Analysis of Influence Produced by Code Changing

Maxim Kuzmin
PROTEI
Saint-Petersburg, Russia
maxriderg@gmail.com

Vitaly Pelin, Ilya Isaev, Ivan Perl
ITMO University
Saint-Petersburg, Russia
pelinvo@gmail.com, ivisaev@corp.ifmo.ru, ivan.perl@corp.ifmo.ru

***Abstract*—This article addresses estimation of source code changes influence on a high-scale software solutions. Described existing approaches address this issue only partially while the proposed solution is aiming to provide an ability of deep influence tracing from changed source code piece through the rest of the project. Such information can be used as a solid and reliable baseline for the risk management and estimation in cases of making code changes in the project or when changing version of used third party library with available source codes.**

I. Introduction

Modern software may consist of thousands lines of code and be developed by large teams. It may depend on number of side projects and libraries. Even if structure is simple, it may be hard to figure out how and where particular changes made in code or referenced library will affect the project.

Let's take a look at a typical situation when project depends or references another code written by colleagues or some third party library. At some point, new version has been released and this function was changed in such a manner:

```
public int computeValue(int input)
{
-    return input / 5;
+    return input / 10;
}
```

Suppose, developer didn't read patch notes or there was no note for this change. Build process passes with no new warnings or errors, all tests are successful. But program works incorrectly. What should a developer do in such situation? Highly likely, algorithm will be so:

1) Search for a defect in code.
2) Check old or write new tests.
3) Check source code of the library.

The reason of troubles is rather simple, but time spent on fixing may be enormously big, depending on a size of whole project.

But it is not the only problem related to libraries. At some moment, project receives new requirements which can't be met with the current configuration of imported libraries and to address this situation it is necessary to migrate to a newer version of used library, that will bring needed features. To decide, what solution to apply, developers need to evaluate, how complicated the migration will be. And this evaluation is not that simple.

Depth of library's influence can be viewed this way:

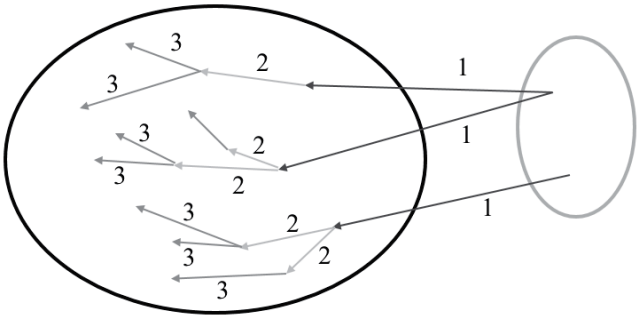

Fig. 1. Propagation of library influence

On a Fig. 1, left circle is a project, right one is a library API. Arrows, marked with "1", represent direct calls to library API - first level of influence, arrows with "2" represent calls of project methods, that call library API - second level of influence. Arrows with "3" are for methods, that are on third level.

It is easy to see, that influence of library can be not only wide, but deep too. To evaluate it without any special tools, experienced developers, who added this library to the project or hardworking and thoughtful developers, that will find all library usages and track its consequences, are needed.

II. Existing solutions

A. Code Review

Team members, chosen as reviewers, read code and leave comments. In some cases, reviewers focus on bugs only, but when team desire maximum quality, they also can seek for architectural defects, inappropriate tool usage and bad coding style. Code review helps to find some bugs on early stage and kick off bad solutions. The whole team works on one piece of code, making it better from points of view of many developers.

But code review has problem, mentioned in introduction - it's hard when it comes to merge branches with a large number of changes. All change influence propagation should be evaluated by developer himself basing on his experience in the project.

B. Diff

Diff [1] - utility to compare two files. It seeks for different strings and prints them with row and column numbers. Easy to

ISSN 2305-7254

use and very helpful when it comes to preparing or reviewing a commit.

It is a very universal utility, but this advantage means that it is low informative for particular language. It gives no information about change influence propagation because it doesn't use source code as semantic structure with special syntax.

C. Specialized source code analysis tools

Source code analysis tools are designed to analyze source code and/or compiled code versions to help in search of security vulnerabilities. Some tools can be plugged in IDE. These are powerful assistants in the development lifecycle, especially for problems that can be detected at early stage of software development, because they provide immediate feedback to developer. That is very useful, especially compared to finding vulnerabilities much later in the development cycle.

Graudit [2] is an open-source script and signature set, supporting .NET languages, that allows to find potential security flaws in source code using the GNU utility grep.

Puma Scan [3] is a software security analyzer that provides real time, continuous source code analysis for C# applications. Supporting different types of licenses, including Mozilla Public License 2.0. Can be built into an IDE and also used via CLI. With Puma Scan, vulnerabilities are displayed immediately in the development environment and appear as spell check and compiler warnings. Integrated Puma Scan security rules silently search for security vulnerabilities and alert if any are found. Security analyzers run on code files as the compiler parses syntax nodes, trees, symbols, code blocks, or semantic models. Identified vulnerabilities are tagged in the source code location by Visual Studio. Over 55 documented vulnerabilities to reference and common secure fixes for them. Rule categories include: Configuration, Cross-Site Scripting, Cryptography, Insecure Deserialization, Injection, Password Management and Validation.

Security Code Scan [4] - open-source project, that is a set of Roslyn analyzers that aim to help security audits on .NET applications.

Project features:

- the tool allows to detect 29 vulnerability patterns with 69 different signatures;
- taint analysis is the capability to track variables in the code flow and trace variable coming from user input, helps reducing false positives;
- code fixes are automated refactoring to fix vulnerabilities at the source;
- it can be integrated to any continuous integration that supports MSBuild.

SonarQube [5] - scans source code for more than 20 languages for issues, vulnerabilities, and code smells. SonarQube has plugins for Eclipse, Visual Studio and IntelliJ. Continuous Inspection, that can be found on project home page, shows where developer stands in terms of quality. This main page also shows an immediate sense of the good results achieved over time.

Tools analysis shows that none of the tools presented above directly solve stated problems. But they are addressing number of important issues applicable for various projects. These tools can in general be characterized in a following way:

Strengths:

- scales well - can be run on lots of software, and can be run repeatedly (as with nightly builds or continuous integration);
- useful for things that such tools can automatically find with high confidence, such as buffer overflows, SQL Injection Flaws, etc;
- output is good for developers - highlights the precise source files, line numbers, and even subsections of lines that are affected.

Weaknesses:

- many types of security vulnerabilities are very difficult to find automatically, such as authentication problems, access control issues, insecure use of cryptography, etc. The current state of the art only allows such tools to automatically find a relatively small percentage of application security flaws;
- high numbers of false positives;
- frequently can't find configuration issues, since they are not represented in the code;
- many of these tools have difficulties analyzing code that can't be compiled. Analysts frequently can't compile code because they don't have the right libraries, all the compilation instructions, all the code, etc.

III. Technology

Problems to be solved:

- inability to predict a complete list of consequences of merge of two feature-rich code branches;
- lack of reliable measurement of complexity and scale of influence of migration from one version of used third-party component to another;
- inability to define influence of particular change under code review for the whole project.

Described solutions do not address stated issues because they are not processing a code structure and take source code for analysis as is. To solve stated issues it is required to dive into source code structure with understanding of referencing between methods to extract changes and influences propagation graphs. On the basis of this it is clear, that some other methods of analysis are needed. It is known, that compilers build source code structure during compilation, so it looks reasonable to use same approaches. These approaches are:

1) Abstract Syntax Tree (AST)
2) Abstract Semantic Graph (ASG)

A. AST

AST is based on parse tree. Parse tree or parsing tree [6], also known as CST - Concrete Syntax Tree, is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.

Fig. 2 is an example of parse tree for expression ”5 * 1 + 9 - 8 / 4”:

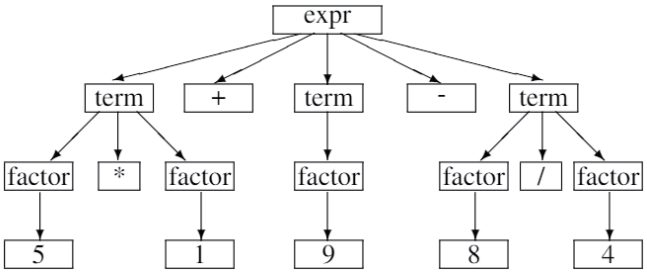

Fig. 2. Parse tree

In case of some operation written in programming language, parse tree for it will be full of syntactic details which are not very useful. These details are redundant because higher level of abstraction is needed.

AST [7], [8] is a tree representation of the hierarchical syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is ”abstract” in not representing every detail appearing in the real syntax. Applied to example, shown earlier, AST will look like:

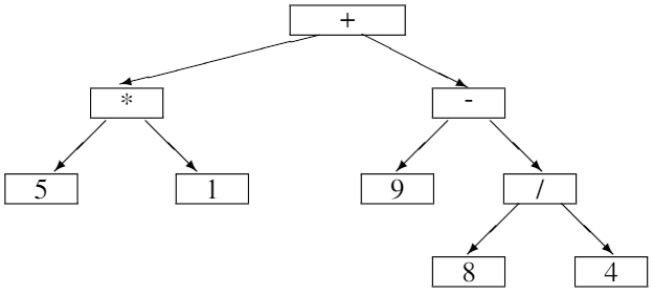

Fig. 3. AST

It is easy to notice, that AST is capturing the essence of the operation without useless syntactic details.

But let’s look on the AST of C# code:

```
namespace HelloWorld
{
  class Program
  {
    static void Main(string[] args)
    {
      Library inst = new Library();
      inst.Work(args);
    }
  }
}
```

AST for this code:

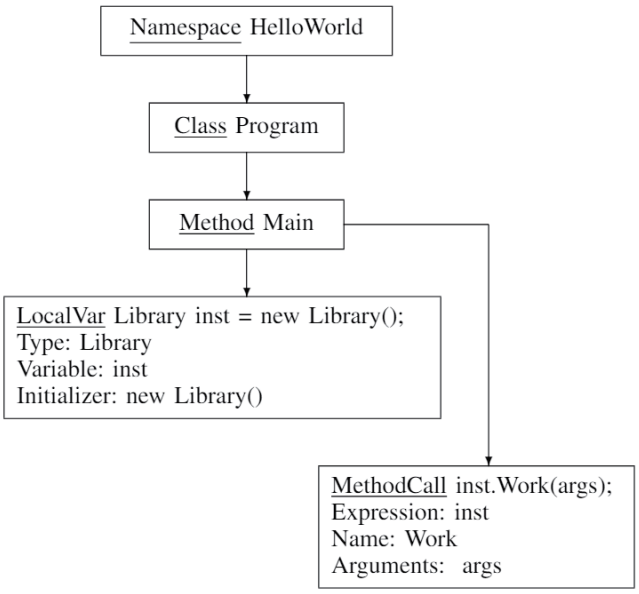

Fig. 4. AST for C#

Example is rather simplified, comparing to real AST, built by compilers or other special tools. But it is enough to notice, that any change applied to source code will change AST structure or particular node in it. AST structure reflects hierarchical structure of sources, when its nodes represent specific operators. So it’s obvious, that comparing two AST - one constructed from old version of code and another one from new version, can be much more informative, than comparing source by diff utility.

Initial version of code analysis tool that was implemented was based on AST of source code which were generated for the code. Soon, it was figured out, that application of AST analysis for the addressed issues was insufficient. It can be illustrated with example, shown in Fig. 4. There is ”MethodCall” node, that has parameter ”Expression” with ”inst” value, which is just a simple string - no reference, just a string. It is a serious disadvantage of AST, so few attempts were made to improve AST generation and processing, but this didn’t lead to the required capabilities. Finally, ASG was a solution, that covered processing gaps, that had place with AST.

B. ASG

ASG [9] is a form of abstract syntax in which an expression of a formal or programming language is represented by a graph whose vertices are the expression’s subterms. It can be described as AST with additional connections between nodes, which represent semantic of connected nodes.

ASG for previous example will look like this:

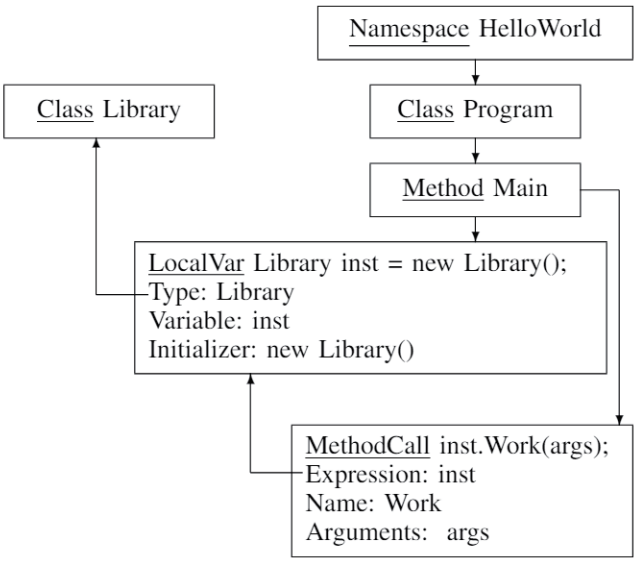

Fig. 5. ASG

New connections give knowledge, that Library in ”Library inst = new Library()” is a reference to class Library and ”inst” in ”inst.Work(args)” is an instance of Library class. With this, it is possible to track propagation of influence of changes in class Library and its method Work, if it will take place.

IV. DiffInfluenceAnalyzer

A. Idea

With AST and ASG it becomes possible to create an utility, that will solve problems, stated in previous section:

- predict a complete list of consequences of merge of two feature-rich code branches;
- provide reliable measurement of complexity and scale of influence of migration from one version of used third-party component to another;
- define influence of particular change under code review for the whole project.

B. Design

DiffInfluenceAnalyzer is based on tool named the .NET Compiler Platform (”Roslyn”) [10], because it is a powerful tool, that can parse code, build AST, extract semantics (as from ASG), etc. Since Roslyn is created and works with C#, the utility is written in this programming language. C# is modern, actively developing language and quite big number of large industrial projects is created with use of it. This gives DiffInfluenceAnalyzer a huge code base to test with and to analyze.

Main workflow of the utility:

1) Get two versions of code.
2) Create AST for both versions.
3) Compare these AST.
4) Find changed entities in new version with ASG.

DiffInfluenceAnalyzer works with two separate copies of project sources, representing two versions. These versions will be represented by C# solutions, because it is most convenient way to analyze sources using Roslyn. Both of them are supposed to be correct, because DiffInfluenceAnalyzer has no aim to replace the compiler.

C. AST example

AST, built by Roslyn from previously mentioned code sample, is demonstrated below:

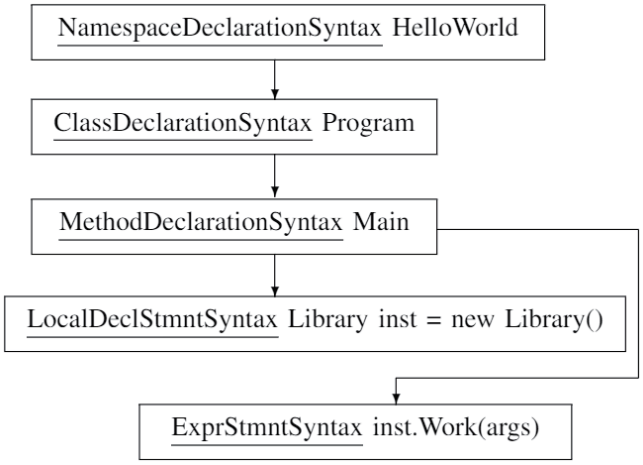

Fig. 6. Roslyn AST

For representation of any language entity declaration Roslyn provides special class. In previous example such classes can be found:

- NamespaceDeclarationSyntax - for namespace declaration;
- ClassDeclarationSyntax - for class declaration;
- MethodDeclarationSyntax - for method declaration;
- LocalDeclarationStatementSyntax (LocalDeclStmntSyntax) - for local variable declaration.

Also, there is class ExpressionStatementSyntax (ExprStmntSyntax), that represents expressions.

The scheme represents basic structure of AST, that’s why it lacks many details.

D. Solution comparison

Final goal of this step - find relevant pairs of entities from both versions, whether they are totally same or have some non-critical differences, that allow to consider them relevant. In other words, for each entity in old version it is mandatory to find its representation in new. To achieve this, AST will be compared this way:

1) Compare namespaces.
2) Compare classes.
3) Compare methods.

Entities from one version will be compared with entities of another on the same level. Comparison on each level consists from two parts - comparison of entity names and comparison of entity contents. Depending on level, comparison can return different results, but there are some common (for all levels) values:

1) Same - name and contents are same.
2) Renamed - same contents, but different names.
3) Reimplemented - same names, but different contents.
4) Different - nor name, nor contents are same.

E. Comparison of namespaces

Each namespace from old version is compared with each namespace from new version. Classes of namespace are assumed as its contents, so these classes are compared, to make conclusion about namespaces. After that, namespace names are compared. Result derivation algorithm:

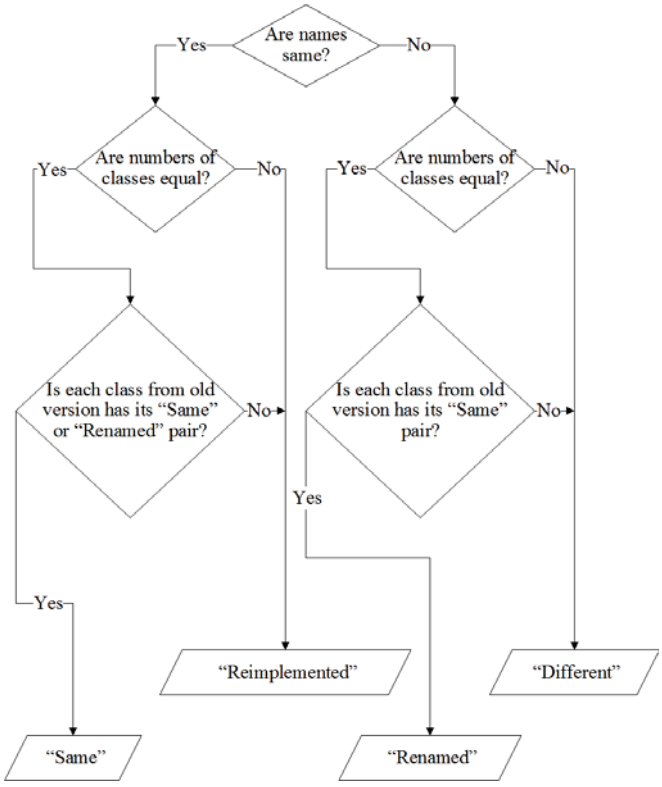

Fig. 7. Namespace comparison result derivation

If names are different and number of classes are equal and each class from old version has its ”Same” pair from new one, namespaces are considered ”Renamed”.

If names are same and number of classes are equal and each class from old version has its ”Same” or ”Renamed” pair from new one, namespaces are considered ”Same”. If not all classes are ”Same” or ”Renamed”, or some classes were deleted or added, then namespaces are considered ”Reimplemented”. In other cases namespaces are considered ”Different”.

F. Comparison of classes

Comparison of classes looks similarly as comparison of namespaces, exclude that content of class is its methods.

Each class from old version is compared with each class from new version. Methods of class are assumed as its contents, so these methods are compared, to make conclusion about classes. After that, class names are compared. Result derivation algorithm:

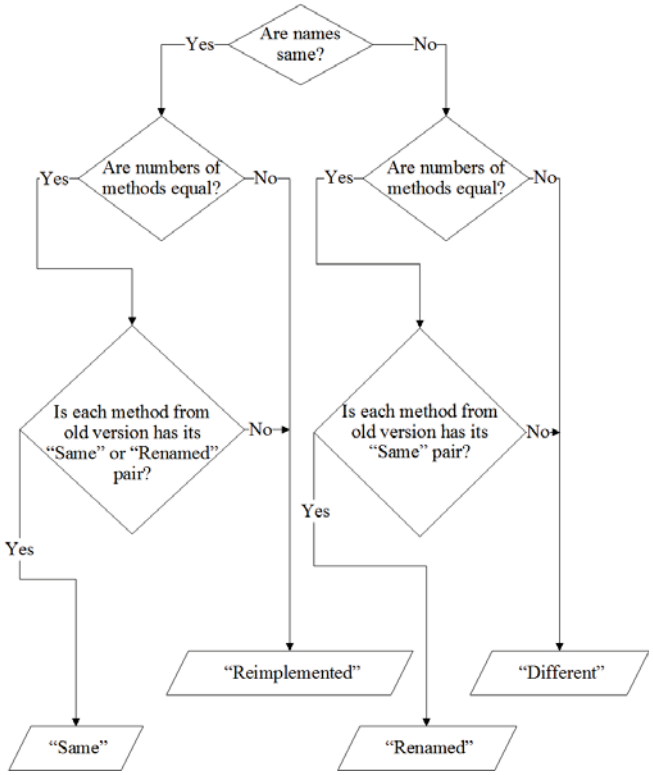

Fig. 8. Class comparison result derivation

If names are different and number of methods are equal and each method from old version has its ”Same” pair from new one, classes are considered ”Renamed”.

If names are same and number of methods are equal and each method from old version has its ”Same” or ”Renamed” pair from new one, classes are considered ”Same”. If not all methods are ”Same” or ”Renamed”, or some methods were deleted or added, then classes are considered ”Reimplemented”. In other cases classes are considered ”Different”.

G. Comparison of methods

Method content is formed of these parts:

- list of types of arguments;
- return type;
- body - number of operators and their values.

Methods have some specific comparison result, that is derived from their nature - ”Overloaded”.

Comparison result derivation algorithm:

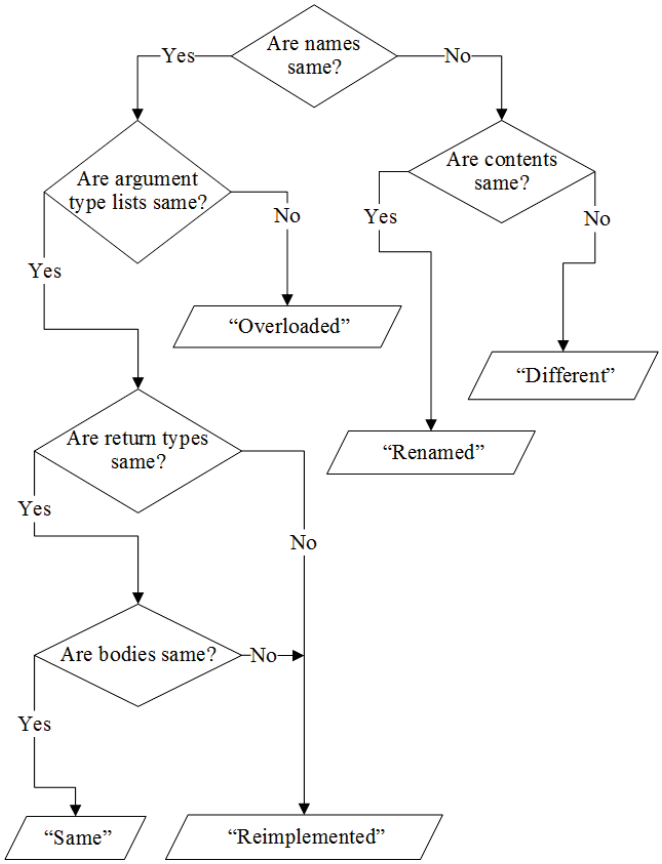

Fig. 9. Method comparison result derivation

If names are different, but contents are same - "Renamed". If at least one content part differ - "Different".

If names are same, but lists of types of arguments are different - "Overloaded". If return types or bodies are different - "Reimplemented". The essence of body comparison is string arrays matching.

In all other cases - "Same".

H. Search for changed entities in new version

All possible entities pairs are compared. Now, pairs that considered "Different" are dropped, because they are probably different originally and such cases aren't interesting for the utility. The interesting pairs are such that aren't considered "Same", because they are exactly those changes, that utility aims to find.

As far as signature of method is known, its definition in AST can be found. But as mentioned earlier, AST lacks of semantics, so calls of this method in the tree can't be found. For such a situation, Roslyn's SemanticModel [11] class, which provides functionality of ASG, is used. This class is created for each source code file and is used to find calls of chosen method. After search completion, its results are printed to command prompt.

I. Sample output

For now, DiffInfluenceAnalyzer output looks like this:

```
Changed methods:
HelloWorld.Program.Main(string[])
HelloWorld.Library.FooBar()

Searching for references on methods...

References on changed methods and influenced methods:

HelloWorld.Library.FooBar()
References:
C:\HelloWorld\Program.cs: (23,22)-(23,34)
C:\HelloWorld\Program.cs: (58,12)-(58,24)
Influenced:
HelloWorld.Program.Main(string[])
HelloWorld.Program.Compute(int)
```

From this output, user can make a conclusion, that methods HelloWorld.Program.Main(string[]) and HelloWorld.Library.FooBar() were changed. HelloWorld.Library.FooBar() was called in file Program.cs from row 23 columns 22-34 and from row 58 columns 12-24. Methods, that are standing on a zero level of change influence propagation are HelloWorld.Program.Main(string[]) and HelloWorld.Program.Compute(int).

V. Conclusion

In this article next problems were pointed out:

- inability to predict a complete list of consequences of merge of two feature-rich code branches;
- lack of reliable measurement of complexity and scale of influence of migration from one version of used third-party component to another;
- inability to define influence of particular change under code review for the whole project.

The study showed, that existing solutions solve these problems only partially, so it is reasonable to create a special utility. DiffInfluenceAnalyzer can find differences between two versions of code and track usages of changed methods. This functionality can help developer to define influence of particular change under code review and predict consequences of merge of two feature-rich code branches. Currently, improvements of the developed solution, that address issue with measurement of complexity and scale of influence of migration from one version of used third-party component to another, are almost finished.

References

[1] Linux documentation, Web: https://linux.die.net/man/1/diff.
[2] Github, Web: https://github.com/wireghoul/graudit.
[3] Puma Scan, Web: https://pumascan.com/.
[4] Github, Web: https://security-code-scan.github.io/.
[5] SonarQube, Web: https://www.sonarqube.org/.

[6] I. Chiswell and W. Hodges, *Mathematical Logic*. Oxford University Press, 2007, p. 38.

[7] A.V. Aho, M.S. Lam, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools, Second edition*. Pearson Education, 2007, p. 41.

[8] K. Cooper and L. Torczon, *Engineering: A Compiler, Second Edition*. Elsevier, 2012, p. 227.

[9] E. Duffy, "The Design & Implementation of an Abstract Semantic Graph for Statement-Level Dynamic Analysis of C++ Applications", *Clemson University. TigerPrints. All Dissertations. 832.*, Dec. 2011, p. 5.

[10] Github, Web: https://github.com/dotnet/roslyn.

[11] Docs Microsoft, Web: https://docs.microsoft.com/dotnet/api/microsoft.codeanalysis.semanticmodel.