Modeling of library functions in an industrial static code analyzer

1st Mikhail Belyaev
Ivannikov Institute for System Programming of Russian Academy of Sciences
25 Aleksandra Solzhenitsyna Str., Moscow, 109004, Russian Federation
mbelyaev@ispras.ru

2nd Egor Romanenkov
Faculty of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
1 building 52, Leninskie Gory Str., Moscow, 119991, Russia
esromanenkov@ispras.ru

3rd Valery Ignatyev
Ivannikov Institute for System Programming of Russian Academy of Sciences
25 Aleksandra Solzhenitsyna Str., Moscow, 109004, Russian Federation,
Faculty of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
1 building 52, Leninskie Gory Str., Moscow, 119991, Russia
valery.ignatyev@ispras.ru

***Abstract*—SharpChecker is an industrial level static analyzer, which is aimed at detection of various bugs in C# source code. Because the tool is actively developed, it requires more and more precise information about program environment, especially about results and side-effects of library functions. The paper is devoted to the evolution of models for the standard library historically used by SharpChecker, its advantages and drawbacks. We have started from SQLite database with the most important functions properties, then introduced manually written C# model implementations of frequently used methods to add support of data container states and have recently developed a model, built by a preliminary analysis of library source code, which allows to gather all significant side-effects with conditions for almost whole C# library.**

***Index Terms*—static analysis, library, source code analysis**

I. Introduction

There are four main methods for finding defects in programs: manual inspection, testing, dynamic analysis and static analysis. Testing requires to prepare a large number of tests, but even if the code is fully covered by tests, some part of program execution cases is left unconsidered. Fuzz testing, or fuzzing, is automatic generation of random correct and incorrect input data and running the program on that input data with error detection. Fuzzing requires a large number of program runs, and its applicability is limited by the difficulty of finding the input data that causes an error. Dynamic analysis also requires to prepare an execution environment and a set of input data, and it covers only the paths reachable on given input data.

Static analysis allows to find defects in programs without execution. The result of a static code analyzer is a list of potential defects found in the program with defect type, origin location and propagation trace.

Source-level static analysis allows the analyzer to get more information about the source code, including variable types and syntax, that is lost after transforming the program into the compiler's intermediate representation. Such information can be used to increase the quality of analysis and to report the defect trace precisely mapped to the source code. Moreover, there are several types of errors that could be discovered only using static analysis, for example, unreachable code, copy-paste errors and comparison of a pointer with `null` after dereferencing it.

Most projects use external libraries in binary form, so source-level static analyzer is unable to get any information about such functions and the resulting analysis quality is significantly reduced. To be able to produce precise results it is necessary to provide the most sensitive information about called external functions, for example, which exceptions can be triggered with their conditions, which values are changed, etc. This paper is devoted to the possible methods for data provision that were tested with the static analysis tool SharpChecker [1], which is also available as a part of Svace [2], [3] static analyzer tool-set. SharpChecker contains several analysis engines: AST, dataflow, symbolic execution [4], taint [5] and experimental machine learning based subsystem [6].

Modeling of external libraries code is an important task

in interprocedural source code analysis. Knowing the preconditions, return values and side effects of external functions can improve the precision and recall of analysis in general. Programming languages such as Java and C# have large standard libraries – Java Class Library and .NET Framework (or .NET Core) respectively, which are widely used by programmers. So this problem gets high priority even for initial analyzer version.

We will demonstrate the evolution of our approaches, that were used by SharpChecker in historical order. At each stage the specified method was sufficient to meet the requirements of the analyzer. During the evolution of algorithms, used by SharpChecker, or because of the development of detectors for new error types it becomes necessary to get higher precision of modeling, to cover huge amounts of new libraries, to model specific properties of several functions. Since SharpChecker is the industrial level tool which is deployed into large companies, we also have to consider such criteria as performance, results stability and determinism. As a result we can't be limited by the single approach and will show how we combine all of them to achieve better results.

The paper is organized as follows. In Section II we define the set of metrics and properties used to evaluate and compare the described techniques. Section III is devoted to the existing and well-tested approaches. In Section IV we describe the newly developed approach that we plan to finish and integrate into production. We present most serious problems that we have solved and address. Evaluation of the proposed approach using artificial examples and a representative set of open-source projects, containing more than 4 million lines of code (LOC) from 24 projects, is shown in Section V. In the last Section VI, we summarize the results of the work and present directions of future research.

II. Comparison criteria

Although SharpChecker uses several approaches for library modeling, it may be due to historical and other reasons, which are not based on the actual drawbacks of any specific technique. We propose the set of criteria that we will use for comparison in the conclusion. We classify the following list as the most important properties.

- *Scalability* – the approximate number of methods that are modeled using the specific approach. We measure only the order of magnitude of the corresponding value.
- *Performance* – the influence of resulting function model on the analyzer performance. Since it is hard to measure it, we will sort all methods and use comparative order as a value.
- *Completeness* is a logical property which clarifies if the given approach covers all method properties or it will be necessary to add something in future. For example, it is possible to specify only a single property of a method, such as whether it can return `null`. It will require a way to specify additional properties, such as precondition for `null` return value, or that method also uses locks or creates a resource that should be managed.
- *Maintainability* means how easy it is to update or correct existing models, when a new library version is published or to supplement with models for newly added functions.
- *Manual correction* – a property which specifies who is able to correct models manually: analyzer developer, user, both or nobody.
- *Size* – the total size in bytes of all models. The average size of a model of a given method could be calculated as a result of division of given value by the total number of methods, covered by the approach.
- *Private code* – a logical property, which means if it is possible for the user to add data about private libraries without sharing it with analyzer developers.

All listed metrics are calculated for SharpChecker, but the order of magnitude of resulting values is valid for general case because of the nature of the analyzed approaches.

III. Approaches

A. Hardcoded semantics

The simplest approach for modeling individual properties of several methods, which was used in the earliest versions of analyzer, is *hardcoded semantics*. It is a very flexible method, because it makes it possible to model almost everything. It is still used by SharpChecker. For example, the semantics of the following functions is hardcoded in the analysis engine:

- `string.IsNullOrEmpty(string)`,
- `string.IsNullOrWhitespace(string)`,
- various built-in, NUnit and xUnit asserts,
- `System.Environment.Exit(int)`,
- `System.Nullable<T>.HasValue`,
- `System.Nullable<T>.Value` and so on.

The main drawback of this approach is inability to scale for substantial share of methods that is necessary to cover, because writing support for each function in different parts of analysis engine is rather difficult. At the same time it allows to represent some unique properties, for example, that `System.Environment.Exit(int)` terminates the execution and that `System.Nullable<T>.HasValue` doesn't dereference `this` and instead compares it with `null`. Since the number of such properties and corresponding methods is very small and these properties are very different, so that the unified approach is not possible, and hardcoding is the only way to cover such situations.

B. Property database

One of the earliest and most important detectors of SharpChecker is related to null dereference. It requires to know, for example, if some method throws `ArgumentNullException`, when it's argument is `null`. For the majority of functions such information is provided in documentation. As a result, the first approach to library function modeling implemented in SharpChecker [1] is extracting some properties of methods from the documentation at MSDN, which is now replaced with Microsoft Docs [7], and storing it in per-method XML files. The extracted properties include:

- method signature, which includes its fully qualified name and types of parameters,
- a list of possible thrown exception types,
- the possibility of returning null,
- which parameters must be non-null.

Other properties are manually added for some methods. Such method properties specify whether a described method:

- disposes `this`,
- enters or exits a synchronization monitor,
- changes the inner state of `this`,
- is *pure* (has no side effects and returns the same value if called with the same arguments multiple times),
- uses multithreading,
- is an obsolete cryptographic algorithm implementation,
- is a virtual method which must be called in any method that overrides it.

Later thousands of XML files were transformed into an SQLite database of method properties, because such database allows more compact storage and faster lookup. Boolean fields are represented as bit flags in a single `Attributes` integer column.

This approach allows a rather compact storage of method annotations, doesn't depend on the availability of library source code, covers all methods that have available documentation at the moment of parsing and allows easy manual correction of annotations.

However, it doesn't represent a large part of method semantics, including exception conditions, field assignments, return value conditions etc., and adding any new information field requires the developers to fill it by hand for all known methods. When new methods are added into .NET Framework, manual fields should also be filled.

An additional significant issue with such database is that it becomes outdated and it's hard to transfer manually added fixes and new information into updated version. Even if we overcome this difficulty, we cannot verify that the resulting database doesn't contain parse errors, which could suddenly appear only on customers' code.

C. Code models

C# programs widely use collections – data structures and methods for data manipulation. C# programming language itself supports LINQ [8] – SQL-like language which is used as a unified API for collections library. For path-sensitive analysis it is very important for the analyzer to be able to understand if any specific collection could be empty or not, because a lot of errors are related with corner cases, when loop iteration over collection will not perform even a single iteration. Listing 1 demonstrates an example of such situation, when `lastSatisfied` could have value `null` if `list` has no elements. However, it can never happen, because an element is always added to the list before the loop, so it's never empty. It is impossible to reuse both described approaches: there are too many types and interfaces to hardcode all of them, and it is impossible to hold internal state of object, which represents a collection, using a database.

```
1  private bool check(object obj) => true;
2  private void foo(IList<object> list, object elem)
3  {
4      object lastSatisfied = null;
5      list.Add(elem);
6      foreach (var elem in list)
7      {
8          if (check (elem))
9              lastSatisfied = elem;
10     }
11     Console.Writeline(lastSatisfied.ToString());
12 }
```

Listing 1. NRE defect example

To overcome the described problem, we have developed another approach that still doesn't require the source code of libraries to be available. This approach is to model the most frequently used methods with handwritten C# code. It isn't necessary for such implementation to be a complete drop-in replacement of the corresponding library method. Instead, such code contains only some key features useful for analysis, such as management of the internal state of an object, exception conditions, data flow between arguments, fields and return value, the possibility of returning null etc. When the analysis engine is improved, it becomes able to analyze such models better, and so the quality of library modeling improves even though the models are not changed.

The implementation of the given method is not very hard. The models are organized similarly to their implementation in the standard library using identical names. All models are joined into the single C# project, which is analyzed before the target code. So, analyzer can differ these implementations from user code only because of special labels and prefer to apply summaries of models instead of using library database.

The main disadvantage of this approach is that the models are written manually, and that means that they cover only a small fraction of .NET Framework methods. Another theoretically possible drawback is that if we managed to write manual models for a large number of methods, SharpChecker would spend a lot of time analyzing these models in the beginning of each run. This disadvantage is important especially for using SharpChecker with small projects. Such modeling is very significant for taint analysis engine [5], because it makes it possible to specify paths of tainted data propagation and cleanup between arguments, object fields and return value.

The described approach seems to have an evident extension. What if it would be possible to perform preliminary analysis of open source standard libraries, save its results similarly to a regular user method and reuse for all further static analyzer runs? To make it possible to explore the proposed approach it is necessary to solve several serious problems. First of all, regular code of library methods implementation is much more complex than used for models, so the analyzer should be very precise, stable and deterministic, because an error of analysis of library will be distributed and multiplied into dozens of errors on target user code. Analyzer must also be very efficient, because the conditions of every interesting property, gathered from real implementation, are big and complex. Analyzer should be able to generate compact summaries, because the

summaries of preliminarily analyzed libraries become a part of tool distribution package. It took several years to overcome mentioned problems, and the most interesting issues and its solutions are presented in the next section.

IV. Library pre-analysis

Growing collaboration of Microsoft with open-source community has lead to development of .NET Core – a cross-platform open-source runtime and class library for .NET programming languages. .NET Core is partially compatible with .NET Framework, although now it doesn't support all classes, methods and features of .NET Framework. .NET 5, which is planned to be released this year, will join .NET Core and .NET Framework into one open platform.

The SharpChecker's interprocedural analysis engine works by traversing the call graph in the reverse topological order, analyzing methods from callees to callers and saving all the knowledge that it has gained about a method into a method summary, which can be kept in memory or written into a file. When the analyzer encounters a call of an already analyzed method, it *applies* its summary to bring that knowledge into the context of current method.

A fully automatic approach to library modeling is to analyze the source code of a library (.NET Core in this case, but the technique could be used with any library with source code) with SharpChecker and to save all summaries of the library methods into a file. Then, when analyzing a user's project, SharpChecker loads the summaries of called library methods from the file and applies them.

Like the previous approach, the quality of library modeling is closely connected with the quality of the main analysis engine. This property has the following consequences:

- when the analysis engine is improved, the quality of library modeling increases;
- improvements that are aimed to improve the quality of models make the main analysis better;
- to get good results when using library summaries, it is necessary to fix many previously unknown bugs in the analysis engine and even to implement some new features.

A. Summary size reduction

Before the introduction of pre-analysis for library modeling, SharpChecker used method summaries during analysis to keep gathered information and cleared summary after the analysis of all callers. The first implementations used identical summaries that were serialized and stored on disk. The resulting file has 0.5 gigabyte size for 137 thousand records. Initially the size was even more, because it contained multiple instances of summaries for each method, as some statistical information is changed after the first serialization and needs to be saved again, and it's impossible to remove previously serialized record from a compressed archive without rewriting the whole archive. To use the approach in production it would be necessary to add that huge file into distribution package, and it is inconvenient to use and update.

A method summary consists of several components, gathered by the common analysis engine and separate error detectors. Analysis of the biggest summaries showed that a lot of space is occupied by the information, required for statistics-based detection of possible null return value dereferences. Since it is not necessary to discover errors in the library during the analysis of the user code, all statistical data for library methods were removed. Summaries of all private methods, which are not available from the user code, were also removed. All these fixes allowed to reduce the total size to 230 MB for 80 thousand records.

Currently we are developing additional more complex features which will also decrease the total file size, such as simplification of stored conditions.

B. Performance issues

Previously used methods for modeling the majority of library functions were unable to store pre-conditions. Despite of them pre-analysis produces tons of conditions for every property. Joined with user conditions they reach performance limits. For example, consider the example at Listing 2. Modeling based on the pre-analysis will generate condition `arg1 != null && arg2 == null` instead of the simpler form `arg2 == null`, build by the previous library modeling subsystem. Moreover, every condition, obtained as a result of analysis of `foo` will contain `arg1 != null && arg2 != null`, significantly increasing the complexity of every further condition.

```
1 void foo(object arg1, object arg2) ]
2 {
3     if (arg1 == null)
4         throw new ArgumentNullException();
5     if (arg2 == null)
6         throw new ArgumentNullException();
7
8     ...
9 }
```

Listing 2. An example of a method throwing ArgumentNullException

To address the problem we develop condition simplification methods. Since all conditions are automatically generated, they have a lot of redundancy. Manual inspection of generated conditions showed that *variable substitution* will remove a lot of useless conditions. We extract the known values from equality for conjunction or its negation for disjunction and substitute value into other components of condition. But conditions, generated by analyzer are very complex. The current limit for every condition is 200000, what means that a condition tree can have 200000 nodes. Condition tree is a non-binary tree, where leaves represent atomic conditions, such as `x = 5`, and non-leaf nodes contain one of three operations: negation, conjunction or disjunction. Thus simplification should be carefully preformed, because it is necessary to prevent multiple traverse of such huge conditions.

C. New features

An example of a feature that has not been implemented yet is field sensitivity. If `string.IsNullOrEmpty(string)` method was implemented like this:

```
static bool IsNullOrEmpty(string value)
{
    return value == null ||
           value == string.Empty;
}
```

Listing 3. Simple implementation of `string.IsNullOrEmpty`

the analyzer would correctly 'understand' its behavior from summaries, removing the necessity of hardcoded semantics. However, in .NET Core this method has the following implementation [9]:

```
static bool IsNullOrEmpty(string value)
{
    return (value == null ||
        0u >= (uint)value.Length) ? true : false;
}
```

Listing 4. The original implementation of `string.IsNullOrEmpty`

The comments say that such trick with a ternary operator and >= instead of == are used to workaround some performance issues with the current JIT compiler. Analysis of such implementation discovered a bug in the control flow graph, where the logical disjunction operator affected only the control flow, but the value for its result was not created. Even after fixing this issue, the analyzer can't infer that `string.IsNullOrEmpty(x) && x == "Hello, world!"` is `false`, because it lacks some kind of field sensitivity: $a = b \not\Longrightarrow a.f = b.f$.

Using summaries allows the analyzer to provide warning traces inside library functions that make it more easy for the user to understand the reason of the defect and to determine if the warning is true positive or false positive. It is similar to Microsoft SourceLink [10] technology that allows to browse and step library source code when debugging.

V. Practical evaluation

The current state of summary pre-analysis implementation doesn't allow to deploy it to the main analyzer branch, since it has significant regressions in analysis results. We consider separate major error detectors consequently to decrease the number of false positives, which were produced by the approach. The results of evaluation are presented for the `NRE.*.ARGUMENT` checker family that should be one of the most beneficial from the new technique.

A. Artificial examples

Evaluation on existing set of tests used during SharpChecker development showed that the analyzer, unfortunately, fails more tests with the pre-analysis summaries than without any function models (except hardcoded). One of the causes is that we focused on null dereference checker during the development of library pre-analysis approach, and some other checkers are either accidentally broken or not properly supported. However, even if we filter only null dereference tests, we still get more failures (18 versus 10) when using pre-analysis summaries. Some tests are specifically designed to check warnings for library methods, which may return null, and now they don't pass because the analyzer doesn't distinguish between library and source methods. Another cause is that this approach is currently in early preview stage, and there are many things to improve, because this method is very sensitive to any analysis errors.

B. Open source projects

Evaluation on a set of 24 open-source projects confirmed that the analyzer is not yet ready to use summary pre-analysis instead of the database. Among new null dereference warnings there are many false positives, and some true positives disappeared. However, there are some new true positive warnings which were not found before due to outdated and incomplete database, and the trace that shows where inside the library the null value is dereferenced is useful for reviewing expert. Some warning changes seem to be accidentally introduced analyzer imprecisions or bugs, because these warnings are not related to external methods at all. We will continue our efforts in development of summary-based approach to make it production-ready, as it has significant advantages over other approaches, and fixing issues that arise during the analysis of libraries helps to improve the analysis quality for user code.

As for the performance, the total slowdown of summary-based analysis is about 1 hour ($\approx 60\%$) on this project set in comparison with property database approach. It may be noted that the slowdown for single project may be slightly less, because for a set of projects the summaries are loaded from file and deserialized multiple times. However, most time is spent due to increased complexity of analysis, because the analyzer has significantly more information about methods, such as value flow and exception conditions.

C. Related work

ReSharper [11] (a code analysis and refactoring plugin for Visual Studio) and Rider [12] (a .NET IDE from JetBrains, which is based on ReSharper analysis engine) use XML annotation files for .NET Framework, .NET Core and many widespread libraries, such as Xamarin, Entity Framework, NUnit, xUnit, Newtonsoft.Json, log4net etc. Currently annotations allow to specify such properties of methods [13]: whether a method can return null, is pure, invokes passed delegate, takes a path as a parameter, is implicitly used through reflection, modifies a collection, is an assertion with condition, terminates the control flow. Annotations can also define method contracts, which specify the dependencies between method input (parameters) and output (returned value, out parameters and control flow effect). JetBrains annotations are documented, regularly updated and released under MIT license, so they should be considered for usage in SharpChecker as a free source of quality improvement.

Coverity static analyzer [14] allows users to specify the behavior of external functions using code models, that are similar to our models [15]. Such models are compiled and analyzed like regular code, but they have special builtin function invocations that represent specific features, for example, `nondet()` represents an unknown condition, `unknown()` represents an unknown value and

`UseAfterFreePrimitives.use(x)` represents a usage of a value. Coverity can produce such models automatically by analyzing source code with `cov-make-library` command, and a set of models for Java and Android classes is bundled with the analyzer. Although this approach allows to use both automatically generated and manually written models, it has the following disadvantages:

- generating models from the internal representation of analysis data loses some information in comparison with serialized summaries;
- constant effort of the analyzer's developers is required to write and support code that generates the model for each feature;
- compilation and analysis of models consumes additional time, though it's less than time required for the analysis of original source code.

Some languages and frameworks have built-in support for annotating code properties and checkers in the compiler that verify these properties. For example, C# adds *nullable reference types* analysis, that makes all reference types non-nullable by default and requires the user to mark all variables which may hold `null` with a `?` postfix. These annotations are preserved in compiled libraries and allow to check, for example, whether a function can return null or can take null as a parameter value. `AllowNull`, `MaybeNullWhen`, `NotNullWhen` and other attributes [16] allow to specify more complex nullability cases. However, conformance to the annotations is not enforced: the compiler emits warnings (not errors) when it detects violations, and these warnings may be suppressed with a postfix `!` operator. So, a static analyzer should ignore nullability annotations for user code and use them for library functions only when there are no other models. Java uses `@Nullable` and `@NotNull` annotations for this purpose, but it adds a third state "— not annotated. The fact that nullability annotations are integrated into the languages and checked by the compiler stimulates programmers to annotate their code; on the contrary, a developer that doesn't use a particular static analyzer wouldn't provide annotations for his library for this static analyzer. However, these attributes, annotations and language features describe only some properties like nullability, and do not replace proper modeling of library functions.

VI. Conclusion

The paper presents all pros and cons of different approaches used by SharpChecker. Major benefits and drawbacks of all described approaches are summarized in table I.

Table I demonstrates that pre-analysis approach is more flexible and covers all features of all other approaches. The major drawback of the technique is significant performance degradation. But the resulting models always have quality suitable for analyzer, because models are generated by the same algorithms and the quality will increase together with improvement of analysis engine itself. It covers all libraries, since most of them have opened sources. Library models can be easily updated and moreover created from private sources by the user independently. The additional advantage of the approach is predictability – it is unlikely to get very poor results due to parse or human error that could suddenly appear on inaccessible code. All algorithms, used for generation and application are tested twice, similarly to a compiler bootstrap.

TABLE I
Caption

	Hardcoded	Database	C# model	Pre-analysis
Scalability	10^1	10^4	10^2	10^4
Performance	0	1	2	3
Completeness	$-$	$-$	$-$	$+$
Unique prop	$+$	$-$	$+$	$+$
Maintainability	$-$	$-$	$-$	$+$
Manual corr.	Developer	Both	Both	Nobody
Size	N/A	27 MB	0.8 MB	230 MB
Private code	$-$	$\mp$	$-$	$+$

The production version of SharpChecker now uses hard-coded semantics and a database together, and used C# models before recent changes. The authors think that usage of four different ways for library modeling is redundant, so when analyzer is ready and the pre-analysis approach is tested enough, it will replace all others completely. But it doesn't mean that this way is the only right technique, because it is impossible to use it for earlier versions, when analyzer was not able to extract and reuse data precisely and efficiently and, moreover, didn't require such high accuracy of modeling.

Further improvement of library modeling may be the analysis of library binaries. Since C# uses CIL [17] as an intermediate representation and it could be decompiled back to C# source code with a reasonable quality and processed as regular sources, so it doesn't require a separate analysis engine for CIL. The most advanced use case can include decompilation and analysis of binary libraries on the fly.

References

[1] V. K. Koshelev, V. N. Ignatiev, A. I. Borzilov, and A. A. Belevantsev, "SharpChecker: Static analysis tool for C# programs," *Programming and Computer Software*, vol. 43, pp. 268–276, 2017.

[2] V. P. Ivannikov, A. A. Belevantsev, A. E. Borodin, V. N. Ignatiev, D. M. Zhurikhin, and A. I. Avetisyan, "Static analyzer Svace for finding defects in a source program code," *Programming and Computer Software*, vol. 40, no. 5, pp. 265–275, 2014.

[3] A. Belevantsev, A. Borodin, I. Dudina, V. Ignatiev, A. Izbyshev, S. Polyakov, E. Velesevich, and D. Zhurikhin, "Design and development of Svace static analyzers," in *2018 Ivannikov Memorial Workshop (IVMEM)*, 2018, pp. 3–9.

[4] V. Koshelev, I. Dudina, V. Ignatyev, and A. Borzilov, "Path-sensitive bug detection analysis of C# program illustrated by null pointer dereference," *Proceedings of the Institute for System Programming of RAS*, vol. 27, pp. 59–86, 01 2015.

[5] M. V. Belyaev, N. V. Shimchik, V. N. Ignatyev, and A. A. Belevantsev, "Comparative analysis of two approaches to static taint analysis," *Programming and Computer Software*, vol. 44, pp. 459–466, 2018.

[6] G. Morgachev, V. Ignatyev, and A. Belevantsev, "Detection of variable misuse using static analysis combined with machine learning," in *2019 Ivannikov ISP RAS Open Conference (ISPRAS)*, 2019, pp. 16–24.

[7] Microsoft Docs. Accessed: Apr. 10, 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/?view=netframework-4.5

[8] E. Meijer, B. Beckman, and G. Bierman, "LINQ: Reconciling object, relations and XML in the .NET Framework," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD 06. New York, NY, USA: Association

for Computing Machinery, 2006, p. 706. [Online]. Available: https://doi.org/10.1145/1142473.1142552

[9] Source code implementation for `string.IsNullOrEmpty()`. Accessed: Apr. 10, 2020. [Online]. Available: https://github.com/dotnet/coreclr/blob/1f3f474a13bdde1c5fecdf8cd9ce525dbe5df000/src/System.Private.CoreLib/shared/System/String.cs#L439-L448

[10] Source Link – a language- and source-control system for providing source debugging experiences for binaries. Accessed: Apr. 10, 2020. [Online]. Available: https://github.com/dotnet/sourcelink/blob/master/README.md

[11] Features – ReSharper. Accessed: May 18, 2020. [Online]. Available: https://www.jetbrains.com/resharper/features/

[12] Features – Rider. Accessed: May 18, 2020. [Online]. Available: https://www.jetbrains.com/rider/features/

[13] External Annotations – Help | ReSharper. Accessed: May 18, 2020. [Online]. Available: https://www.jetbrains.com/help/resharper/Code_Analysis__External_Annotations.html

[14] Coverity Static Analysis. Accessed: May 18, 2020. [Online]. Available: https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/SAST-Coverity-datasheet.pdf

[15] Coverity 2018.09 Command Reference. Accessed: May 18, 2020. [Online]. Available: https://www.academia.edu/38375284/Cov_command_ref

[16] C# Reserved attributes: Nullable static analysis | Microsoft Docs. Accessed: May 18, 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/attributes/nullable-analysis

[17] CIL – Common Intermediate Language. Accessed: Apr. 10, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Common_Intermediate_Language