Measuring Code Modernity of the C# Language Codebases

MARKS TROICINS, University of Twente, The Netherlands

This research addresses the problem of determining the modernity of software systems by analyzing the use of new language features and their adoption over time. The concept of modernity signatures is used throughout the research to represent the point in time that the codebase would have been written. This can provide valuable insights into the health of a codebase, the evolution of the codebase, and the evolution of the programming language itself. The modernity meter is developed to analyze different codebases involving C# running within the .NET runtime environment using static analysis methods. It will aid in calculating and analyzing the modernity signatures. The research describes the technical details of the modernity meter, analyzes the obtained modernity signatures, and concludes the study.

Additional Key Words and Phrases: Abstract Syntax Tree (AST), Code Modernity, Modernity Signature, C#, Static Analysis

1 INTRODUCTION

In the software engineering field, the concept of "modernity" can be defined as the extent to which the source code of a software system utilizes new features and capabilities of the programming language it is written in, as described in a study conducted by Admiraal et al. [7]. The formal definition of a modernity signature is provided in Section 5. It is important to investigate the modernity of a codebase as it can provide meaningful insights into its evolution and the evolution of the programming language itself.

First, the active use and adoption of new language features can serve as an indicator of the overall health of a codebase. Intuitively, actively maintained and updated code will also naturally use newer language features at least to some extent. Regularly maintained code is also more likely to be robust, secure, and scalable, while outdated and neglected code is more prone to bugs, security vulnerabilities, and performance issues according to Lehman [5]. In essence, lower modernity is indicative of the codebase moving towards becoming a legacy.

Second, analyzing the modernity of software systems can help understand how the system is advancing and evolving over time. By identifying trends in the adoption of new features across the codebases, we can gain insights into the practices of the developers and estimate how rapidly is the system evolving. For example, we can find and analyze bugs, and issues regarding the security of the codebase [4, 6].

Third, the investigation of modernity can aid in assessing the language evolution itself. It is insightful to analyze if the new features are adopted and used by developers as this can indicate if the language designers are investing their time wisely in incorporating these features. By examining the uptake of these features, it becomes possible to assess whether the efforts of language designers are aligned with the needs and preferences of the developer community.

This paper researches what analysis techniques are applicable to estimate the modernity of the C Sharp (C#) language [1], which is a widely-used programming language developed by Microsoft Corporation [2]. It is the fifth most popular programming language according to the TIOBE index [3] at the time of writing. By analyzing the adoption of new language features and best practices over time, we aim to provide insights into the evolution of C# codebases and the effectiveness of modern coding practices. While analysis platforms for C# already exist, the topic of modernity in the context of C# has been an under-researched field with many open gaps left, which are going to be partly addressed in this paper.

1.1 C# Language Levels

C# continues to evolve, solidifying its place as a powerful and versatile programming language in the developer community. As of November 2023, the latest release, C# 12.0, brings a host of significant enhancements to the language. Among the most notable are collection expressions[1] and primary constructors[2], which introduce new, expressive ways for developers to write and organize their code.

In addition to these syntactic advancements, C# frequently incorporates features that enrich the language without altering its grammar. For instance, C# 11.0 introduced UTF-8 string literals, enabling automatic encoding of string literals into their UTF-8 byte representations. Such features enhance the language's functionality while maintaining compatibility with existing code.

Every codebase that utilizes C# can be assessed based on its language level, defined as the minimum C# version required for the code to compile and run correctly. For example, as of May 9, 2024, the ShareX[3] codebase necessitates C# version 10.0 or higher. Modern .sln[4] (solution) files encapsulate this information, detailing the minimum required C# version and .NET[5] framework, ensuring that all dependencies and versions are correctly identified and managed, but does not evaluate the modernity of the codebase which leaves it an open topic.

1.2 Static Code Analysis

Code examination without the execution of it is called static code analysis [10]. This technique has a lot of usage, for example, it can be used to find bugs in codebases [4] or to analyze how secure they are [6]. Static analysis is used in this research to analyze the codebases,

TScIT 41, July 5, 2024, Enschede, The Netherlands

[1] https://github.com/dotnet/csharplang/blob/main/proposals/csharp-12.0/collection-expressions.md
[2] https://github.com/dotnet/csharplang/blob/main/proposals/csharp-12.0/primary-constructors.md
[3] https://github.com/ShareX/ShareX
[4] https://learn.microsoft.com/en-us/visualstudio/extensibility/internals/solution-dot-sln-file?view=vs-2022
[5] .NET is a secure, reliable, and high-performance application platform. https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet

thus the definition of the modernity signature and the implementation of it is built around it. There already exists an open-source implementation of the C# compiler with an API surface for building code analysis tools which is called Roslyn[6]. The implementation provided by .NET is used as a foundation for the implementation of the modernity meter for this research.

Most of the existing C# analysis software that can be found open-source or that are described in scientific articles have a strong focus on standards enforcement and security. The standards enforcement tools warn the user of problems in a codebase like formatting errors or code smells, and some of them might even propose a solution or edit the code autonomously. Security-focused tools either work similarly or apply patches or extra logic to code if it is being executed. All these tools have in common that they rely on pattern matching to find parts that need attention, which is achieved with static analysis.

2 PROBLEM STATEMENT

This paper aims to define the modernity signature that can be calculated with the static analysis. The calculated modernity signature is analyzed to determine if the static analysis is an appropriate technique to evaluate the modernity of a codebase. More precisely, we will answer the following research question:

To what extent can we use static analysis methods to reliably determine the modernity of the codebases developed in the C# language?

To aid in answering the research question, the following subquestions are answered:

RQ1 How can we define a modernity signature of a C# codebase?

RQ2 What factors influence the modernity signature defined in ***RQ1***?

RQ3 To what extent can the static modernity signature of an unknown C# codebase be used to represent the point in time that the code would have been written?

3 RELATED WORK

Deep research on code modernity was conducted by Admiraal et al. [7]. This study introduces the concept of modernity signature and formally defines it. The modernity signature is calculated for codebases developed in PHP and Python languages, thus the approach is slightly different for them. The paper also discusses different normalization techniques that can be applied while calculating the modernity signatures. Van den Brink et al. researched the derivation of modernity signatures for PHP systems with static analysis, which served as an inspiration for this research in some ways [13].

Static analysis of C# codebases has been performed before, for example by Koshelev et al. [9]. This article considers various aspects of static analysis of C# programs in order to detect the maximum number of software bugs in an acceptable time. The paper also discusses some methods that take into account popular features of C# at all levels of analysis. Similar research was conducted by Sharma et al. where they explored code smells in open-source C# repositories [12], which are indicators of quality problems that make software hard to maintain and evolve. Another study conducted by Shaukat et al. researched and compared eight different existing C# static analysis tools within one software system [11].

4 METHODOLOGY

The modernity signature will use statistics about the usage of specific syntax patterns in the abstract syntax tree (AST) of the selected codebase to derive its modernity. For example, the following grammar rule was introduced as a replacement for the `multiplicative_expression`, which added a rule for the `range_expression`:

```
multiplicative_expression :
    range_expression
    | multiplicative_expression '*'
        range_expression
    | multiplicative_expression '/'
        range_expression
    | multiplicative_expression '%'
        range_expression ;
```

Range expressions were added in version 8.0 of the C# language, which also affected the grammar rule of the `multiplicative_expression`, where `unary_expression` was replaced by `range_expression`. That means, if we detect the presence of `range_expression` in an AST of the codebase, we can conclude that it requires at least C# version 8.0 to run it. Then this process of associating language features and a language version its introduced in, and its detection in the AST has to be done for all the grammar rule changes throughout the evolution of the language. The occurrence amount of each detected version-specific feature is saved in a defined data structure, which represents the modernity signature for a selected codebase.

4.1 Developing and Calculating a Modernity Signature

The Roslyn analysis platform can provide an Abstract Syntax Tree representation of the C# codebase, which then can be traversed with the interpreter pattern. The interpreter compares each node to the specific syntax pattern and check for other conditions to determine if it is associated with any of the documented features. Each occurred version-specific feature then should be written to a C# dictionary[7], that stores language versions as keys, with the corresponding values representing the number of features specific to each version. This is how it is defined with C# language:

```
Dictionary<double, int> modernitySignature =
    new Dictionary<double, int>();
```

This definition is very specific to the C# language and needs a broader mathematical definition so that it is understandable. This dictionary can be described as a set of tuples, where the tuple stores the language version and the amount of version-specific features detected in the codebase. The dictionary holds the modernity signature for a codebase at a specific version, which is then added to a list with the same dictionaries but for different versions of the repository. Then this list defines the modernity signature of the repository. A formal mathematical definition is provided in Section 5.1 based on this approach.

[6]https://github.com/dotnet/roslyn

[7]https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-8.0

4.2 Testing the Modernity Signature

Testing throughout the development of the modernity signature is a crucial part of the process. There is no room for testing in terms of the definition of the modernity signature itself, but a lot of testing should be done within the implementation, to check if the documented features are recognized correctly. The testing methodology that is used throughout the development of the modernity signature is unit testing. By applying this testing strategy, the recognition of each documented feature was tested in isolation. For instance, one of the test cases to check if the extension of partial methods[8] is recognized correctly is:

```
partial class MyClass
{
  partial void PartialMethod();
}
partial class MyClass
{
  partial void PartialMethod()
  {
    Console.WriteLine("test");
  }
}
```

By applying this testing strategy, the recognition of each documented feature was tested with one to three different test cases. By testing each documented feature with several edge cases, we reduced the amount of false detects and false positives, which is the biggest concern for the modernity meter, as it should be as accurate as possible to aid in answering the research question.

4.3 Collecting Codebases

When the modernity meter is developed and tested, it needs to be applied to real existing projects, to calculate the modernity signatures for them. Then, the modernity signatures can be examined to determine if it is possible to represent the point in time at which the codebase would have been written. GitHub [14] is used to find relevant repositories for the calculation. There are some requirements and reasons for them, that a repository should meet in order to be analyzed within the research:

I The main language used across the repository should be C#. That means that at least 85 percent of the codebase should be written in the C# language.
Reasoning: The tool can only analyze the C# code.
II The repository should contain an .sln file with a consistent path to it.
Reasoning: The current implementation can find a .sln file only in the main repository's directory, without an .sln file it is unable to load the C# code.
III The repository should have a commit history dating back to its creation, with at least 300 commits in total.
Reasoning: There should be a reasonable amount of commits spread throughout the evolution of the repository so that it can be analyzed at different points in time.

[8]https://github.com/dotnet/csharplang/blob/main/proposals/csharp-9.0/extending-partial-methods.md

Various C# codebases were found that satisfy the stated requirements. The codebases were selected from the 50 most starred public repositories using C#. Table 1 presents the selected repositories for the calculation of the modernity signature. There are repositories that met the conditions but were not selected. The reasons for this are discussed in Section 7.3.

Table 1. Open source repositories selected for modernity signature calculation

Project	Link to repository	Versions analyzed	Time frame of versions
Files	Link	93	10/02/2019 - 13/05/2024
Clean Architecture	Link	65	08/08/2017 - 10/04/2024
.NET Aspire	Link	55	09/01/2024 - 13/06/2024
ShareX	Link	31	23/02/2017 - 09/05/2024
SteamTools	Link	29	21/07/2022 - 27/04/2024
MQTTnet	Link	17	08/05/2022 - 23/05/2024

5 SIGNATURE DEFINITION

5.1 Formal technical definition

A modernity signature provides a quantitative measure of the extent to which a codebase utilizes features from different versions of its programming language. Formally, we define the modernity signature of a codebase as a list M of length N,

$$M = [C_1, C_2, \ldots, C_N],$$

where each element C_i corresponds to a set, which is defined as:

$$C_i = \{(V_j, f(V_j)) \mid V_j \in V\},$$

at codebase's version i. Here V is the set of documented language versions and f is a function mapping each version to the number of versions-specific features. An example of C_i in a JSON[9] format:

```
{
  "7.1": 3036,
  "7.2": 49,
  "7.3": 1880,
  "8.0": 564,
  "9.0": 683,
  "10.0": 200,
  "11.0": 38,
  "12.0": 948
}
```

To construct the modernity signature, we first document the features introduced in each version of the C# language. There are currently 16 different versions of the C# language according to documentation provided by Microsoft[10]. Then we parse the selected

[9]https://www.json.org/json-en.html
[10]https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history

codebase to a syntax tree to identify instances of these features, where the amount of the pulled versions of the selected codebase is represented by N.

Each documented feature F is associated with the minimum version V_{min} that supports it. This means that the presence of feature F in the selected codebase directly contributes to the calculation of the static modernity signature by populating the set C_i and possibly incrementing minimal language version to run the codebase at version i, meaning that the codebase requires at least language version V_{min} or higher to run.

Finally, the overall modernity signature M is a list that stores all the minimum versions required to support detected instances of documented features F in the codebase.

It is important to note, that the research does not consider the impact of the different versions of different variations of the .NET and only analyzes documented features that were introduced in different versions of the C# language. Proper documentation of version features exists only for versions 7.1 and higher, so the research omits the versions before 7.1.

5.2 Implementation

The implementation and the testing of it is based on the methodology presented in Sections 4.1 and 4.2. We reference the implementation as a "C# Codebase Modernity Meter" or just a "Modernity Meter".

The first implementation was developed to work with test cases and was not capable of calculating the modernity signature for the whole codebases (repositories). The modernity meter generated an AST from the provided code snippet, and then Depth-First traversed it within the interpreter pattern. Each node is then compared with different syntax patterns and checked for other conditions if such are needed to determine if this is a version-specific feature. For example, this is a simple condition to recognize list patterns that were introduced in the C# version 11.0:

```
if (node is ListPatternSyntax)
{
  data.modernitySignature[11.0] += 1;
}
```

There also exist more complex features for recognition, such as records with sealed base ToString override, detection of this feature is done with these conditions:

```
if (node is RecordDeclarationSyntax
    recordDeclaration)
{
  foreach (var member in
      recordDeclaration.Members)
  {
    if (member is MethodDeclarationSyntax m &&
      m.Identifier.Text == "ToString" &&
      m.Modifiers.Any(SyntaxKind.SealedKeyword) &&
      m.Modifiers.Any(SyntaxKind.OverrideKeyword))
    {
      data.modernitySignature[10.0] += 1;
    }
  }
}
```

Once the first version of the modernity meter was tested and finished, it had to be adjusted to calculate the modernity signature for the repositories. To achieve that, the modernity meter uses the capacities of Roslyn, which can create a virtual workspace for the codebase from an .sln file. Once it opens the solution in the virtual workspace, it provides a list of all classes written in C#. For each class, the modernity signature is calculated and then combined for all the classes. The modernity signature is then saved in a JSON format. At this point, the signature is still calculated only for one version of the repository, which is insufficient to meet the formal definition. The implementation was adjusted with the possibility of analyzing the repository after finding a commit with tags “v”, “version” or “release”. If the repository lacks tags on the commits, every n-th commit is analyzed until the repository reaches the state where its structure is unprocessable by the modernity meter. At this point, we have collected modernity signatures of each analyzed version of the repository. To analyze the data in an understandable and presentable manner, it is Max normalized according to the research conducted by Zubcu [15] and plotted.

The repository containing the source code of the C# Codebase Modernity Meter developed by the researcher can be found by accessing this link: https://github.com/MarkOneLove/Modernity-Analyzer.

6 RESULTS

The developed modernity meter calculated modernity signatures for the repositories specified in Table 1. Calculated Max normalized modernity signatures can be observed in Figures 1 - 6. It is important to mention, that all the modernity signatures were calculated with the same final version of the modernity meter discussed in the second part of Section 5.2. Next, we analyze the calculated modernity signature for each selected repository and interpret the results.

Figure 1. Active use of documented version-specific features began only after 26/08/2020, which might indicate that the codebase did not have a lot of source code or it consisted of features from versions below 7.1. For most of the versions, the use of the associated features grows towards 28/01/2024, which is the date of the latest version. In the final version, the most used features are from versions 7.1, 7.3, and 12.0.

Figure 2. Features from version 8 were used the most throughout the observed history of the repository. We can also observe a decrease of these features from the peak towards 30/11/2017 and growth towards the present time. In the final version, the most used features are from versions 8 and 7.3.

Figure 3. Features from versions 7.1 and 7.3 were used the most throughout the history of the repository with inconsistent growth and decrease. We can also observe active usage of features from version 12.0 after the release of the language version. This rapid growth might indicate that a big part of the codebase was refactored and adopted the new capacities of the

language.

Figure 4. Only features from versions 7.1, 7.2, and 7.3 were used until 08/01/2022, after which features from version 10.0 were adopted and mostly used. This indicates that the codebase might have gone through a big release that adopted practices of that time.

Figure 5. Consistent use of features from version 7.1 and 12.0 (after the release of the version) can be observed. We can also observe the rapid growth and the outstanding amount of features used from version 10.0 after 12/02/2023, which can signal about a refactoring of the codebase as version 10.0 was released way before 12/02/2023.

Figure 6. The consistent growth of feature usage from versions 7.1, 7.3, 9.0, 10.0, and 11.0 can be observed throughout the analyzed time frame. This indicates that the repository might have been expanded bit by bit with code that followed the practices of the previous repository's versions, which makes sense as only 2024 and a small part of 2023 years were covered. There are some decreases in the usage of features, but generally, they grow, which might mean that sometimes the code was refactored, but only small parts of it.

All in all, the obtained results look almost as expected and provide room for analysis and conclusions, which are drawn in the related section.

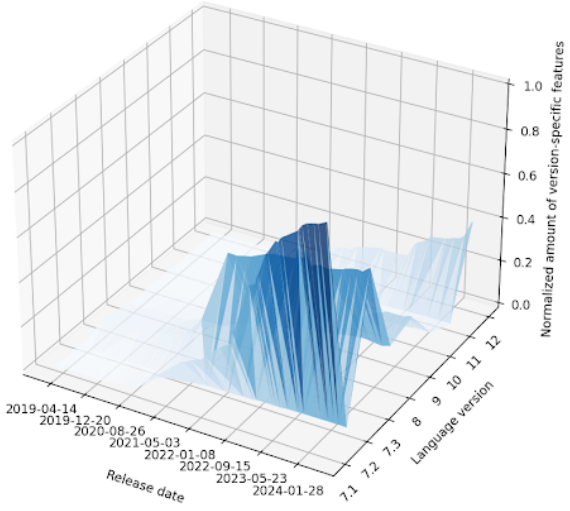

Fig. 1. Files Max Normalized Modernity Signature

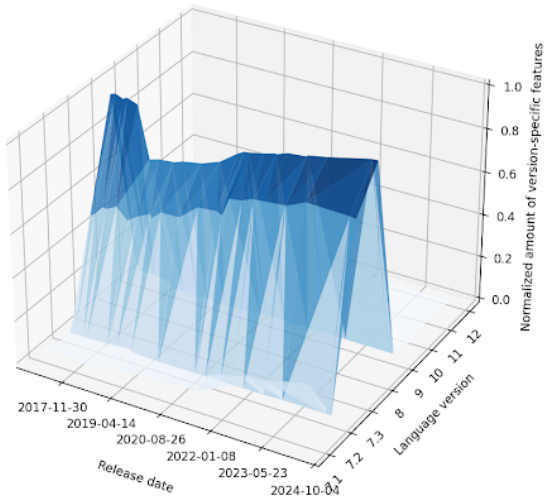

Fig. 2. ShareX Max Normalized Modernity Signature

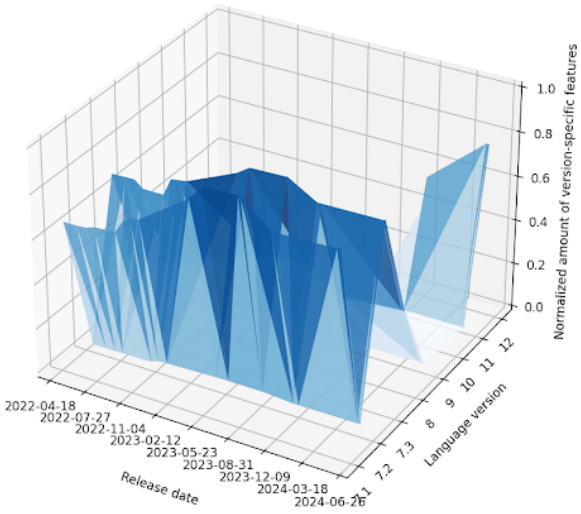

Fig. 3. MQTTnet Max Normalized Modernity Signature

7 DISCUSSION

This section discusses what factors influence the modernity signature and is important to understand how the definition of it was developed. We also discuss how the calculated modernity signature represents the point in time when the codebase would have been written. Another important topic that is discussed is the nuances of the selection of the repositories for the signature calculation and why only six of them were selected.

7.1 Factors Influencing the Modernity Signature

The definition and the implementation of the modernity signature were not developed immediately and have gone through a lot of research and refactoring. One of the concerns regarding the modernity signature is what factors influence it. The most obvious answer is the evolution and the features of the language itself, but in the context of C# it is a bit more complicated, as it is dependent on the runtime environment, which has its own versions. The first idea was to change the definition to a 2-dimensional list and store the versions of the runtime environment in it, but this idea was quickly discarded for several reasons. First, there are no features that can be detected from the AST that would be specific to the .NET version, they all are specific to the C#, but not the environment. Second, there are conceptually different variations of the runtime environment, such as .NET Framework, .NET Core, .NET Standard, and .NET, which makes it quite hard to linearly order them and only makes the signature more complex for analysis. Third, it would not extend the signature with useful data for analysis, as it would just be a version associated with a C# version, which is not informative. Conclusions about the minimum required runtime environment version can be drawn based on the minimal required version of the C# language, but this is irrelevant to this research.

7.2 Codebase Development Point in Time Representation

By analyzing the obtained modernity signatures, we can represent the trends of the version-specific feature usage and make some assumptions about the evolution of the codebase like it was done in the section with results. Based on the observed trends, we can estimate at what point in time and during which time frame was the repository mostly developed and extended with code. With the current implementation, it is not sufficient enough to estimate the age of the codebase, as the modernity meter fails to scrape the oldest versions of the repository and correctly process them. Generally, it is possible to represent the relative point in time that the code would have been written by observing feature usage trends, peaks, and skewness from the calculated modernity signatures.

7.3 Open-Source Repository Selection for Signature Calculation

There were many repositories that satisfied the requirements stated in Section 4.3, but only the ones mentioned in Table 1 were actually used for the modernity signature calculation. There are several reasons why a lot of open-source repositories were unable to be processed by the modernity meter.

First, in many of the repositories, an .sln file was either not present or was located in different directories throughout the history of the repository. For this reason, it was not possible to calculate the modernity signature for the whole history of the repository or for the solid time frame, which is at least one year.

Second, some of the repositories have a lot of source code and the calculation of the modernity signature can take a lot of hours or even days. For example, it took more than two hours to calculate the signature for one version of the Roslyn repository using a personal computing system with an eight-core processor and 32 GB of RAM. Because of this some of the repositories were skipped out, as there was not enough time to generate a modernity signature, which would be sufficient for the observation and analysis.

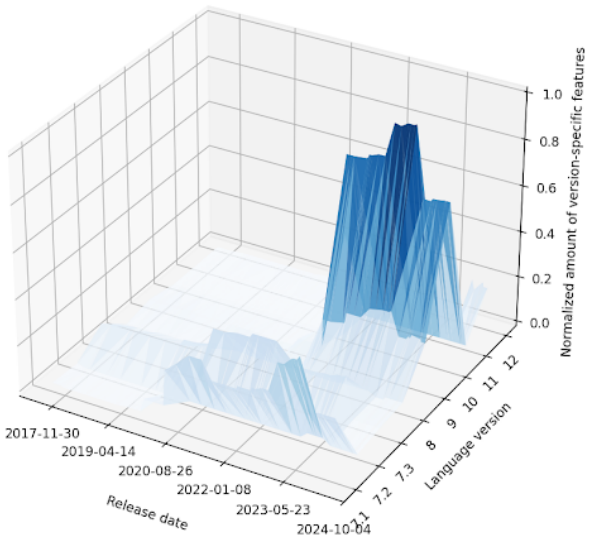

Fig. 4. CleanArchitecture Max Normalized Modernity Signature

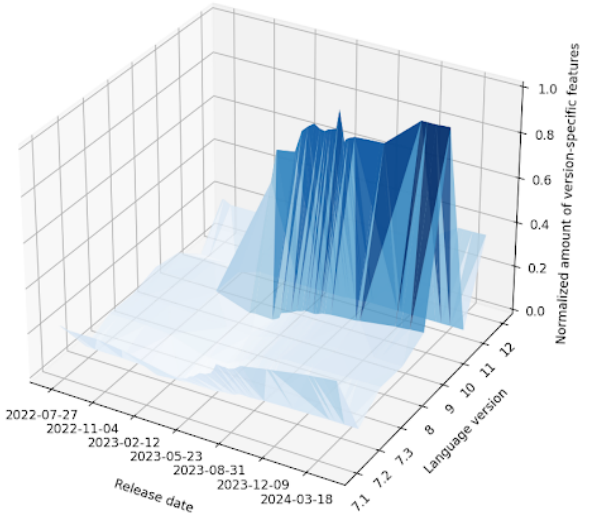

Fig. 5. SteamTools Max Normalized Modernity Signature

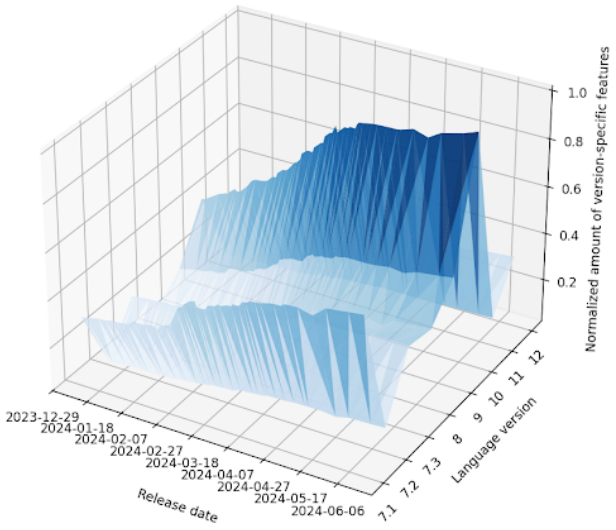

Fig. 6. Aspire Max Normalized Modernity Signature

8 CONCLUSION

8.1 C# Modernity Calculator Conclusion

The modernity meter itself is a proof of concept, thus it is not perfect and can be improved in different ways. Here we mention some of the improvements that we feel can make the tool better.

First, the modernity meter currently makes type I and type II errors, which stand for false positives and false negatives. For example, it can detect the presence of features from the latest version at the time of the codebase when that version was not released yet, or it can skip out the presence of some version-specific features. For this reason, the obtained results are not ideally accurate.

Second, the modernity meter can only recognize features starting from C# language version 7.1 and omits the existence of all features before that version. This is important and needs to be highlighted. This is because we could not find proper documentation of features from the versions below 7.1.

Third, the structure of the implementation can be refactored from the interpreter to the visitor pattern to achieve more efficient traversal and skip out syntax nodes that are not associated with different versions of the C# language. It will guarantee that nodes are visited only once and will provide more accurate results. The visitor pattern also makes code more readable as each visit method would be associated with one or few features which is much more convenient than a list of conditional statements within one or few methods. This improvement can be done to achieve better efficiency, which was not a requirement for the modernity meter.

8.2 Research conclusion

In this section, we answer the set of sub-research questions created to aid in answering the research question itself.

We have answered ***RQ1*** in Section 5. Both the formal mathematical definition and the details of the implementation of it were explained.

Next, we answer ***RQ2*** in Section 7.1. The factor that influences the modernity signature the most is the evolution of the C# language itself, another factor, that could have an influence but was omitted by the research, is the runtime environment.

Finally, ***RQ3*** is answered in Section 7.2. It is possible to represent a relative point in time when the codebase would have been written by observing peaks, skewness, increases, and decreases of the calculated modernity signature.

To conclude, static analysis can be used to determine the modernity of the C# language codebases, as most of the version-specific features affect the grammar of the language, thus it is possible to detect those by applying the discussed approach. It is not possible to detect all version-specific features, as some of them affect the compiler, standard libraries, concurrency, or other language aspects that are undetectable with applied static analysis methodology. Thus it can not be applied to the full extent.

9 FUTURE WORK

9.1 Methodology

The research is sufficient to answer the defined research question, but there are some parts that were left under-researched and could provide interesting results. By observing the obtained results, it would be interesting to find out what exact features from each language version are the most used and why. By researching each feature separately, current results could be interpreted in more detail, which could explain why some of the results look unnatural at first glance. By researching each feature, we could also point out trending features and the ones that were almost never used, which could be meaningful for C# language designers and developers to estimate their work from another perspective.

9.2 Normalization

The modernity meter currently applies just one normalization technique, which is Max normalization, when the modernity signature is collected for different versions of the codebase. This approach leads to different normalization techniques just scaling the amount of version-specific features, but not providing any insights. A good addition to the modernity meter would be applying normalization techniques straight away when analyzing the syntax tree, as it could provide additional information about the evolution of the codebases.

REFERENCES

[1] High-level programming language developed by Microsoft that runs on the .NET Framework. https://learn.microsoft.com/en-us/dotnet/csharp/
[2] Microsoft Corporation. https://www.microsoft.com/nl-nl/
[3] Indicator of the popularity of programming languages. https://www.tiobe.com/tiobe-index/
[4] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler and J. Penix, "Using Static Analysis to Find Bugs," in IEEE Software, vol. 25, no. 5, pp. 22-29, Sept.-Oct. 2008, https://doi.org/10.1109/MS.2008.130
[5] M.M. Lehman, On understanding laws, evolution, and conservation in the large-program life cycle, Journal of Systems and Software, Volume 1, 1979, Pages 213-221, ISSN 0164-1212, https://doi.org/10.1016/0164-1212(79)90022-0
[6] B. Chess and G. McGraw, "Static analysis for security," in IEEE Security & Privacy, vol. 2, no. 6, pp. 76-79, Nov.-Dec. 2004, https://doi.org/10.1109/MSP.2004.111

[7] Chris Admiraal, Wouter van den Brink, Marcus Gerhold, Vadim Zaytsev, Cristian Zubcu, Deriving modernity signatures of codebases with static analysis, Journal of Systems and Software, Volume 211, 2024, 111973, ISSN 0164-1212, https://doi.org/10.1016/j.jss.2024.111973

[8] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler and J. Penix, "Using Static Analysis to Find Bugs," in IEEE Software, vol. 25, no. 5, pp. 22-29, Sept.-Oct. 2008, https://doi.org/10.1109/MS.2008.130

[9] Koshelev, V.K., Ignatiev, V.N., Borzilov, A.I. et al. SharpChecker: Static analysis tool for C# programs. Program Comput Soft 43, 268–276 (2017). https://doi.org/10.1134/S0361768817040041

[10] Robert Charles Metzger, 14 - The Way of the Computer Scientist, Editor(s): Robert Charles Metzger, In HP Technologies, Debugging by Thinking, Digital Press, 2004, Pages 473-507, ISBN 9781555583071, https://doi.org/10.1016/B978-155558307-1/50014-8

[11] R. Shaukat, A. Shahoor and A. Urooj, "Probing into code analysis tools: A comparison of C# supporting static code analyzers," 2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST), Islamabad, Pakistan, 2018, pp. 455-464, https://doi.org/10.1109/IBCAST.2018.8312264

[12] T. Sharma, M. Fragkoulis and D. Spinellis, "House of Cards: Code Smells in Open-Source C# Repositories," 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, ON, Canada, 2017, pp. 424-429, https://doi.org/10.1109/ESEM.2017.57

[13] W. Van den Brink, M. Gerhold and V. Zaytsev, "Deriving Modernity Signatures for PHP Systems with Static Analysis," 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), Limassol, Cyprus, 2022, pp. 181-185, https://doi.org/10.1109/SCAM55253.2022.00027

[14] Developer platform that allows developers to create, store, manage, and share their code. https://github.com/

[15] Cristian Zubcu. 2023. Effect of Normalization Techniques on Modernity Signatures in Source Code Analysis. https://essay.utwente.nl/96034/ Publisher: University of Twente.