. Transactional data is mostly associated with data generated from operations over data already present in the system or created through the already existing data, whereas Master data is not so volatile and doesn't change frequently during the application execution.

A MES is a data-instensive application. In a MES, the Master Data can be seen as the system configuration, such as the resources and the flow, whilst Transaction data can be for example the data produced during operations over a material.

1.2 Motivation

It is hard to maintain large test-suites, that naturally grow over time, as software is developed. Over the years there have been many suggested and implemented solutions for software testing with test-suite minimization, prioritization, hybrid and multi-objective techniques to reduce testing costs [Cos15] [Hal12] [YLW09].

With time the test suite will naturally grow if the software continues to be developed, and the larger the test-suite, the harder and more expensive to maintain it. Data-driven test-suites normally have less tests than conventional test-suites, that is because the skeleton of a test can be reused and

use values for the test from a data source, performing the test with different values as the data source provides them. The transformation of the tests into data-driven tests would also allow the creation of new tests by simply adding values into the data source from which the values are used in the test with no programming skills required, instead of writing code for a new test. When an existing test-suite was developed with conventional tests, depending on its size it can be very expensive and demanding to convert all the tests into a data-driven tests.

Critical Manufacturing's MES software tests are designed so the they generate the data needed to run which creates a high dimension database just for the tests, but some of this tests need the same data to be able to run, an indicator that the tests database size can be reduced. Being able to identify tests that need the same data to run will allow to identify which tests should be refactored to use a database transaction so they can share the same master data between without harming the integrity of the test, ensuring the data will remain as it was after running the test.

With the release of white box compilers such as Roslyn, there is now an opportunity with less code complexity to perform static code analysis over tests to explore new forms of test suites optimizations such as detecting similar tests [Mic17].

1.3 Objectives

This project intends to explore the capabilities of the white box compilers to perform static code analysis in a test suite to know if it possible to reduce the time required to run the tests.

The goal is to develop a tool able to find resemblances between tests, and as a decision support tool identify tests that need the same data to execute the test.

Due to the dimension of the current test suite of Critical Manufacturing's MES it also is intended to develop a tool that is able to convert conventional tests into data driven tests that work with Microsoft Tests Framework and integrated with Visual Studio, the company's most used IDE.

1.4 Document Structure

This dissertation is divided in 5 chapters.

In the first chapter 1 it is introduced the context of the dissertation, the identified challenges, the objectives and expected contribution.

Chapter 2 contains the literature review about MES, Data-Intensive applications, techniques used to reduce time and number of tests in the development of data-intensive applications and also in chapter 2 it is explained the advantages of a white box compiler, Microsoff Roslyn and its advantages to perform static code analysis.

In chapter 3 it is described the implementation and integration in Visual Studio of a decision support tool that is able to identify resemblace in tests set-ups.

Chapter 4 explains the advantages of data-driven tests over conventional testing and the implementation of a tool in visual studio that transforms a conventional test into a a data-driven test.

In chapter 5 are presented the conclusions of this work development and implementation, as well as possibilities for future research and work.

Chapter 2

Literature review

2.1 Data-Intensive Applications

An application is considered data-intensive when data is its main challenge, in volume, complexity and the speed at which it changes [Mar17]. As these applications' flows are mostly governed by the data content, testing demands a high variety of data to guarantee the detection of faults.

2.1.1 Manufacturing Execution Systems

Manufacturing Execution System, or MES, is an information system that drives the execution of manufacturing operations. The main goal of such systems is to achieve and to maintain high performance in a highly competitive and rapidly changing manufacturing environment [Cri17]. One of the challenges of a MES is to share the data gathered and generated with other information systems [BS10]. It is an important bridge between the shop floor and the enterprise higher level information systems that benefit when the systems are well integrated gathering more knowledge about the production process, to monitor and enforce the correct execution of the many steps that may be involved in the production while delivering work instructions. The gathered data of a MES is used to optimize procedures, detect and solve problems.

2.1.1.1 Benefits of MES

Some of the key benefits of a MES are related with the production phase, helping with the automation process thus reducing direct labour costs, data entry time, manufacturing cycle time, paperwork and WIP (Work in process) management. We can point some of the advantages by grouping them. At the factory management level for instance, it allows to keep track of the materials and resources, control the routing and dispatching, collect data and operation data store, generate reports, give an overview of the factory, and more. It also brings benefit in terms of quality, visibility and intelligence, scheduling, operational efficiency and in the integration and automation process.

One of the benefits not related with the process execution itself is the ability to keep records of the production process and material used in it, reducing the paper usage and reduce traceability time in case of an anomaly detection in the product.

2.1.1.2 Generated Data

As MES help to monitor and control the execution process, great volumes of data flow through the system and are stored. Information such as the product, the materials used to produce the product and the process that needs to be followed. Materials need to go through different steps and may have to follow different flows, passing through different resources that can have one or more services.

2.1.1.3 Applications

A MES can be used in different industries but the most common are, semiconductors industry, solar industry, electronics industry and LED industry [Bei12]. However, in the volatile business environment of smart products and services, the software must be customizable and modular enough to be easily integrated with new industries and businesses [Cri17].

2.2 Software Testing

Software tests are used to check if a system meets different requirements such as reliability, performance, behaviour validation and others. In an on-going software development project, its test suite will grow as more code is written. To ensure that the software was not broken with the new changes, the tests are executed again, this is called regression testing. In a large project running the complete test suite can take several hours rising the project's costs.

2.2.1 Minimization

Selecting tests to remain in the test suite or to be or not executed implies the comparison of pair of tests, to do so there's a need to compare each test code coverage impact. There are several tools to profile the code coverage of tests [YLW09].

Some established metrics of comparison between tests are the Coverage difference that compares the code coverage between tests, control variance which is the distance between test cases based on the difference in control statement executions, temporal variance, and other metrics [RGR16]. The combination of metrics with different weights provides good reference for the test selection phase in the test suite minimization task. The study of these metrics lead a Microsoft research team to develop a tool that uses program profiles and static execution and clusters the tests through their classification, which then allow to make an informed test suite reduction [VCT09].

Even maintaining an approximated coverage in the test suite reduction process, there's the possibility of loosing fault detection ability [JH01]. Some techniques deliberately maintain overlapping tests in the test suite to face such problems [JG07].

There are other approaches able to produce more than one collection of minimal sets allowing the developer then to prioritize them such as RZoltar, a tool implemented using Eclipse's plug-in GZoltar [CA13]. This technique considers a program as a collection of components, a test as a tupple with an input and output and regards the test suite as a set of tupples.

Algorithm 1 Map Coverage Matrix into Constraints

Input: Matrix A, an error vector e, number of components M, number of test cases N
Output: Conjunctions of disjunctions Ω
$\Omega \leftarrow \emptyset$ ▷ empty conjunction
for all $i \in \{1 \ \ldots \ M\}$ **do**
    $\Omega' \leftarrow \emptyset$ ▷ empty disjunction
    for all $j \in \{1 \ \ldots \ N\}$ **do**
        if $\omega(i, \ j)$ **then**
            $\Omega' \leftarrow \Omega' \vee j$
        end if
    end for
    $\Omega \leftarrow \Omega \wedge (\Omega')$
end for
for all $k \in \{1 \ \ldots \ N\}$ **do**
    if $e(k)$ **then**
        $\Omega \leftarrow \Omega \wedge (k)$
    end if
end for
return Ω

$$\begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \end{array} \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \quad \begin{array}{c} e \\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \end{array}$$

Figure 2.1: Coverage matrix [CA13]

By tracking the components usage in each test it is possible to create a matrix that maps each test with a set of components and to maintain track of detected errors there's also a matrix that keeps track of each test result which can be seen in 2.1. The tool developed by Campos and Maranhão [CA13] can efficiently minimize the original test suite maintaining code coverage and fault detection by solving the constraint problem and obtaining the minimal hitting set. The constraint problem is the encoded relation between a test case and its testing requirements in a coverage matrix and mapping it into a set of constraints, solving the constraint problem means finding a set of tests that maintains the original coverage, minimizing the number of tests without losing coverage hence the name minimal hitting set.

2.2.2 Database state preservation

Testing a Data intensive application requires testing the application logic and its interaction with the database, in this way a test case consists not only of input and output parameters values and the procedure to invoke, it also comprises an initial database state [Hal12].

In such tests, it is needed to ensure that a system cannot reach a situation where the data is left in an inconsistent state caused by a business rule violation. Castro suggested an approach where the data is part of the model state, and to do so it is needed to define a record data structure and a transition structure to deal with the state changes and the different phases of testing sequence [CA11]. In Castro's approach the validation of tests is done through the use of database transactions that preserve the database original state, this is achieved by involving each test in database transactions and the database consistency is verified only at the end of each testing sequence before returning so it is possible to propagate the database changes to the following tests that can depend of an action performed in the current test and allow the program logic to do possible corrections upon expected violation of business rules [CA11].

2.3 White-Box Compilers and Static-Code Analysis

A traditional compiler takes the source code as input and gives back an assembly, this means that all the information gathered in the compilation pipeline process is never available to anyone and is lost as soon as the output of the compilation is given. A white-box compiler exposes the information across the different phases of compilation including syntactic and semantic information, a powerful resource for a developer to perform static code analysis.

2.3.1 Roslyn - Microsoft Code-Analysis

Roslyn is a compiler developed by Microsoft written in C# and supports .NET C# and Visual Basic programming languages. It offers the capabilities of a compiler but it is easily expanded and adapted. Roslyn exposes API's for syntactic analysis, semantic analysis, dynamic compilation and code emission. Differently of most compilers, Roslyn is a White-Box compiler, that makes it useful for this work as it allows to do a powerful static code analysis. The most relevant features of Roslyn for this work are the Syntax Tree API and the Symbol API.

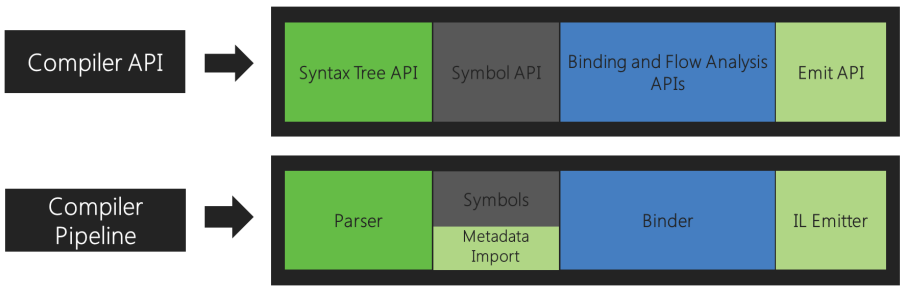

Figure 2.2: Roslyn pipeline and API

2.3.1.1 Syntax Tree API

The Syntax Tree API allows a user to obtain the syntax tree for a given document with code if written in one of the languages supported. The syntax tree is the most fundamental structure exposed by the compilers API and they represent the lexical and syntactic structure of the code [Mic17]. Beside the code analysis possibility, this offers the possibility to create tools to modify the code structure by rearranging, adding or removing syntax nodes in a code-aware manner, avoiding having to use direct text edits.

2.3.1.2 Symbol API

The Syntax Tree information alone is insufficient to describe all the declarations and logic in the source, it is not enough to to identify what is being referenced, for example types with the same name defined in different locations (even in other assemblies), for that we use the semantic model that it is possible to obtain through Roslyn Symbol API, the semantic model also encapsulates the language rules. With the semantic model of a source file obtained through this API you can use it to discover symbols referenced at a specific location, the resultant type of any expression, how variables flow in and out of regions of the source, diagnostics and answers to more speculative questions [Mic17].

2.4 Code clone detection

Some times in software development one may want to reuse a routine or fragments of it in another location. Similar code in software systems are called code clones and they are considered to be one of the most common code smells [SZ12] [FBB+99]. In previous studies was determined that the number of code clones increases in proportion to the growth of the code base [DGHZ11]. Some IDEs such as VS offers features that allow to select a part of the code and extract it as a new method so one can avoid repeating the code by replacing the repeated code by an invocation to the extracted method. Code clones are considered a potential hazard to the software, i.e. if we consider a scenario where one developer performs a bug fix in cloned code, the other clones will still have the bug if not manually updated too [DGHZ11]. Another application of code clone detection techniques is to find plagiarism for example in a context of assignments done by computer science students [BFL+10] [BG14].

2.4.1 Code Clone Detection approaches

There have been different approaches to detect code clones, Shafieian and Zou have grouped them into four groups; textual, lexical, syntactic and semantic [SZ12]. Different techniques use different approaches for structuring and preparing the information gathered from the source code and then comparing it. In figure 2.3 we can see the typical approach for clone detection, and even though the process can vary depending on the type of clone detection most have some of the steps shown [RCK09].

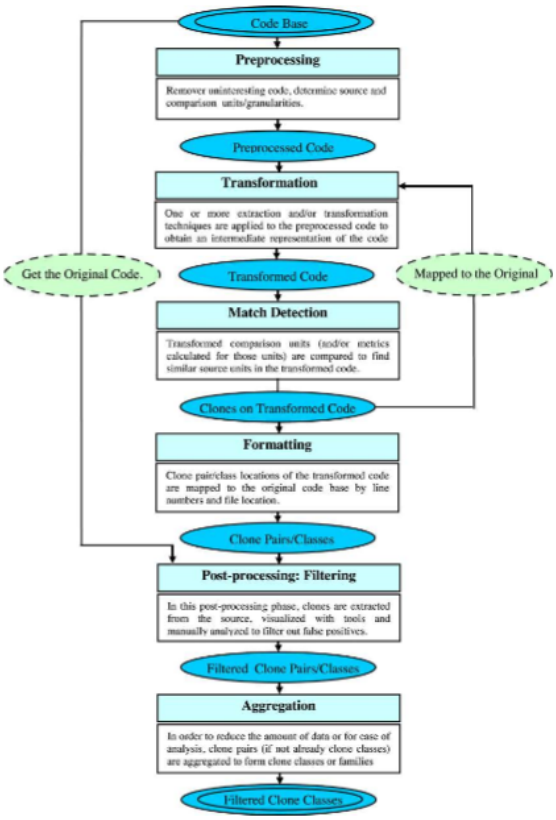

Figure 2.3: Common Code Detection [RCK09]

Textual approach for code clones Detection

It's used to find code similarities in the source code with little or no transformation. Some tools do a normalization of the text before comparison to reduce the amount of irrelevant text under analysis, in its essence it uses text comparison techniques with little awareness of the code or programming language [SZ12].

Lexical approach for code clones detection

This approach more robust regarding non code related text present in the source code files. It works by transforming the source code into a sequence of tokens in a similar form as a compiler does, and then searches for other sequences that match it or its sub-sequences [SZ12].

Syntactic approach for code clones detection

In a syntactic approach, the source code is transformed into a syntactic tree respecting the language rules, including the tokens values. The search for clones is done by taking a syntactic tree and locate in the program other semantic trees that have similarities with tree based techniques [SZ12].

Semantic approach for code clones detection

This approach is also language dependant. It takes into consideration the types of the variables and their flow through out the code. This approach has a higher abstraction of non relevant text and the order that it is written, it can be viewed as a graph and can be used graph techniques to find resemblances that allow to determine if they can be clones [SZ12].

2.5 Summary

In this chapter it was explained what a Manufacturing Execution System is, its applications, advantages and why it is considered a data-driven application. Some techniques applied to data-intensive applications test-suites to reduce the number of tests to run while trying to maintain the test coverage are mentioned and described. This work benefits from white-box compiler Roslyn's capabilities, how it can be used to perform static code analysis which allowed to develop a new approach for code-clone detection inside data-intensive applications tests regarding their set-up, and so it is done a review over these topics.

Chapter 3

Test Resemblance tool: Finding resemblances between tests set-up in data-intensive applications

3.1 Problem description

Critical Manufacturing's MES is developed mainly in C# programming language. The test suite used is a Visual Studio Solution, composed with many projects that test different features of the MES, and each test generates the master data that needs to perform the test verification each time it runs, so it is ensured that the state of the data to be used remains unchanged until the test's verification. In this work we call the generation of the master data needed for a test to run, its set-up. The tests' set-up in Critical Manufacturing MES' tests is always the same provided that it was not modified, and as verified during this project parts of a test set-up are replicated in other tests as seen in 3.5. Wrapping a test within a database transaction would allow to establish a constant master data that can be reused across tests and makes unnecessary to do the set-up of a test for each run, meaning that the database will be lighter and the execution of a test will be faster.

The problem that arises to obtain a constant master data is the identification of the tests that share the same set-up. In Critical Manufacturing MES' test suite, the set-up of the tests is the creation of scenarios that generate the master data, those scenarios can have relationships and/or dependencies between them for example a Product is generated through an instance of *ProductScenario*, and that product scenario may need a flow scenario that describes its flow in shop floor, and the flow may have step from step scenarios, all generated through the respective scenario classes and may have dependencies of other entities. The more dependencies an entity has, less likely is to be shared in between tests, and the more complex the entity set-up more difficult it is to determine if other entities have in fact the same representation, if they are clones.

3.2 Critical Manufacturing tests solution

3.2.1 Structure and organization

Critical Manufacturing's MES features visual studio tests' solution currently has 32 different projects to test different features or groups of features, those projects combined have a total of 899880 LOC and 3696 tests which currently need more than 5 hours to run (in dedicated machines). A Visual Studio solution contains projects, and each project contains documents that have the source code.

TestScenarios project is a dependency for every other project in the solution, as we can see in figure 3.1. This project contains the implementation code responsible for scenarios generation that subsequently creates the master data for tests usage. Given the tests solution's dimension, manually identifying the target constant master data is inconceivable, it is an arduous, lengthy, expensive and error prone task.

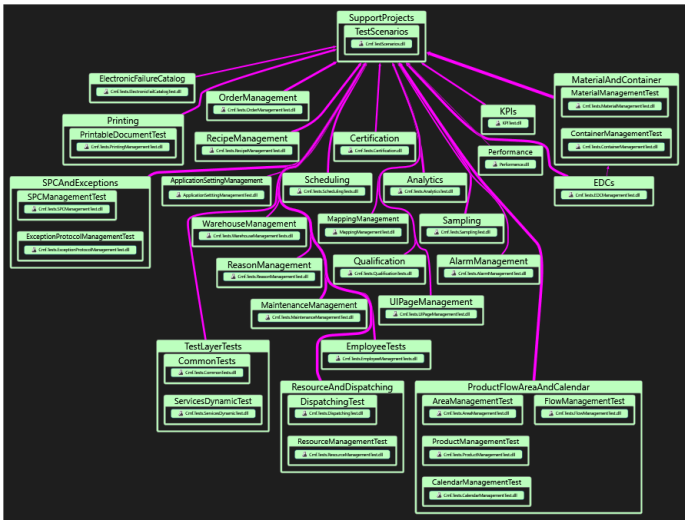

Figure 3.1: CMF Features Test Solution

3.3 Implementation

We use the Roslyn compiler to obtain compiler information about the tests solution. The base data structure of Roslyn is the workspace, which can contain one or more Visual Studio solutions and during this work it was both explored adding the tests solution into a manually created workspace and also obtaining the solution directly from Visual Studio environment. The architecture of one of Roslyn's workspace data structures can be seen in figure 3.2.

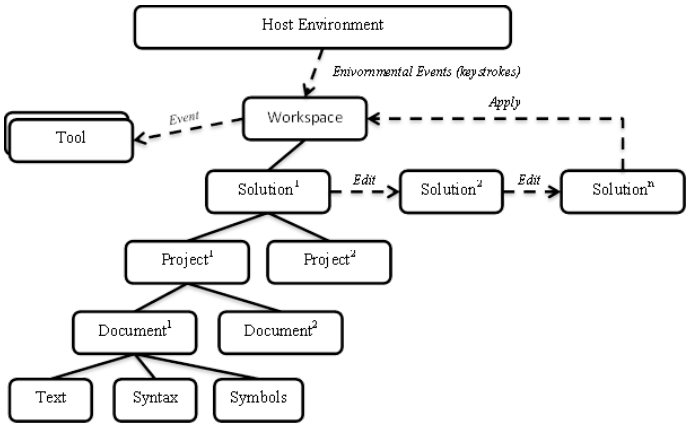

Figure 3.2: Roslyn Workspace architecture [Mic17]

3.3.1 Finding tests

To obtain all tests of the solution it is needed to explore all its projects, and for each project's document obtain its c# syntax tree to use with Roslyn's Syntax Tree API. The syntax tree root element of a C# code document is the *CompilationUnit*.

Roslyn's Syntax Tree API provides a class, *CSharpSyntaxWalker*, that contains a method *Visit(SyntaxNode)* with an overloading for each syntax node existing types, and for each syntax node it visits its children syntax nodes with a depth first approach. *CsharpSyntaxWalker* alone does nothing, but it is useful to extend this class and override the *Visit_NODETYPE_()* method for the type of syntax node(s) that is wished to analysed of perform an action over. Another viable option to explore specific types of syntax nodes is to use Microsoft LINQ to collect the desired syntax nodes, the disadvantage of this approach is its efficiency when targeting different kinds of syntax nodes.

```
1    [TestClass]
2    public class ResolveBOMForMaterialTest
3    {
4        [TestMethod]
5        public void ResolveBOMForMaterialNormalFlowTest()
6        {
7        // test body
8        }
9    }
```

Listing 3.1: Test declaration

A test using Microsoft visual studio default testing framework is a method with an attribute "TestMethod" that is within a class with the attribute "TestClass". A test syntax node kind is Roslyn is the same as any other method declaration, and its type is *MethodDeclarationSyntax*. What allows to determine that the method is a test is if its attribute list contains the attribute

above mentioned, "TestMethod". The attribute list syntax node kind is *AttributeList* and it is a direct child of the *MethodDeclarationSyntax* and the parent of the *Attribute* that contains the identifier whose value is "TestMethod". We can visualize the relation between a method, attribute list and attribute syntax nodes in figure 3.3, obtained from the *Syntax Visualizer* tool that ships with *Microsoft CodeAnalysis SDK* and is integrated with Visual Studio, the visualizer excludes the "Syntax" string from the various node types.

Figure 3.3: Method syntax tree outline

In this work it was implemented a class that extends Roslyn's *CSharpSyntaxWalker* named *TestMethodCollector* with overridden method *VisitMethodDeclaration(MethodDeclarationSyntax node)* that adds the method to a set if "TestMethod" is the value of one of the tokens of the *MethodDeclarationSyntax* attributes.

3.3.2 Identifying Scenarios

Each syntax node, if not a leaf, is the root of sub-syntax tree of the document's syntax tree. In a *MethodDeclarationSyntax's Block* node a declaration of a named variable is a *VariableDeclarationSyntax* syntax node type, naturally an instance of a scenario declaration is then a *VariableDeclarationSyntax*.

A *VariableDeclarationSyntax* must have a child of type *IdentifierNameSyntax* whose token value is the type name of the variable, see figure 3.4. Knowing the type name of a variable is not enough to determine if it is an instance of a known scenario type, as a software developer knows, it is possible to exist different types with the same name if the full qualifier name (in C# composed by assembly name, full namespace and class name) is not the same, furthermore, C# type inference allows to declare variables without explicitly specifying its type by using the *var* keyword.

In order to produce a compilation object, the compiler must determine the type of the variables, in compiler language, must know how to link the type name with the symbols of the symbols table. As mentioned in 3.2.1, *TestScenarios* project contains the code that generates the scenarios, so, through Roslyn's Symbol API, given a syntax node it is possible to know the expression's symbol and verify if it is a known scenario reference of that *TestScenarios*' assembly. The symbol type of a *VariableDeclarationSyntax* is done through the semantic model of the document as we can

Figure 3.4: Variable Declaration syntax tree outline

see in Listing 3.2. After obtaining the symbol it is possible to compare it with the known scenario symbols from *TestScenarios*, which implies that the known scenarios symbols have already been obtained.

```
1     /// <summary>
2         /// Gets the type of an IdentifierNameSyntax from SemanticModel if able and
              saves it in <see cref="t"/>
3         /// </summary>
4         /// <returns>true if <see cref="t"/> is set, false otherwise</returns>
5         public static bool GetTypeSymbol(SemanticModel s, IdentifierNameSyntax i,
              out ITypeSymbol t)
6         {
7             t = null;
8             TypeInfo info;
9             if (s != null && i != null && (info = s.GetTypeInfo(i)).Type != null)
10            {
11                t = info.Type.OriginalDefinition;
12                return true;
13            }
14            return false;
15        }
```

Listing 3.2: Obtaining symbol of *IdentifierNameSyntax*

3.3.3 Extracting tests' scenarios and their configurations

As demonstrated in 3.3.2 it is possible to determine through Roslyn's Symbol API which *VariableDeclarationSyntax* nodes are declarations of a Scenario. A named variable declaration has always a name (token) that identifies it and allows to identify where it is used in other expressions. A *VariableDeclarationSyntax* of a named variable always contains a child of type *VariableDeclaratorSyntax*, which in its turn always has a token child *IdentifierToken* that is the name of the declared variable. A *VariableDeclaratorSyntax* can also have an *EqualsValueClauseSyntax* that allows to immediately initialize the variable with either a LiteralExpressionSyntax such as a

string, numeric value, null, amongst others and also with an *ObjectCreationExpressionSyntax* to create an instance of the object matching type as we can see in figure 3.5.

Figure 3.5: Variable Declarator syntax tree outline with object creation

To determine the configuration of a scenario, it is taken into consideration the scenario instance initialization done with a *ObjectCreationExpressionSyntax* syntax nodes and the assignment to the scenario members done with *AssignmentExpressionSyntax*s' syntax nodes, each occurrence of this syntax nodes is then saved into a data structure that represents its configuration.

In an *ObjectCreationExpressionSyntax* it is desired to obtain the arguments in the expression if there are any. The arguments of an *ObjectCreationExpressionSyntax* descend from a list of arguments for that constructor and its type is *ArgumentListSyntax*.

An *AssignmentExpressionSyntax* can be separated in two main expressions, the left expression and the right expression. The left expression is placed before the equals token and contains the expression that indicates the member to whom the assignment value is being made, the member can be a simple identifier whose name value is previously set in a *VariableDeclaratorSyntax* syntax node, or can be a more complex expression such as a *MemberAccessExpressionSyntax* which represents the access to a field or property of an object, also, a *MemberAccessExpressionSyntax* can contain other *MemberAccessExpressionSyntax*, as shown in figure 3.6.

Figure 3.6: Assignment Expression with member access expressions

The right expression of an *AssignmentExpressionSyntax* contains the value given to the left expression, its syntax node can be for example *LiteralExpressionSyntax*, an identifier, a *MemberAccessExpressionSyntax*, a combination of the previous, an *ObjectCreationExpressionSyntax* and more. For each *AssignmentExpressionSyntax* whose left member is a scenario or a scenario member, the assignment is added to that scenario configuration data structure, but if the right expression is an *ObjectCreationExpressionSyntax* then it is added only this expression as its constructor.

For every scenario argument syntax node of its creation it is saved its value in the scenario configuration data structure, if an argument is composed by more that one syntax nodes then the resulting value is each syntax node value concatenated with a plus sign and the follwoing syntax node value. The same principle is applied for the right expressions of the scenario's *AssignmentExpressionSyntaxes*, but in this case it is also created a relationship between the scenario member access expression to whom the assignment is done and the result value of the right expression, without the scenario identifier as it will not be taken into account for the comparison between scenarios. It was identified a special case when one of the syntax node is an identifier of a scenario or the left most identifier of a *MemberAccessExpression*, better described in 3.3.3.1. It is important to mention that as the values are obtained from the syntax nodes directly they will not be influenced by trivia syntax such as comments or code indentation, a normalization guarantee.

3.3.3.1 Dependencies between scenarios and/or variables

An argument of a creation or one of the syntax nodes in the right expression of an assignment can be another scenario or other scenario property/member. This is important when comparing two scenarios of the same type between different tests because their configuration depend on other scenario configuration, meaning that even all other members are set equally but the scenario they depend on are set-up differently it should not be assumed that they are equivalent. In these cases, the value used to replace the syntax node dependencies on this scenario is the hash value of the scenario configuration that is also used to do a comparison between scenarios configuration of the same type. How to normalize the configuration and calculate the hash value is demonstrated in 3.3.4. An example of dependency between scenarios is shown in listing 3.3, where a step scenario member assignment depends on a flow structure scenario.

```
1 [TestMethod]
2 public void ResolveBomForMaterialWhereMaterialIsNotActiveTest()
3 {
4   // ...
5   FlowStructureScenario flowScenario = new FlowStructureScenario("F", "F:3");
6   StepScenario stepScenario = new StepScenario();
7   stepScenario.Entity = (flowScenario.Entity.FlowSteps[0].TargetEntity);
8   // ...
9 }
```

Listing 3.3: Dependencies on other objects

3.3.4 Scenario comparison

As stated before, scenarios are the generators of master data in this system and to know if the master data can be reused in different tests the scenarios of those tests must be compared. In this work, scenarios are compared by a hash value, a technique also used in some text approaches for code clone detection. The string used to generate the hash is composed by the meta-data name of the scenario type and then a concatenation of, first, the values of the arguments syntax nodes of its constructor ordered by location in the constructor and with their index as prefix, and in second, the assignment right expressions syntax nodes values with the left expression minus the scenario identifier as prefix.

With a hashing technique the order is important and so is data normalization. To guarantee the order of the members in the configuration string for hashing, primarily it is done a test analysis for each test in the solution, collecting all the assignment expressions and constructors by scenario type. After the analysis it is known the maximum number of arguments for each scenario constructor and every different left expression in their assignments, making it possible to establish an order for each scenario type when constructing the string that will be hashed and used as a configuration identifier, and the order for each member and argument is stored in a map with the identifier as key and the index as value. The calculated values of the argument and assignments syntax nodes are then stored in an array at the correspondent position. The string formation used to get the scenario configuration hash is partially shown in Listing 3.4.

```
1   /// <summary>
2  /// Sets the <see cref="EncodedValue"/> property by encoding the concatenation of
       its members values of EncodedValues if the members also contain scenarios.
3  /// </summary>
4  /// <returns></returns>
5  private string SetEncoding()
6  {
7      // ...
8      sb.Append($"symbol:{Symbol.MetadataName}");
9      if (MembersEncodedValues != null)
10     {
11         for (int i = 0; i < MembersEncodedValues.Length; i++)
12         {
13             var member = MemberIndex.Where(kv => kv.Value == i).First();
14             var memberName = member.Key;
15             if (MembersEncodedValues[i] == null)
16             {
17                 continue;
18             }
19             var concat = $"{memberName}:{MembersEncodedValues[i]}\n";
20             sb.Append(concat);
21         }
22     }
23     // ...
```

```
24 }
```

Listing 3.4: Dependencies on other objects

3.3.5 Tests comparison

A test set-up is a collection of scenarios used in a test method. One test set-up can be reused if its set-up is the same or part of a larger set-up in other tests. To compare tests in this work we get its scenarios from the set-up and look for them in other tests, the comparison result is given in percentage, the number of scenarios found in the test for comparison that exactly match the configuration of scenarios in the one used as base of comparison divided by the number of its scenario configuration, as in expression 3.1.

$$Resemblance = \frac{NumberOfMatchingScenariosConfigurationsinTest2}{NumberofScenariosConfigurationsinTest1} * 100 \quad (3.1)$$

$$(3.2)$$

If the resemblance value of a comparison between two tests is an exact match, 100%, then it is considered that the set-up of the test used as base of comparison is included in the set-up of the second test.

3.4 Development environment integration

3.4.1 Integration with the IDE

CMF's most used IDE is VS, and it is the platform used to write tests and develop the MES. Visual Studio was designed to load and execute its components through a Managed Extensibility Framework (MEF) and Microsoft provides an SDK (VSSDK) to extend the IDE also through MEF components. This tool was integrated with VS using the VSSDK to extend its functionalities for both usability and to allow CMF's developers to take advantage of the tool in their development environment.

3.4.2 Using the tool in Visual Studio

To use developed tool an user must install the *vsix* file that adds the new functionality into VS. The *vsix* file is obtained through the compilation of the software developed. With CMF's Tests Solution open in the IDE, upon a right click over a test method in the context menu that shows, the functionality will be available as shown in figure 3.7. Activating the tool will then open the Resemblance Tool Window where it will be shown the other tests in the solution with their name, document and project ordered by test set-up resemblance from higher to lower.

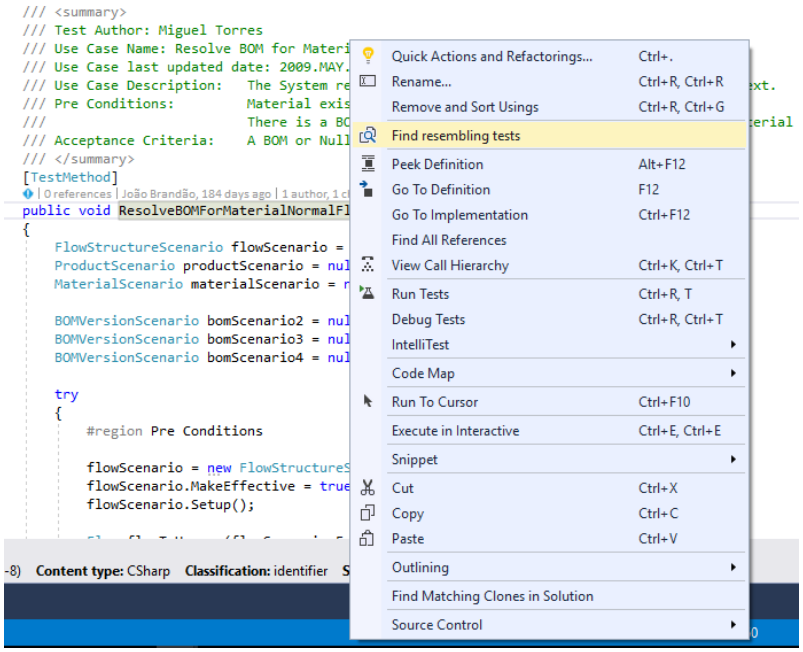

Figure 3.7: Resemblance Tool in Visual Studio

3.5 Results

The resemblance tool extension has proven to be a valuable decision support feature with the generated output, for instance, with a subset of 100 tests of the solution, it was found 9 tests who were exactly replicated (the full body of the test and not only the set-up) in different files after looking at the results and investigating manually the comparisons that had 100% match (meaning that the base test method set-up is contained in the other). In the CMF's tests solution the average time on a cold start to finish the comparison between the selected test and all the others was 47 seconds, but as long as the tests are not modified the scenarios configuration are kept in memory and the following comparisons take an average of 33 seconds to perform. Visual studio also has a feature find matching clones in solution, the option can also be seen in figure 3.7, but it obligates to select the code inside the method to look for clones, which can be an issue if the test set-up contains non relevant code to the set-up in between. Visual studio's find matching clone matching clones in solution feature in average took 4 minutes and 30 seconds to perform the code clone detection and revealed itself more susceptible to the order of the tests scenarios set-up, names and spacing. In figure 3.8 we can see that the visual studio native tool for clone detection also suggest the different parts of the same method as possible clones, that is because it also looks for partial clones which is not ideal for the same purpose that the tool developed in this work.

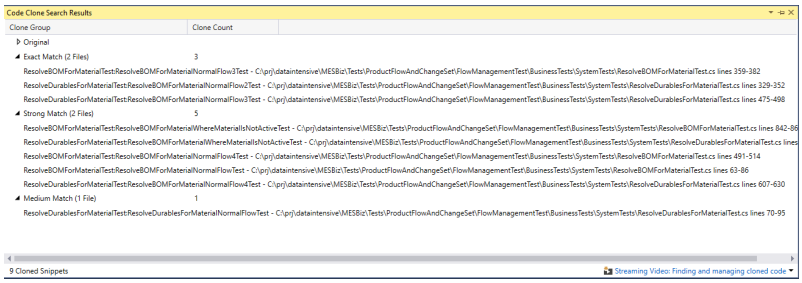

Figure 3.8: Visual Studio Find Matching Clones result

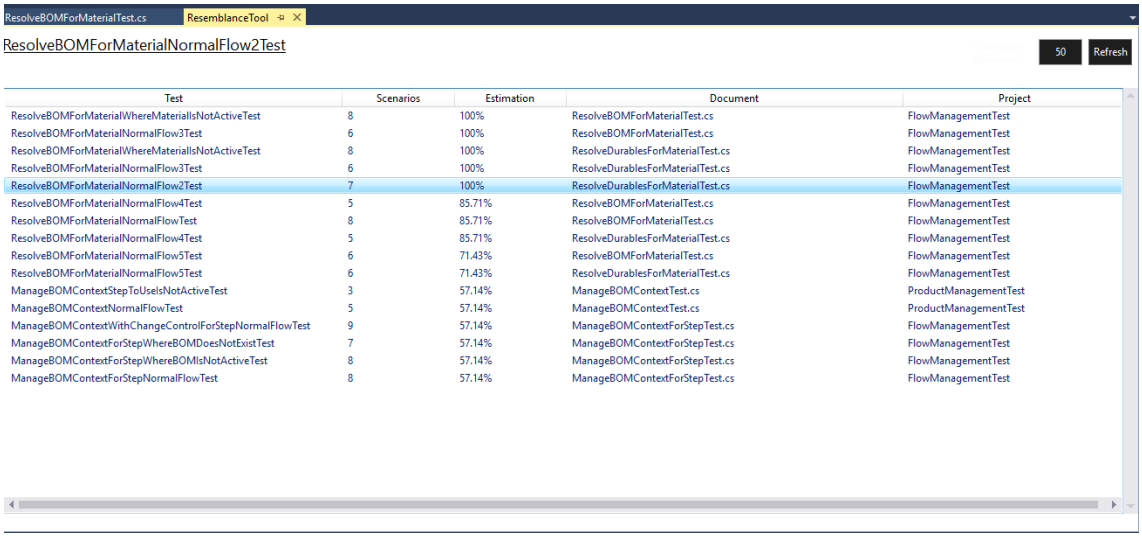

Figure 3.9: Test Resemblance Tool result window

Chapter 4

Data-driven transformation tool - From conventional to data-driven tests

4.1 Problem

Critical Manufacturing's tests solution was developed using conventional tests and has at the moment 3696 tests to verify the software. Has a data-intensive application with a necessary set-up to test the different functionalities of CMF's MES, there are cases where tests generate and need the same kind of data but with different values. With conventional designed tests it means that to fully test a functionality in the system considering variations according to the values of the data, the same test structure is repeated many times, and the code is repeated through the tests solution increasing the written code, thus harder to maintain.

One way to reduce code repetition in tests is using data-driven tests, a technique to reuse the test skeleton by loading data from disk to assign to variables of the test. Converting a test suite developed with conventional tests into data-driven tests would be very demanding and expensive. Besides reducing the lines of code by reusing it, data-driven tests allows people with no programming skills to create new tests by simply adding more data into the data provider for the test.

CMF's tests set-up are done using in-test specified parameters, using Roslyn to perform static code analysis, in this work it is developed a tool, "Data-Driven Transformation Tool" for Visual Studio that obtains those parameters and allows to convert a conventional test into a data-driven test.

4.2 Implementation

As seen in chapter 3, a test is a method declaration with an attribute "TestMethod" inside a class with the attribute "TestClass", the identification process of tests is done in the same way as in the Resemblance Tool.

A data driven test in the default Microsoft's unit test framework is defined with two more attributes besides "TestMethod" attribute, "DataSource" that provides data source-specific information for data-driven testings and "DeployItem" that specifies a file to be deployed allong with the assemblies before running a test. Visual Studio supports different file formats to link data to a test, such as sdf, xlsx, csv and others. In this work it is used a csv file to store the data extracted from a test due to its usage simplicity and how easy it is to add another line to create another test data row.

The Data-Driven Transformation Tool is a component for Visual Studio, it was developed using the "Analyzer with Code Fix" Visual Studio project template that ships with the Microsoft Code Analysis SDK and it is added to the IDE using the MEF that also loads the other IDE components. The project can be separated into two parts, the analyser and a code fix provider.

4.2.1 Analyser

The analyser is the responsible to verify if the code syntax node at the IDE's caret position matches some defined conditions and launch the appropriate code fix provider with the syntax node under analysis. In 4.1 it is possible to see the code used to verify if a the method declaration that it is under analysis is a test with the call *HasAttributeName(node, _testMethodAttribute)* and not yet converted into a data-driven test, the test must also contains literal syntax node to allow reporting the diagnostic to the code fix provider.

```
1     private void AnalyzeMethodDeclaration(SyntaxNodeAnalysisContext context)
2         {
3             var node = context.Node as MethodDeclarationSyntax;
4             if (!TestMethodCollector.HasAttributeName(node, _testMethodAttribute)
                  ||
5                 TestMethodCollector.HasAttributeName(node, _dataDrivenAtt))
6             {
7                 return;
8             }
9
10            if(node.DescendantNodes().OfType<LiteralExpressionSyntax>().Any())
11            {
12                var diagnostic = Diagnostic.Create(Rule, node.Identifier.
                      GetLocation(), node.Identifier.ValueText);
13                context.ReportDiagnostic(diagnostic);
14            }
15        }
```

Listing 4.1: Analyzer code to register code fix suggestion

4.2.2 Code fixer

The code fix provider is where code changes are purposed and shown to the user that has the option to accept or ignore them.

The conditions to convert a conventional test into a data driven are being a test method within a test class, and to contain *LiteralExpressionSyntax* syntax nodes, these contain the values that will be added to a csv data file and loaded at run time for the test execution. The supported *LiteralExpressionSyntax* support kinds implemented were the following:

- int32
- int64
- int16
- float (Single)
- Decimal
- bool
- string

When a code fix is available the suggestion is done in light bulb adornment, introduced in Visual Studio 2015. Hovering over the code fix suggestion a pop up with the preview of the code changes will be shown, those changes will only be applied when the user clicks the suggestion as it is in other native visual studio light bulb code fix suggestions.

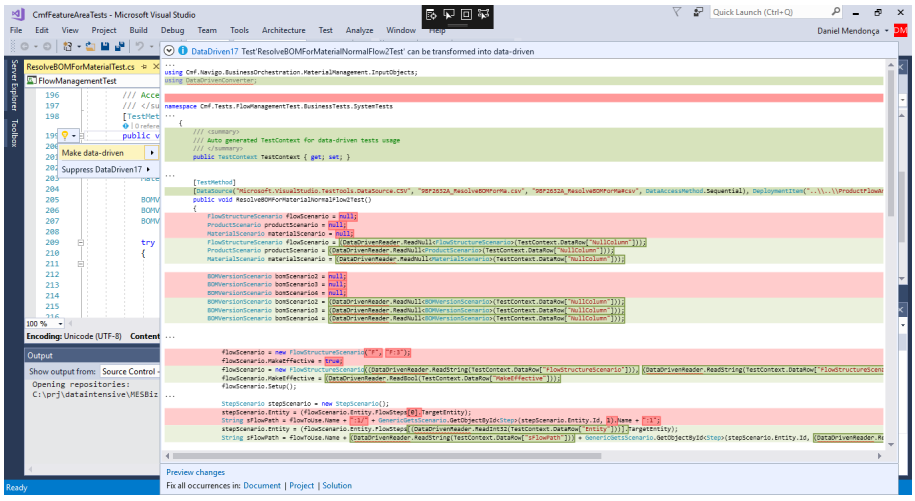

Figure 4.1: Light bulb data driven test conversion suggestion

Roslyn syntax tree are design to be immutable, meaning that it is not possible to change any syntax tree, nor part of it, this is what allows multiple users to work in the same solution at the same time without conflicts [Mic17]. The only way to change a syntax tree is to create a copy

of it with replaced, new or deleted syntax nodes and then replace the syntax tree itself with the new one. Roslyn Syntax Tree API includes classes to help the creation of new syntax trees. The *SyntaxFactory* is a static class used to create new syntax nodes that can then be added to a new syntax tree or occupy the position of node to be replaced in the new tree. Creating syntax nodes is a very verbose operation, listing 4.2 demonstrates the addition of a property to a class. The *DocumentEditor* class contains methods to generate a new syntax tree with the proposed changes, an instance of it can be obtained through an existing document. *DocumentEditor* keeps a queue of changes and returns a new document after processing those changes. The code fixer provider uses the document returned to replace the old document instead of modifying it.

4.2.2.1 Test Context

A data driven test to work in the Visual Studio test framework needs the test class to have the *TestContext* property. TestContext is responsible to link the data from the csv file specified in the method's deployment attribute with the test. To keep the conversion simpler to a user the conversion tool adds the property to the class if it does not exist yet, and reuses it if it already exists. This is done by taking all the properties declared inside the class through Roslyn's Syntax Tree API, get their types through the semantic model of the document using Roslyn's Symbol API and check if one matches with the known TestContext symbol, if it does the property token text value is taken and used in the conversion, otherwise the property is created and added to the beginning of the class block.

```
1   private string AddTestContextProperty(DocumentEditor editor,
        ClassDeclarationSyntax c, SemanticModel model)
2        {
3            var context = model.Compilation.GetTypeByMetadataName(
                 _contextMetadataName);
4            if (context == null)
5                return String.Empty;
6
7            string tcTokenText = TestContextExists(c, context, model);
8
9
10           if (tcTokenText != String.Empty)
11           {
12               return tcTokenText;
13           }
14
15           var typeName = SyntaxFactory.ParseTypeName(context.Name);
16           PropertyDeclarationSyntax property = SyntaxFactory.PropertyDeclaration(
                 typeName, _defaultContextName)
17               .AddModifiers(SyntaxFactory.Token(SyntaxKind.PublicKeyword));
18           property = property.AddAccessorListAccessors(
```

```
19                  SyntaxFactory.AccessorDeclaration(SyntaxKind.GetAccessorDeclaration
                        ).WithSemicolonToken(SyntaxFactory.Token(SyntaxKind.
                        SemicolonToken))
20                  );
21              property = property.AddAccessorListAccessors(
22                  SyntaxFactory.AccessorDeclaration(SyntaxKind.SetAccessorDeclaration
                        ).WithSemicolonToken(SyntaxFactory.Token(SyntaxKind.
                        SemicolonToken))
23                  );
24
25              property = AddTestContextDocumentation(property);
26
27              editor.InsertMembers(c, 0, new List<SyntaxNode>() { property });
28
29              return _defaultContextName;
30          }
```

Listing 4.2: Adding or reusing a test context

4.2.2.2 Replace Literal syntax nodes with Test context expressions

It is possible to obtain the literals of a test having its *MethodDeclarationSyntax* syntax node either by a *CSharpSyntaxWalker* extended class or through LINQ queries over the syntax node. To replace the literal node with a TestContext node, first it the literal must be given a name that will represent its column in the csv file. The naming of the literals are done taking into consideration the expression that contains them, for example, if the literal is in an *AssignmentExpressionSyntax* its column name in the csv will be the text value of the rightmost *IdentifierNameSyntax*, and if there is more than one literal in the same expression the following column names will have the occurrence number appended. The name of the column is obtained in an inside out approach, that is, the expression that will be used to give the column name is found by visiting the literal ascendant nodes until an assignment expression, invocation or variable declaration is found. An example of how the name is obtained through an assignment expression can be seen in listing 4.3. As for the value that will be used in the test, it is the literal syntax node text value that will be stored in the data rows of the csv for that column.

```
1           private StringBuilder GetColumnHeader(AssignmentExpressionSyntax
                expression, StringBuilder builder = null, string prefix = null)
2       {
3           if (expression == null)
4               return null;
5
6           var header = builder ?? new StringBuilder();
7           if (prefix != null)
8               header.Append(prefix);
9
```

```
10              var memberAccess = expression.Left.DescendantNodesAndSelf().OfType<
                    MemberAccessExpressionSyntax>().FirstOrDefault();
11              var member = DIAP.Helpers.MemberAccessExpressionHelper.
                    GetRightMostIdentifier(memberAccess);
12              if (member == null)
13              {
14                  var identifiers = expression.Left.DescendantNodesAndSelf().OfType<
                        IdentifierNameSyntax>();
15                  if (!identifiers.Any())
16                      return header;
17
18                  member = identifiers.First();
19              }
20
21              header.Append(member.Identifier.ValueText);
22
23              return header;
24          }
```

Listing 4.3: Get Assignment expression literal csv column name

The column names will be used as as the argument for the TestContext bracketed argument, and when the new syntax nodes are generated it is invoked the *ReplaceNode* method of *DocumentEditor* with the literal syntax node and the generated TestContext expression that will add the replacement to its queue. Microsoft CodeAnalysis SDK offers the possibility to add add additional documents to the visual studio project, and so, when the user accepts a test conversion suggestion by clicking in the light bulb option the document is added to the project and visible in Visual Studio's project explorer. Though the file is added to the project, when exploring the project additional documents through Roslyn's API the document is not visible, a bug found during this project to Microsoft.

4.2.2.3 Renaming Columns

The column names automatically generated for the csv data files sometimes are not the clearest and might force a person that wants to create a new data row for the same test to either look at the code or to read the tests specifications documents which. Taking this in consideration, for usability reasons it was added a code refactoring provider that allows to rename a column in the csv and update all its usages in the code. The analyser for this functionality is different than the data-driven transformation as the test must already be data-driven, and the IDE's caret position must be in the literal syntax node that represents the csv column name. Using a code snippet added to the tool an assignment expression is added to the test code whose value is an input given by the user, the input will be used to change the selected column name in the csv and its occurrences in the code, see figure 4.2.

Figure 4.2: Rename csv column

4.2.2.4 Removing and Adding Columns

A data-driven test might not need to have all its literal expressions in the csv file, some may be always the same value thus it does not makes sense to bloat the file with another column that only hinders the file readability and edition. When the IDE's caret is on top of a csv column name of a TestContext argument, there is another action added to Visual Studio's light bulb that offers the capability to remove the column from the csv and restore the column value present in the first csv data row, see 4.3.

Following the same principle it is also possible to add a column of a literal that was removed from the csv back to it, taking into account that if the file has more than one data row the value will be replicated throughout all data rows.

Figure 4.3: Remove csv column

4.2.3 Results

This functionality was validated by CMF's interns that knew little or nothing about Microsoft Test Framework and CMF's tests implementation.

The tool was considered useful in the internal demonstrations but was not used in production as it was developed for Visual Studio 2017, that will be adopted in the future but currently is still used Visual Studio 2015 for development. Nevertheless the interns could successfully convert a test into data-driven tests with just a click, create new tests by copying lines and modifying the column values in the csv.

Renaming, adding and removing columns functionality were considered to be more user friendly and less error prone than manually editing the csv files and the test code.

Chapter 5

Conclusions

In this project is presented an innovative approach into finding test clones regarding their configuration. Microsoft CodeAnalysis SDK provides developers rich code information, and it is easy to combine the information with Visual Studio SDK and create development tools that increase developers productivity.

Roslyn is an open source project and is being continuously updated and during this project it received one major update and several less significant. Roslyn is not fully documented but has an active community and its developers answer questions and give advices in public forums. The fact that it is open source allows to easily extend classes and look at the code to implement new features by looking at what already exists.

This dissertation had both a scientific component more relevant for academic purposes, and an enterprise component more relevant to Critical Manufacturing company that purposed this dissertation. In an academic perspective the most relevant contribution is the developed approach to detect tests configurations resemblances. In the enterprise perspective, both the integration of the developed tool with Visual Studio to suggest tests that can share the same master data, and the tool also integrated with Visual Studio that allows to convert conventional tests into data-driven tests can have a real impact in the testing process of the company.

5.1 Conclusions

5.1.1 Resemblance tool

The development of this decision support tool was considered successful, though some limitations have to be considered. The tool developed is able to explore all the tests in a visual studio solution, create the data structures necessary to perform test comparisons and do the comparison in about 44 seconds, that is only 15% of the mean time that the clone detection tool that comes with Visual Studio needs to analyse the solution and give the output of the possible code clones found. This tool is also unique, to the best of my knowledge, considering that its focus is the test set-up, the master data preparation for a data-intensive application test, abstracting the rest of the code that is not directly associated with data generation.

If indeed in the future tests' execution are involved in database transactions, there is no obvious reason that can impede tests to work with a constant database, using the same master data created with the help of the developed tool, thus reducing the time needed to run the tests. There is also no obvious reason that indicates incompatibility between the usage of the tool as a decision support tool to reach the constant master data and the usage of other optimization techniques for test suites such as database reduction techniques and prioritization.

During the development it was not considered the possibility of scenarios members invocations modifying the master data generation, though in the cases tested invocations had no influence over the master data outcome. Scenarios can have dependencies over other scenarios, in the case of a circular dependency detection a default value for the first scenario is given to the second to continue its configuration.

5.1.2 Data-driven tests conversion tool

The tool developed to transform conventional tests into data-driven tests follows the Visual Studio extensions guidelines and its behaviour is the expected as any other native Visual Studio extension. Using the tool to do the conversion is safer than doing it manually as it avoids common errors as misspelling file names, columns names, column values or even the csv file format. The main advantage of this tool is the conversion speed, what would take a developer minutes is done almost immediately after a click.

5.2 Future Work

During this project it was identified some aspects that could improve the developed tools, and work for future research.

- The column names generated by the conversion tool are not in most cases self describing for what the values represent, for example the naming of columns that were generated from literals that are arguments of an invocation or a creation expression could be named from the token text value in its declaration, this could be obtained exploring further Roslyn's Symbol API.

- In the test resemblances tool algorithm, when determining the value of a scenario member assignment, the right member expression sometimes an operation such as string concatenation, or numeric operations. Currently the value of the expression is the concatenation of each value, and hash value of a scenario if present. It is possible to know the scenario member type that has a value being assigned, which possibly is enough information to know how to handle the operations present in the right member thus possible to do those operations and evaluate the value that it will have in runtime to use in the configuration.

- The scenario comparison result in the resemblance tool is binary. This means that if one member is different between two scenarios they will be considered completely different,

regardless if that member has any relevance in the generated master data or not. It is believed that with enough scenarios, varying their configurations and analysing the resulting master data it would be possible to use machine learning techniques that classify the importance of their members on the resulting master data. If this can be verified and the members important classified, the comparison output could be done either by machine learning classification techniques or a continuous value with a threshold, and provide different results for test configurations comparisons, which could then be tested against the current approach to determine the best approach.

References

[Bei12] Inês Faro Beirão. *Mobile Graphical Interface Framework for Production Controlling and Monitoring*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, 2012.

[BFL+10] Romain Brixtel, Mathieu Fontaine, Boris Lesner, Cyril Bazin, and Romain Robbes. Language-independent clone detection applied to plagiarism detection. In *Proceedings - 10th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010*, pages 77–86, 2010.

[BG14] Maciej Bartoszuk and Marek Gagolewski. A Fuzzy R Code Similarity Detection Algorithm. *Communications in Computer and Information Science*, 444 CCIS(PART 3):21–30, 2014.

[BS10] Alba Berzosa and Javier Sedano. Integrating Manufacturing Execution and Business Management systems with soft computing. *Proceedings of the 10th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE*, (June), 2010.

[CA11] Laura M. Castro and Thomas Arts. Testing data consistency of data-intensive applications using quickcheck. *Electronic Notes in Theoretical Computer Science*, 271(1):41–62, 2011.

[CA13] Jose Campos and Rui Abreu. Leveraging a constraint solver for minimizing test suites. *Proceedings of the International Symposium on the Physical and Failure Analysis of Integrated Circuits, IPFA*, pages 253–259, 2013.

[Cos15] Jorge Costa. A Multi-Objective Approach to Test Suite Reduction. page 90, 2015.

[Cri17] Critical Manufacturing. Critical Manufacturing Framework, 2017.

[DGHZ11] Yingnong Dang, Song Ge, Ray Huang, and Dongmei Zhang. Code clone detection experience at microsoft. *Proceeding of the 5th international workshop on Software clones - IWSC '11*, page 63, 2011.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: Improving the Design of Existing Code. *Xtemp01*, pages 1–337, 1999.

[Hal12] Klaus Haller. Test data provisioning for database-driven applications. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6121 LNCS, pages 113–117, 2012.

REFERENCES

[JG07] Dennis Jeffrey and Neelam Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, 33(2):108–123, 2007.

[JH01] James A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *IEEE International Conference on Software Maintenance, ICSM*, pages 92–103, 2001.

[Mar17] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, Inc., 1 edition, 2017.

[Mic17] Microsoft. . NET Compiler Platform Overview, 2017.

[RCK09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[RGR16] Raúl H. Rosero, Omar S. Gómez, and Glen Rodríguez. 15 Years of Software Regression Testing Techniques - A Survey. *International Journal of Software Engineering & Knowledge Engineering*, 26(5):675–689, 2016.

[SZ12] Saeed Shafieian and Ying Zou. Comparison of Clone Detection Techniques - Technical Report 2012-593. *Research.Cs.Queensu.Ca*, 2012.

[VCT09] Vipindeep Vangala, Jacek Czerwonka, and Phani Talluri. Test case comparison and clustering using program profiles and static execution. *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - E*, page 293, 2009.

[YLW09] Qian Yang, J. Jenny Li, and David M. Weiss. A survey of coverage-based testing tools. In *Computer Journal*, volume 52, pages 589–597, 2009.