ASTES

Advances in Science, Technology and Engineering Systems Journal Vol. 5, No. 6, 561-570 (2020)

www.astesj.com

Second International Virtual Conference on Multidisciplinary Research 2020

ASTESJ

ISSN: 2415-6698

sharpniZer: A C# Static Code Analysis Tool for Mission Critical Systems

Arooba Shahoor[1,*], Rida Shaukat[1], Sumaira Sultan Minhas[1], Hina Awan[1], Kashif Saghar[2]

[1]*Software Engineering Department, Fatima Jinnah Women's University, Rawalpindi, 46000, Pakistan*

[2]*NESCOM, Centre for Excellence in Science and Technology, Rawalpindi, 46000, Pakistan*

ARTICLE INFO

Article history:
Received: 10 September, 2020
Accepted: 07 October, 2020
Online: 20 November, 2020

Keywords:
Software Testing
Software Quality
Verification and Validation,
Safety-Critical Programming

ABSTRACT

Until recent years, code quality was not given due significance, as long as the system produced accurate results. Taking into account the implications and recent losses in critical systems, developers have started making use of static code analysis tools, to assess the eminence of source code in terms of quality. Static code analysis is conducted before the system is sent into production. The analysis aims to identify the veiled defects and complex code structures that result in the decrement of code quality or are likely to become a cause of malfunction during execution. To address this line of work, this research paper presents a static code analyzer for C#, named as sharpniZer.

The key purpose of this tool is to verify the compliance of the source code written in C#, in congruence with the target set of rules defined for analysis as per the accepted industry standards set particularly for the development of mission-critical systems. sharpniZer efficiently figures out the lines of source code that hold probable concern appertain to the category of design rules, usage rules, maintainability rules, inefficient code, complexity, object model and API rules, logical rules, exception, incomplete code, and naming conventions. Each violation encountered in source code is ranked by the severity level as: critical, major, and minor. The tool shall prove to be worthwhile, especially if utilized in critical systems.

1. Introduction

The prevalence of software has raised serious concerns in the software industry to think of ways in which software quality can be ensured. How the quality of the software system or the underlying source code can be quantified may differ from system to system. However, the system must be optimized and proficient in terms of parameters including maintainability, testability, reusability, resource consumption, etc [1].

Most of the aspects of software quality largely depend upon the skills and expertise of the development team. As the requirements from enterprises regarding system functionality are becoming critical, the quality of source code is likely to diminish. A bad-written code leads to increased resource consumption and decreased efficiency and productivity [2, 9].

The hidden defects in source code, if not identified and addressed appropriately, can ultimately lead to unreliable system behavior during execution. The systems need to be thoroughly tested for the detection of loopholes left during development. The extensively used techniques include static code analysis and dynamic code analysis. However, the practice under consideration for this work is that of static code analysis.

1.1. Current Scenario

As previously stated, static code analysis has become indispensable particularly for safety (or mission) critical systems where there is no room for even trivial bugs at the static or runtime stage. History testifies several incidents that stress the need of using code analysis tools. Below we will briefly discuss 3 such incidents where ignoring trivial coding flaws cost the developers heavy losses.

1.1.1. The Ariane 5 Explosion

Ariane 5, a heavy-lift launcher, made to launch a 3-ton payload into orbit, cost a total of 10 years and $8 billion to be manufactured. On its first launch in July 1996, the rocket soon diverted from its path of flight due to a diagnostic produced as a result of a software exception. The exception was caused during

*Corresponding Author: Arooba Shahoor, arooba.shahoor@gmail.com

a data conversion [11]. Specifically, a 16-bit value of type float was being converted to a 16-bit signed number, i.e. the value being converted was much greater and could not be possibly represented by a 16-bit number, consequently leading to an operand error [12]. Since it had reached an angle of attack of more than 20 degrees, the launcher started to disintegrate, and finally obliterated itself through a self-demolishing method, along with 4 satellites that it carried within itself [13].

1.2. Patriot Missile Error

At the time of gulf war (2 August 1990 – 28 February 1991), 28 American soldiers lost their lives, and yet 100 others severely wounded when the US missile defense system failed to detect the Scud missile that the Iraqi forces launched [5]. The root cause of the failure lies in the conversion of time from integer to floating value, which consequently resulted in a major shift in the range gate [6, 28].

1.2.1. AT & T (American Telephone & Telegraph) Network Goes Down

About 75 million call requests were not answered and approximately 50 % of the network of AT&T went down on January 15, 1990, when a bug in a single line of code incurred the network failure for several hours [7]. Specifically, the problem occurred where a break statement within a switch case, was nested within an if clause. Companies linked to its services faced fiscal setbacks, among which was "American Airlines", whose incoming calls were reduced by two-third owing to the network crash [8, 29].

The incidents mentioned above, stress upon the need and significance of thorough analysis of the code before the runtime state. This paper presents a static code analysis tool for C#, named as sharpniZer. The rest of the paper is organized as follows: part B of this section will delineate the basic aim and purpose of the utilization of static code analyzers in the software development process. Section 2 reveals the solution proposed in this paper i.e. a static code analyzer for C# that is designed to detect such seemingly non-destructive but critical bugs usually ignored in the development of mission-critical systems. This section will also discuss the categories in which the rules implemented by the proposed analyzer can be generally segregated. Part 2 of section 2 will present the methodology adopted for the development of the tool. It also indicates the flow of the application. Section 3 outlines the system features and characteristics of sharpniZer. Each module is briefly presented; the figures attached to each module shall assist the reader in gaining better insight into the tool. Section 4 gives a comparison of the proposed tool with other existing tools in a tabular format. The textual summary of the comparative analysis is given in section 5. Section 6 presents the direction of future work i.e. discussion of the ways the tool can be extended or modified, to cater to the rapidly changing market needs. Section 7 concludes the research presented in the paper.

1.3. The Goal of Static Code Analyzers

Static code analysis chiefly focuses on examining the source code, without actually executing it, to help ensure that the code is abiding by the established coding standards. This helps recognize such flaws and code constructs that may decrease code quality, or in the worst-case scenario, incur a costly disaster month or years later. The article an overview on the Static Code Analysis approach in Software Development [30] talks about and objectives for developers to introduce static code analyzers in the testing phase of the development life cycle and here we summarize six of those factors

- Saves time and money

Reviewing code manually can take up much time, in comparison to which automated static code analyzers are much faster. These analyzers can take large sets of known bugs (common as well as uncommon) and combine them with special algorithms to track them anywhere in the code in hand, which enables the bugs to be detected in a matter of seconds which would otherwise take hours or even days.

- In-Depth Analysis

With manual testing, it is highly likely to overlook some code executions paths, which is not the case in automated testing. Automated testing is performed as we build our code, so we can get an in-depth analysis of where potential issues might lie, under the rules applied by the user.

- Accuracy

It is always possible for manual reviews to be inaccurate or error-prone. This is another aspect where code analyzers come in handy. They can specify exactly where and when the error occurs without any ambiguity. Issues such as stack overflows or race conditions, that only show up when code goes into production, are hard to figure out if not resolved beforehand (through static code analysis) since the scenarios are less likely to be able to be recreated.

- Ease of use

A clear benefit the analyzers provide is their ease of access and usability. These analyzers come in stand-alone as well as integrated form, you can easily integrate analyzers with your code editing tool. Moreover, everything is performed automatically by the analyzer, it does not require any profound or in-depth knowledge or expertise on the part of the user.

- Comprehensive Feedback/Overview

At the end of the analysis, the user is provided with a detailed overview of the code and the violations (often with visualizations), which helps the user to get to the action and work on fixing the violations there and then.

- Uniform Code

Code analyzers analyze the code against set coding standards, which means by introducing a particular code analyzer for code testing, the corporate is enforcing the use of best coding practices which will consequently set a uniform coding practice among all coders and programmers. This will facilitate developers in an easier and quicker understanding of each other's code.

2. Proposed solution

Taking into account the concerns about the quality of code and recognizing that the cause of recent losses associated with the critical systems is the lack of appropriate code analysis, a static

code analysis tool, named as sharpniZer, has been developed. sharpniZer efficiently analyzes the source code written in C# against the target set of (150) rules defined for analysis as per accepted industry standards. As soon as the analysis completes, the user is presented with the results of the analysis in multiple forms so that the user can easily gain a thorough insight into the quality of the code at hand.

Before diving into the implementation of the tools, let us first present the 10 categories in which the selected 150 rules are divided. The rules have been accumulated from multiple industry-wide accepted standards such as MISRA, JPL, CERT, CWE, etc. The rationale behind the selection of these 150 rules is the frequency of their violations along with their pertinence in the development of mission-critical systems.

2.1. Description of categories of rules

1. Usage Rules: This is the category of rules that ensures the proper utilization of the .Net Framework and correct usage of the common types provided by it. Failure in complying with these rules may have diverse repercussions; they can be complexity, maintainability, or performance issues.

This category contains rules such as "Dispose() & Close(): Always invoke them if offered, declare where needed." and "Do not omit access modifiers".

The former rule recognizes the problem that Class instances often have possession of unmanaged resources, such as database connections. Not disposing of these resources implicitly or explicitly after they are no longer needed, may lead to memory leakage. Similarly, not closing SQL connections after utilizing them, may result in a lack of connections available in case of connecting to the database again.

2. Design Rules: Every programmer or developer is unique and has his style when it comes to coding or implementing logic. Therefore, it is imperative that some standard of coding style is maintained that will be followed by all the developers. Design rules will thus outline certain guidelines for the way logic is to be implemented.

As an example, consider the following rule: "'out' and 'ref' parameters should not be used". 'Out' and 'ref' are required when variables have to be passed by reference. This requires above-average experience and skills in working with pointers and more than one return value as well as an understanding of the difference in the concepts of 'out' and 'ref'. Not all developers are expected to be of the same competency, and therefore to bring all developers to a level playing field, it is recommended to rather shun the use of out and ref.

3. Object Model and API Design Rules: These are the rules that are geared towards object-oriented features such as inheritance, encapsulation, etc. An example rule for this category is: "Always prefer interfaces over abstract classes." Since interfaces are not coupled with other modules, it can be independently tested, unlike abstract classes, that can be only tested during integration testing. Moreover, since in C# a class can inherit from one only class but multiple interfaces, it is highly recommended to prefer interfaces when the developer has a choice between them and abstract classes.
4. Exceptions: Exceptions are yet another part of the code that needs to be catered for with much caution. Consider the following rule: "If re-throwing an exception, preserve the original call stack by omitting the exception argument from the throw statement".

This rule guides and warns us about the case when we want to re-throw an exception. It tells us that if we re-throw the exception with the exception (ex) argument, the compiler will not consider it a "re-throw" of the original exception rather it will consider it as a new exception that occurred where the throw (ex) statement was written. Likewise, many such guidelines need to be considered when using exceptions in our code.

5. Complexity Rules: These rules deal with common coding practices that may seem harmless at the time of implementation but in the long run result in what has become known as spaghetti code, where the flow of control is hard to keep track of. Particularly for critical systems, spaghetti code proves to be extremely risky, since in such systems one can't afford to overlook the validation and verification of every possible path of the control flow [4].

An example rule for this category is "Continue statement should not be used" or "go to statement should not be used". Since goto and continue statements provide an alternative way to exit from control structures, for large code files, it becomes inevitably difficult to keep track of and verify all the paths of the control flow.

6. Inefficient code: Rules of this category cater to the coding practices that negatively affect the performance of the system that may either be the result of greater time or memory consumption.

For example, a rule for this category of issues is "Avoid string concatenation in loop". Since string is an immutable type, every time a string is concatenated using '+' or using 'Concat()', a new memory location is reserved for the newly formed string. If the number of iterations is unknown or is very large, such practice would highly deteriorate the performance of the system due to unnecessary consumptions of memory.

7. Incomplete code: As is suggestive of the title, rules that fall in this category cater to the sections of code that give the impression that some part of code is left out, be it intentionally or mistakenly. Such coding faults, above all things, raise the issues about the understandability of code [10].

As an example, to this category of rules, consider this rule: "Ensure that a 'switch' includes cases for all 'enum' constants", this rule suggests that if a switch statement is based on a variable of type enum, then leaving any of the constants in its cases would be considered as a coding error. To avoid this blunder as well as to enhance the understanding of the switch statement, it is advised to handle all constants of the enum.

8. Logical errors: Rules in this category cater to situations when the coder might feel that the logic or the approach used for implementing a code section is correct, but actually, that code section ends up serving some other purpose, as opposed to the one the coder had for it in mind. Such blunders result in

unexpected results, confusion in the understanding of the code, or redundancy and decreased performance [3].

Example rules for such scenarios are as follows: "Do not perform self-assignment" or "Do not compare identical expressions". Looking at these rules, it is obvious that they prevent redundancy overhead by warning the user not to waste time and memory on something that will have no effect or use whatsoever. Therefore, such an implementation should be regarded as a coding error.

9. Maintainability Rules: This category of rules chosen for our system is the one that deals with the issue of maintainability. These rules ensure that the size of the code or modules and values used within, do not exceed a particular limit, for better performance and efficiency of the system. One example of this category is "Avoid creating files that contain many lines of code". Very large files will not only create problems of maintainability but will lead to many other issues such as:

- Merge conflicts; if more than one developer needs to work on the code at the same time.
- Network traffic; if the system uses a version control system to which the entire file needs to be transmitted for every small change.
- Poor organization of code; since large file size might be the result of appending everything to the same file, rather than building related things separately in respective files.

10. Naming Conventions: This category outlines the conventions set for naming code files and identifiers. The identifiers include properties, variables, methods, fields, parameters, namespaces, interfaces, and classes. Certain guidelines for naming have a much more critical purpose than mere consistency, take for example this rule: "Avoid using the same name for a field and a variable". Since the field is just a local variable of the class, declaring any other local variable with the same name as the field will cause confusion between the two, therefore it is strongly recommended to use different names for fields and variables in general [16].

2.2. Implementation Details

The tool, sharpniZer, is developed in WinForm on .NET Framework of Visual Studio. For development, Waterfall model is chosen, whereas the implementation language is C#. The implementation process and logic is discussed in detail below:

In creating the analyzer, we made use of SDK called Roslyn, which provides the user with useful API's used for C# code parsing and analysis of language constructs. An analyzer created with the Roslyn SDK, inherits Microsoft's CodeAnalysis base class. Given a file as a string, Roslyn can parse it and create a syntax tree from it, using the CSharpSyntaxTree Class of the Microsoft.CodeAnalysis package. It can then access the root of the syntax tree using the following statement.

```
var root = (CompilationUnitSyntax)tree.GetRoot();
```

Once we get the root of the syntax tree, it is fairly easy to access the descendant elements (declarations, expression, etc.) in the code. Microsoft.CodeAnalysis library enables us to access various elements of a node. For e.g Microsoft.CodeAnalysis.CSharp. Syntax enables us to access any type of declaration in C# code.

To show how our analyzer detects the different categories of problems and how checks are implemented, we will present the snippets of code demonstrating the implementation of 2 of the 150 rules implemented by sharpniZer.

Rule: Do not assign to local variable in return statement

When the assignment is made to a variable in a return statement, the variable goes out of scope and the value assigned is never read. Hence resulting in code redundancy. Redundant source code is one that is bloated, less reliable, and difficult to maintain. Any expert programmer would agree that the harder it is to maintain the code, the more likely it is to contain bugs.

Moreover, in a highly knitted team setting where multiple developers are involved, any such code written by a programmer will create much confusion for another to read or understand, resulting in a great deal of wasted time and mental energy. In the following section, we present and explain the snippets from our program which detects and warns about such redundant assignment statements in C# code.

First, all the method declarations in the code are stored in a list.

```
IEnumerable<MethodDeclarationSyntax>
methodsdecroot.DescendantNodes().OfType<MethodDeclarati
onSyntax>();
```

Next, all the variable declarations within each method are stored.

```
foreach (var method in methodsdec) {
   foreach (var VarDec in
method.DescendantNodes().OfType<VariableDeclaratorSyntax>
()) {
        VarDecsList.Add(VarDec.Identifier.ToString());
   }
```

Then, all the return statements within each method are looped through.

```
foreach (var returnStatement in
method.DescendantNodes().OfType<ReturnStatementSyntax>()
) {
```

After which we loop through all the assignment expressions within each method are looped through.

```
foreach (var assignment in
method.DescendantNodes().OfType<AssignmentExpressionSyn
tax>()) {
```

The following code then checks if any of the assignments are made to the variables declared within the method (local variables) and is done in a return statement.

```
if (returnStatement.Contains(assignment) &&
VarDecsList.Contains(assignment.Left.ToString())) {
```

If so, then the line number of that particular assignment is added to the warnings list

```
warnings = "Do not assign to local variable in return statement";
lineno = assignment.GetLine()+1;
WarningsList.Add(lineno + "@" + warnings + "@" + level);
```

In the above example, the rule was implemented by working on merely syntactic level. However, if the rule to be implemented requires some more information about the nodes/syntaxes (such as types, members, namespaces and variables which names and other expressions refer to), we have to be able to retrieve the symbols of the expressions or declarations.

We can retrieve symbols once we have the semantic model for the syntax tree. The method SemanticModel.GetDeclaredSymbol() is used to get the symbol of a given declaration syntax, whereas SemanticModel.GetSymbolInfo() returns the symbol of an expression syntax. The implementation of the second rule presented here makes use of this.

Rule: "Use throw instead of throw e (e for exception) whenever rethrowing the exception"

Throw statements are used so that if, during the execution of a software program, an unexpected condition occurs, the system is unable to process the next statement and instead throws an 'exception' error that specifies the line where the problem occurred, along with the line the throw statement was specified at. Now, when that exception is caught, we can choose to re-throw it. However, when re-throwing the exception, most programmers make the mistake of using the throw statement with the exception object (e); doing so will make the stack trace information within the exception restart at the current location, such that it will then point to the line where exception was thrown, rather than where the problem, causing the exception, occurred.

Though it might not seem like a critical issue per se, any programmer would acknowledge that incorrect stack trace information can lead to much confusion and thereby potential disruption of any logic that would be based on it. Below we present our code implementation that would detect such statements (if any) within C# source code.

First, all the catch clauses in the code are stored through the descendants of the root node (as in the previous example).

```
IEnumerable<CatchClauseSyntax> catchcaluses =
root.DescendantNodes().OfType<CatchClauseSyntax>().ToList(
);
```

Next, the throw statements within a catch clause are stored.

```
foreach (var catchclause in catchcaluses) {
  var throws = catchclause.DescendantNodes(n => n ==
catchclause ||
!n.IsKind(SyntaxKind.CatchClause)).OfType<ThrowStatementS
yntax>() .Where(t => t.Expression != null);
```

Then, all the throw statements, that were saved in the list, are looped through, to get the expression symbol of each throw.

```
foreach (var @throw in throws) {
  var thrown =
model.GetSymbolInfo(@throw.Expression).Symbol as
ILocalSymbol;
```

If the expression symbol matches the exception identifier, it means the throw statement specifies the exception, which violates the rule. Hence the line number of that throw statement is added to the warnings list.

```
if (Equals(thrown, exceptionIdentifier)) {
   warnings = MessageFormat;
   lineno = ( @throw.GetLine()+1).ToString();
   mylist.Add(lineno + "@" + warnings + "@" + level);
}
```

3. System features and usage

Upon launching the tool, the user is asked to browse a .CS file or folder containing.CS files. Once the files are loaded, users can select the categories of rules upon which the analysis is to be conducted. The tool allows users to selectively enable the categories of rules (by default, all categories of rules are enabled). The analysis result presents the defiance and violations in source code in congruence with the underlying set of rules. An analysis summary is presented in the form of a dashboard, presenting the results in tabular form. The graphical representation adds to the visualization of analysis and assists users in gaining deeper insight into the results. An overview of the flow of the analysis process of sharpniZer can be seen in Figure 1.

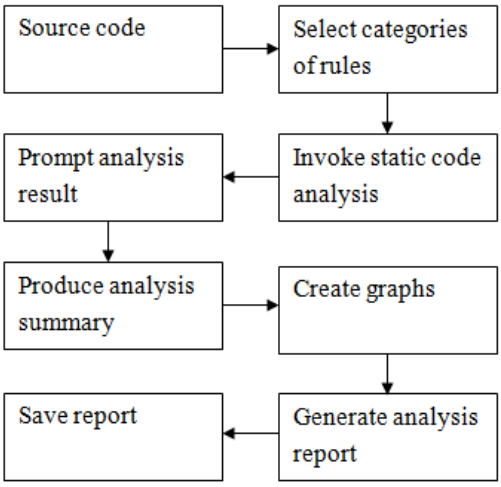


Figure 1: Methodology for conducting analysis

The above mentioned features of the system will now be elaborated per module below

3.1. Browsing C# file or project folder

User may CHOOSE C# file(s) to be analyzed in two ways:

Using the "Choose C# File" file option: allowing the user to choose any code file with the extension of .CS. After the file is selected, the content of the file will be displayed to the user in the Analyze tab (Figure 4). Clicking the Start Analysis button in the Analyze tab will start the analysis of the selected file.

Figure 2: Home page of sharpniZer

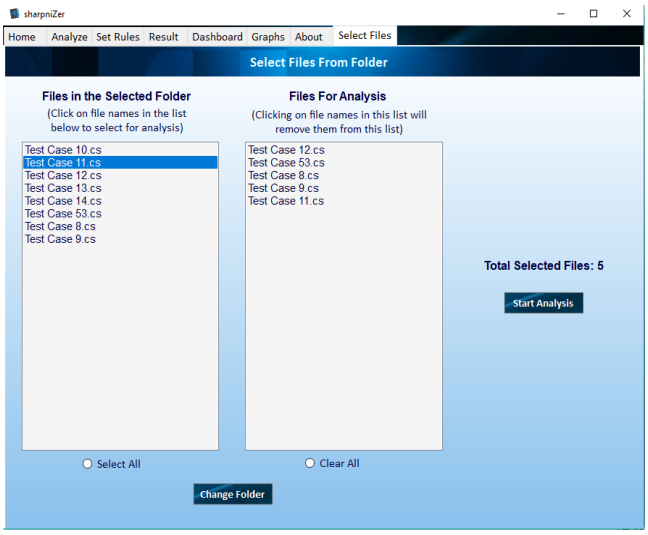


Figure 3: Selection of files for analysis

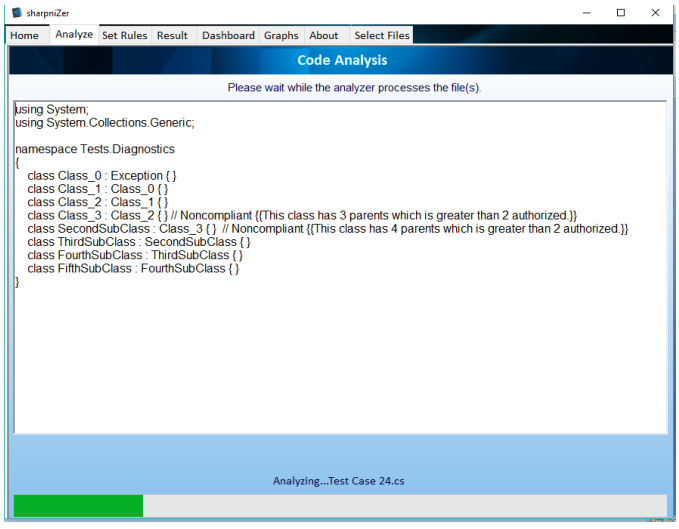


Figure 4. Analysis progress

Using "Choose C# Project" option: The user may choose any folder containing .CS files in the folder itself or its subfolders. If the user wants to deselect a file within that folder, they can click on that file's name in the "Files for Analysis" list (Figure 3), that file will be removed from the list of the files to be analyzed. The selected files will be analyzed sequentially and the progress of the analysis for each file will be displayed (Figure 4).

3.2. Select categories of rules for conducting analysis

Users can select or deselect any of the 10 categories of coding standard rules provided. The available categories can be seen in Figure 5.

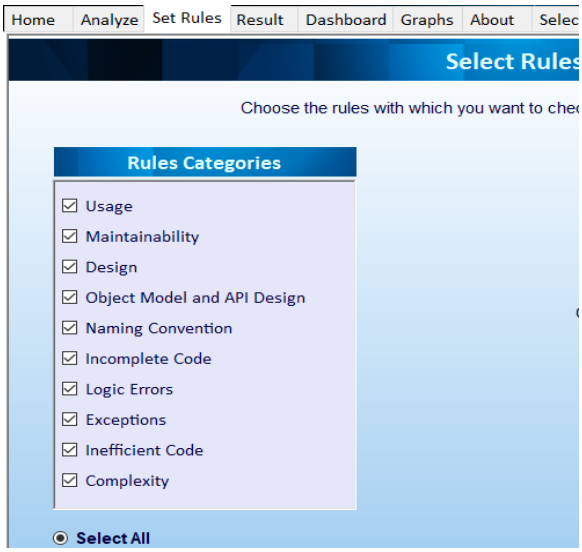
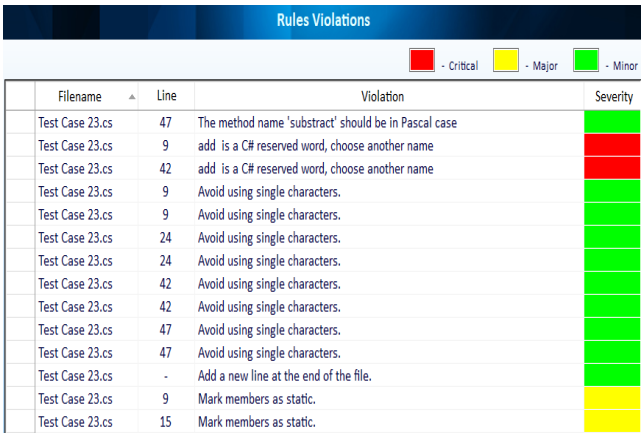


Figure 5. Selection of categories

3.3. Viewing analysis results

As soon as the analysis of the selected files(s) finishes, the tool switches to Results tab (Figure 6), which displays the result of the analysis. The analysis result specifies the filename(s), line no. at which the violation has been encountered, the violated rule, and the severity of the violation.



Filename	Line	Violation	Severity
Test Case 23.cs	47	The method name 'substract' should be in Pascal case	
Test Case 23.cs	9	add is a C# reserved word, choose another name	
Test Case 23.cs	42	add is a C# reserved word, choose another name	
Test Case 23.cs	9	Avoid using single characters.	
Test Case 23.cs	9	Avoid using single characters.	
Test Case 23.cs	24	Avoid using single characters.	
Test Case 23.cs	24	Avoid using single characters.	
Test Case 23.cs	42	Avoid using single characters.	
Test Case 23.cs	42	Avoid using single characters.	
Test Case 23.cs	47	Avoid using single characters.	
Test Case 23.cs	47	Avoid using single characters.	
Test Case 23.cs	-	Add a new line at the end of the file.	
Test Case 23.cs	9	Mark members as static.	
Test Case 23.cs	15	Mark members as static.	

Figure 6: Analysis result

3.4. Viewing Dashboard

The Dashboard presents the analysis summary. The numerical/tabular form of the results allows the user to gain deeper insight into the outcome (Figure 7). It also allows the user to generate an analysis report in a .pdf file that can be saved/downloaded and shared with team members (Figure 8).

Dashboard presents the following content:

- Total File(s) Selected
- Total Lines of Code
- Total number of violation(s)
- Number of violation(s) per file
- Category Violation(s) for all file(s)
- Category of violation(s) per file
- Severity of violation(s) for all file(s)
- Severity of Violation(s) per file

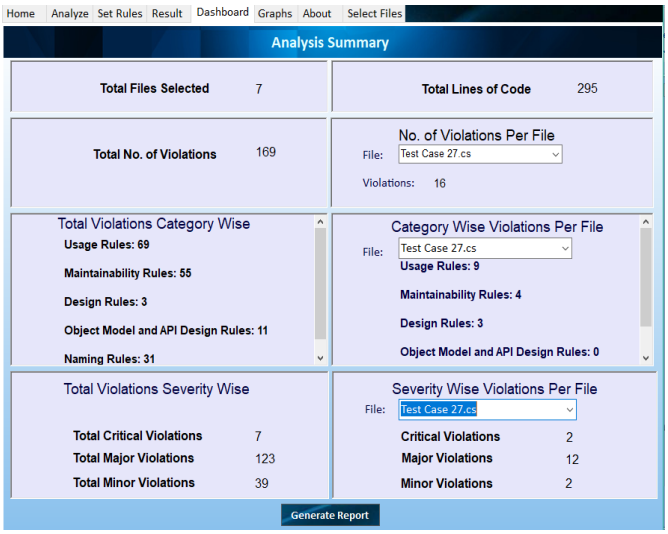


Figure 7: Dashboard

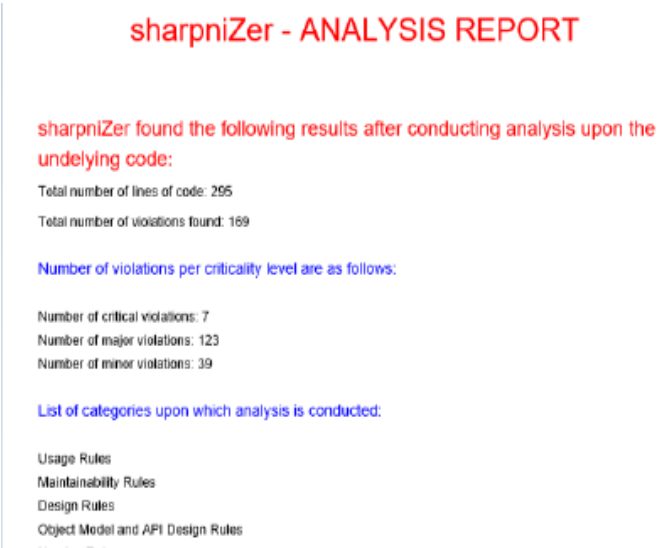


Figure 8: Analysis Report

3.5. Visual representation of Analysis Result

The Visual representation of results is often considered to be more comprehensive and elaborate, therefore, sharpniZer generates graphs by utilizing the numerical values of analysis results (Figure 9). The graphs can be generated for total violations, as well as for each file (if the analysis is conducted upon multiple files).

- The bar chart represents the total violations per category of rules chosen.
- The pie chart represents the total violations of rules per severity level.

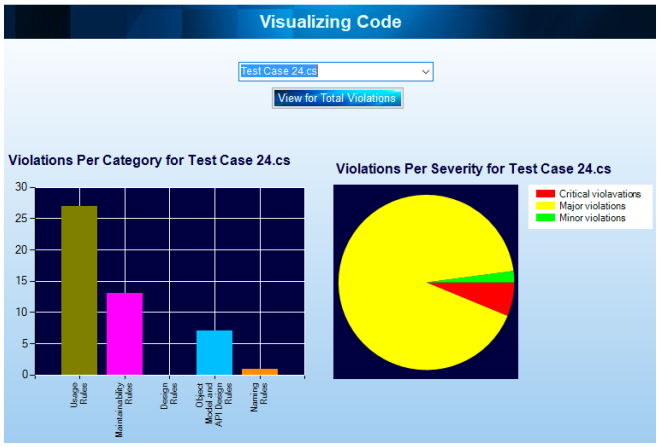


Figure 9: Graphical representation of analysis results

4. Review and Comparison of the Existing Tools with sharpnizer

Below we provide a review of other existing 9 tools that analyze code written in C# (a detailed comparison of C# code analyzers could be found in our previously published review article Probing into code analysis tools: A comparison of C# supporting static code analyzers [27]).

The 9 tools chosen for comparison in this research were selected based on their overall ranking resulting from numerous surveys combined with the fact that prior literature has proved these tools to be the most popular among researchers. The majority of the values for our evaluation are attained by gathering information from surveys and public reviews of users. Rest are procured by self-executing the tools and observing the performance in light of the mentioned 10 parameters.

Table I defines the parameters taken into consideration for comparison. These parameters were selected based on the survey of the code analyzer features most commonly considered by companies when selecting one to assess their mission-critical systems. Table II AND III present the value for each tool against these parameters. Whilst keeping the comparison authentic and unbiased, Tables II and III manifests the benefits sharpniZer has over some other quite decently known tools in use today.

5. Results

From the results of the comparative analysis presented in Tables 1 and Table 2, we can see that sharpniZer's installation process and usability is easier than some of the tools available. Though the number of rules implemented is yet lower than most in comparison, it supersedes many in providing customization of rules' selection, categorizing the nature of violations and covering a wide spectrum of the coding standards. In addition, unlike some of the contemporary tools, sharpniZer does not require code to be compiled first to perform analysis.

Table 1: Comparison Parameter

S. No.	Parameter	Definition
1	Usability	How user-friendly is the tool for a layman (scalable from easy to hard) (The classification was done based on the pubic reviews of the users of the tools listed and those of our client) [13]-[26]
2	Installation	How convenient is the process of installation/integration of the application (scalable from easy to hard) (Classification was done based on published surveys and our client's reviews) [13]-[26]
3	No of Defined Rules	The number of rules/metrics defined within the tool, so that the tool might assess compliance of these rules, during the analysis of codes.
4	No. Of Categories of Defined Rules	Number of types of coding standard violations defined by the tools
5	Requires compiled code	Does the tool require the code to be compiled before it can perform analysis on it?
6	Graphical representation	Does the tool display the graphical representation of the analysis result?
7	Selection of Desired Categories of Rules to be Applied	Does the tool allow the user to select specific categories of rules to check the code compliance with?
8	Severity categorization based on the nature of the violation (Blocker, critical, major, minor)	Does the tool specify the severity of violations in results (minor, major, or critical)?
9	No. of Standards for Code Compliance	Number of coding standards with which the tool checks the compliance of the code at hand
10	Is the Tool Optimized for Mission Critical Systems?	Does the tool predominantly check the compliance with rules defined for the development of mission-critical systems?

Table 2: Comparative Analysis

S. No.	Tool	Usability	Installation	No of Defined Rules	No. of Categories of Defined Rules	Requires Compiled Code?
1.	Ndepend	Normal [13, 14]	Easy [13, 26]	More Than 150 Default Code Rules	14	No
2.	PVS-Studio	Hard [23, 25]	Easy [23, 25]	450 Diagnosis Rules	29	No [23]
3.	Resharper	Easy [21, 26]	Easy [21, 26]	1700+ Code Inspections	10	No [21]
4.	Fxcop	Easy For GUI, Hard for Command Line [19, 25]	Easy [19, 25]	More Than 200	9	Yes

5.	Visual Code Grepper	Easy [18, 26]	Easy [17, 26]	Unknown	3	No
6.	Nitriq	Easy [20, 25]	Difficult [20, 25]	Over 40 Pre-Written Rules Defined	Unspecified	Yes
7.	Parasoft Dot Test	Easy [23, 26]	Untested/ Not Reviewed	Above 400	20	No
8.	Coverity Scan	Hard [16, 25]	Easy [16, 25]	Unknown	30	No
9.	sharpniZer	Easy	Easy	150	10	No

Table 3: Comparative Analysis

S.No.	Tool	Graphical Representation	Selection of Desired Categories of Rules to be Applied	Severity Categorization Based on Nature of Violation (Blocker, Critical, Major, Minor)	No. of Standards for Code Compliance	Is the Tool Optimized for Mission Critical Systems?
1.	Ndepend	Yes [14]	Yes	Yes [13]	Unspecified	No
2.	PVS-Studio	No [24]	No	No	3	Yes[24]
3.	Resharper	No [21]	No	No	Unspecified	No
4.	Fxcop	No	Yes	No	1	No
5.	Visual Code Grepper	Yes [26]	No	Yes	2	Yes [18]
6.	Nitriq	Yes [20]	Yes	No	Unspecified	No
7.	Parasoft DotTEST	Yes [24]	Yes	Yes	3	Yes
8.	Coverity Scan	Untested/ Not Reviewed	No	No	2[15]	Yes [15]
9.	sharpniZer	Yes	Yes	Yes	5	Yes

Due to successful realization of chief requirements for testing static code of mission-critical systems, sharpniZer is currently employed by NESCOM (a military research organization of Pakistan) to test the software (written in C#), embedded in some of the mission-critical and ammunition systems developed by the organization.

6. Future Work

The proposed tool currently encompasses 150 coding rules, obtained from standards (established for the development of mission-critical systems) such as MISRA, CERT, CWE, and JPL and also from Microsoft. The rules list can be extended to cover more rules and standards. Moreover, custom rules can be made

part of the tool that will allow users to define their own rules through a query language.

Refactoring is yet another feature that can be incorporated into the system. Through Refactoring, the violation will automatically be corrected in the code as the user clicks on the violated rule in the Results tab. Also, the tool can be expanded to cater to source codes of other languages along with C#

7. Conclusion

This paper discusses the burgeoning issue of overlooked bugs in static code of critical systems, and proposes a static code analysis tool, sharpniZer, that analyzes the code specifically written in the C#, and precisely identifies all the discrepancies and deficiencies in the source code as per the coding standards set for the development of mission-critical systems, enhancing the overall efficiency and reliability of the code.

Conflict of Interest

The authors declare no conflict of interest.

Acknowledgment

This research paper is a product of the effort of 5 authors, the references to whom are specified at the beginning of the article. All authors have read and agreed to the published version of the manuscript. The detailed roles of each were as follows:

Conceptualization: Rida Shaukat and Arooba Shahoor Data curation: Arooba Shahoor Writing—original draft preparation: Rida Shaukat Writing—review and editing: Dr. Sumaira Sultan Minhas Supervision, Ms. Hina Awan Funding acquisition: Dr. Kashif Saghar.

References

[1] A. Costin, "Lua Code: Security Overview and Practical Approaches to Static Analysis," 2017 IEEE Security and Privacy Workshops (SPW), San Jose, CA, 132-142, 2017.

[2] S. A. Fatima, S. Bibi and R. Hanif, "Comparative study on static code analysis tools for C/C++," 2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST), Islamabad, 465-469, 2018.

[3] T. Delev and D. Gjorgjevikj, "Static analysis of source code written by novice programmers," 2017 IEEE Global Engineering Education Conference (EDUCON), Athens, 825-830, 2017. DOI: 10.1109/TII.2016.2604760

[4] A. S. Novikov, A. N. Ivutin, A. G. Troshina and S. N. Vasiliev, "The approach to finding errors in program code based on static analysis methodology," 2017 6th Mediterranean Conference on Embedded Computing (MECO), Bar, 1-4, 2017.

[5] "The Patriot Missile Failure," Soundwaves, 2016. http://www-users.math.umn.edu/~arnold/disasters/patriot.html.

[6] "Lethal Software Defects: Patriot Missile Failure « Barr Code," Barr Code, 2014. https://embeddedgurus.com/barr-code/2014/03/lethal-software-defects-patriot-missile-failure/.

[7] "AT&T Corp. (American Telephone & Telegraph)," Ad Age, 15-Sep-2003. http://adage.com/article/adage-encyclopedia/t-corp-american-telephone-telegraph/98327/.

[8] "AT&T's History of Invention and Breakups," The New York Times, 13-Feb-2016. https://www.nytimes.com/interactive/2016/02/12/technology/att-history.html.

[9] R. Kirkov and G. Agre, "Source Code Analysis – An Overview", Cybernetics and Information Technologies, **10**(2), 2014.

[10] R. Plosch, H. Gruber, C. Korner and M. Saft, "A Method for Continuous Code Quality Management Using Static Analysis," 2010 Seventh International Conference on the Quality of Information and Communications Technology, Porto, 370-375, 2010.

[11] "The Explosion of the Ariane 5," Soundwaves. "The Patriot Missile Failure," Soundwaves, 2019. http://www-users.math.umn.edu/~arnold/disasters/patriot.html

[12] "ARIANE 5 Failure - Full Report," Safeware: System Safety and Computers, 2018. http://sunnyday.mit.edu/nasa-class/Ariane5-report.html

[13] "NDepend In Review," NDepend In Review - CraigTP's Blog, 2018. https://blog.craigtp.co.uk/Post/2017/12/17/NDepend_In_Review.

[14] C. N. D. W. www.ndepend.com, "NDepend Reviews 2018 | G2," G2 Crowd, 01-Sep-2018. https://www.g2.com/products/ndepend/reviews

[15] "27. Coverity Scan," 27. Coverity Scan - Python Developer's Guide, 2019. https://devguide.python.org/coverity/.

[16] Synopsys, "Coverity Reviews 2018 | G2," G2 Crowd, 22-Jul-2018. https://www.g2.com/products/coverity/reviews.

[17] Algaith, P. Nunes, F. Jose, I. Gashi, and M. Vieira, "Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools," 2018 14th European Dependable Computing Conference (EDCC), 2018.

[18] Nccgroup, "nccgroup/VCG," GitHub, 03-May-2016. https://github.com/nccgroup/VCG.

[19] FxCop, 2017. https://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx.

[20] M. Valdez, "Home," Home, 2017. http://marcel.bowlitz.com/.

[21] "Nitriq Code Analysis for .Net," Home : Nitriq Code Analysis for .Net, 2017. http://www.nitriq.com/.

[22] "ReSharper: Visual Studio Extension for .NET Developers by JetBrains," JetBrains, 2017. https://www.jetbrains.com/resharper/

[23] "Service Virtualization, API Testing, Development Testing," Parasoft, 2017. https://www.parasoft.com/.

[24] "PVS-Studio: Static Code Analyzer for C, C and C#," PVS-Studio: Static Code Analyzer for C, C and C#. https://www.viva64.com/.

[25] J. Novak, A. Krajnc and R. Žontar, "Taxonomy of static code analysis tools," In The 33rd International Convention MIPRO, 418-422. IEEE, 2010.

[26] H. Prähofer, F. Angerer, R. Ramler and F. Grillenberger, "Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application," in IEEE Transactions on Industrial Informatics, **13**(1), 37-47, Feb. 2017. DOI: 10.1109/TII.2016.2604760

[27] 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, 2018, 455-464, doi: 10.1109/IBCAST.2018.8312264.

[28] E. Ogheneovo, "Software Dysfunction: Why Do Software Fail?. Journal of Computer and Communications, **02**(06), 25-35, 2014. DOI:10.4236/jcc.2014.26004

[29] M. Faizan & P. Dhirendra. "software testing, fault, loss and remedies. **6**. 553-568, 2019. DOI: 10.1109/52.382180

[30] I. Gomes, et al. "An overview on the Static Code Analysis approach in Software Development." 2009.