Cyclomatic Complexity Analysis on Unity Visual Scripting, Unreal Engine Blueprint, and Unity C#

Ziyuan Guo

***Abstract*—Visual Programming Languages(VPLs) are languages that allow users to create programs by manipulating the elements of the language graphically instead of textually. With the current rise of visual programming, it is crucial to have a systematic study on the maintainability of some mainstream VPLs. To understand the maintainability of VPLs, I selected two 3D software development platform, Unity and Unreal Engine, and compared their VPLs options with the C# code from Unity. I selected Cyclomatic complexity to examine their maintainability and find out that the Unity Visual Scripting has the biggest Cyclomatic Complexity score among the three lanaguages.**

I. Background

Visual programming language(VPL) has been widely applied in various industries and have been demonstrated to have the potential for the improvement of the basic problem competence for coders with little experience in programming languages [3]. Recently, more and more programmers without formal software development training started to develop software applications, and it has motivated major companies such as Microsoft and Amazon to invest in tools and frameworks for the programmers with less formal programming experience. With this trend, the market for coders with no formal education in software development is expected to reach $21.2 billion by 2022.

VPLs are programs that create softwares by arranging graphical elements rather than textually specifying every element, which make programs to have a stricter syntax rule. Given this unique characteristic, they become a helpful tool for programmers with little experience in coding and make it easier for them to start learning software development. Therefore, VPLs play an important role in the future market of low-code development [4].

Among all the platforms that are widely used by end-users to develop software applications, Unity is one of them with huge popularity for 2D and 3D application development such as game, simulation or for virtual reality. According to their website, Unity has over 2.5 million registered developers and 500,000 monthly active users [2]. However, different from the widespread popularity of Unity, their VPL, Unity Visual Scripting(UVS), is not as popular. By applying World of Code [5] to look up UVS files on the GitHub, I only is able to find around 3000 UVS files on the internet. On the contrary, one of the main competitors of Unity, Unreal Engine, has over 15 million files according to a dataset extracted by Eng Kalvin [1]. This inconsistency, and with the rapid development of VPLs for low-code market, motivated me to perform a static analysis on the Unity Visual Scripting using the Cyclomatic Complexity (CC) and compare it with Unreal Engine and also the Unity's default scripting language C#. The Cyclomatic Complexity is a static analysis tool that is widely used by the software engineering community. It works by analyze the branching statements of a program, and is calculated with the formula given the control flow graph (CFG):

$$CC = Edges - Nodes + 2 * (EndNodes)$$

A lower CC value represents a better maintainability. I choose the CC is because of the graph nature of the VPLs, making an analysis of its control flow graph intuitive and convenient.

In this report, I will analyze related literature on VPLs, Unity and tools I have applied to perform this analysis in section 2. Then in section 3, I will demonstrate in-depth the methods I applied to be able to acquire the data and how do I acquire the control flow graph to acquire the CC value. Then in section 4, I will perform an analysis on the result and what is their significance. In section 5, I will talk about potential future work that could be done to further explore the topic. In section 6, I will conclude the project finding and propose some threat to the validity of the project.

II. Related Works

Currently there has been an increasing number of software applications developed by end users with no formal software development training. In this paper: "Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review", it provided an in-depth research on the overview of this "low-code" development environments in the current world [7]. According to this paper, VPL is one of the important tools for End-User Development (EUD) given its unique ability to allow users to develop applications with graphical elements, allowing beginners with little experience to have a better user experience, an easier entrance to programming, and to avoid syntactic errors that are common to beginner coders [2]. Moreover, according to the authors, the VPLs are being used by domains other than education and gaining more and more popularity with its ability to allow users to create tailor applications conveniently. This paper provided a throughout statistical analysis on the current research done on VPLs and analyzed 30 VPL tools being discussed in the papers and analyzed the tools from multiple dimensions.

Beside a statistical analysis, here is another paper that provided a study on the effect of VPL on learning outcomes. In this paper, the authors performed a study with 65 fifth-grade students. In this study, the students are asked to design science experiments with VPL applications based on Scratch.

This research demonstrated that with the used of VPLs, the students demonstrate a significant increase in ability to "Designing Algorithms", "Problem Solving Competence", and "Basic Programming Competence" [3]. This is a proof that VPL could be an effective tool especially for people with little experience in programming to gain a basic programming skill.

For the use of cyclomatic complexity, this paper provided comprehensive research on the C# code with cc, which is also performed by me in this project. In this paper, it applied cc, Halstead's volume, and line of code on various C# projects, and provided a detailed example on static analysis using cc [6].

III. Methodology

To perform an analyzation of the CC for UVS, Unity C#, and Blueprint, there are three major steps. As illustrated in *Figure 1*, the first step is to collect the dataset for the three types of languages. Then with the dataset, I will process them into control flow graphs. The last step is to use the control flow graphs to calculate the CC for each of the programs.

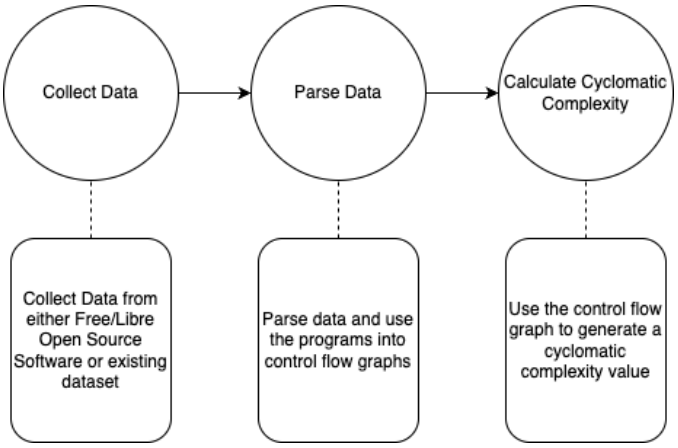

Fig. 1. Overview of Methodology

A. Part 1: Data collection

1) Unity Visual Scripting: I have previously acquired UVS files through my previous work for Professor Hindle by applying the WoC platform. To acquire UVS code, I have attempted many different ways to look for the UVS files. The UVS files are asset files for a Unity project, and it has extensions of ".asset". However, the challenge I faced during the data extraction process for UVS is that in Unity, there are various kinds of asset files for different purposes, and they all end with ".asset" extension. Moreover, the number of non-UVS ".asset" files outweigh the UVS files in a ratio more than 1000 : 1. To effectively collect UVS files without the data pollution of other asset files for Unity, I used the method of filtering out all the files end with a ".asset" extension and use WoC's showCnt() function to filter the content of the files and look for "Unity.VisualScripting" in the file. This required a large computation and currently I haven't gone through the entire database of WoC. For this project, I used 133 UVS files that I found using this method.

2) Unity C#: For the Unity C# code, I collected them using WoC with similar approach. The C# in Unity doesn't have a unique extension that's different from regular C# code. To effectively filter out the Unity C# files, I noticed a distinct pattern in Unity C# code that by default they are stored in a directory named "Assets/". Therefore, I set up the query to find files with file names(including directory) that contain both "Assets/" and ".cs". With this set up, I am able to collect a list of Unity C# codes. Then I applied a code to download them from GitHub which is similar to the UVS codes.

3) Blueprint: To understand the data extraction of the Blueprint graphs, it is necessary for me to introduce the data structure of a Blueprint graph. Blueprint graphs are consisted of nodes and edges, where each end point of edges is called a "pin". This pin plays a crucial rule given that it provides information of node type, and edge direction.

For the Unreal Engine Blueprint files, I queried from the dataset titled: "Under the Blueprints: Parsing Unreal Engine's Visual Scripting at Scale" that is published by Zenodo [1]. This dataset contains data of over 15 million Blueprint files and have various tables organized in SQL format. However, this database doesn't provide a direct way to access the content of graphs. To generate a list of nodes and edges for a single Blueprint graph, I need to query through various tables.

The first step of the data collection is to access the "graph" table to randomly select 1000 graphs and their unique identifier. Then I go into the "pin_linkedto_pin" table to acquire the "pin" of the edges. The database doesn't directly provide the node information for the two different pins, so I need to use the pins information to acquire the node information from the "pin" table. With the "pin" information, I acquire the node information in the "node" table. After extracting all the information, I need for the graph, I then organize the data into a JSON format and store it locally.

B. Part 2: Data Parsing

After acquiring the data, the next step is to generate a control flow graph for the calculation of CC. Different from the previous sequence of data collection, I am going to talk about the Unity C# control flow graph generation first given it is distinctly different from the other two languages as a textual language.

1) C#: To acquire a control flow graph generation for the C# code, I applied the .NET Compiler Platform, Roslyn, for the C# codes. The Roslyn is able to compile the C# code and then generate a syntax tree for the provided program. However, it is not trivial to use the generated syntax tree to create the control flow graph. With the syntax tree, the next step is to transform it into a SyntaxList of "statements". The .NET framework provided a comprehensive library with various classes implemented for different types of statements. The base class for all types of statements is "StatementSyntax". The parser takes a list of "StatementSyntax" that represent the whole program it needs to parse, then uses a for loop to go through each of the statements.

Given that the end goal of this parser is to generate control flow graph, I only need to focus on four types of branching statements with classes: “IfStatementSyntax”, “WhileStatementSyntax”, “ForStatementSyntax”, and “SwitchStatementSyntax”. If a statement is not one of those syntaxs, then I only add a node for that statement block and continue to the next statement. Given that only branching statements separated different blocks of statement, the C# the compiler organized the whole block between branching as one statement, so that no redundant node will be added for a sequence of non-branching code lines.

For each type of the branching statement syntax, I handle a logic flow that builds a control flow graph by adding nodes and adding edges, which is logically similar to the algorithm for UVS and Blueprint. The .NET framework provides corresponding attributes for each of the four-branching syntax. For example, for the IfStatementSyntax, the statements for if-then part are contained in ifStatement.statments, and the statements for if-else part(if they exist) are contained in ifStatement.Else, making them convenient to access. The parsing function is recursive, and by calling them on the bodies of each of the branching statements(such as if-then body, for-loop body), I am able to generate control flow graphs for the programs.

However, there is a major challenge for the generate of the control flow graph is that the Roslyn require a compilation for it to work and the Unity C# code often contains various custom classes making it unable to compile. The currently solution is to artificially replace the logic of the C# classes with equivalent logical statements, and run the CFG generation program on the translated file.

2) UVS and Blueprint: The UVS and Blueprint are VPLs composed of nodes and edges. However, for each node it could take multiple inputs or have multiple outputs, with non-constant for each type of nodes. However, after analysis of those two VPLs, I discovered an approach to directly use the VPLs to generate the control flow by focusing on only the branching nodes.

The approach I took is to only select four types of nodes for the VPLs: if node, for node, while node, and switch node and use its information to create a control flow graph. I created a node class for the information storage of the VPLs’ nodes and a CFG_node class only for the creation of the control flow graph. I realized that for non-branching node, there is only one possible edge to continue the program, therefore, by traversing through the nodes and perform operations base on the branching nodes I am able to efficiently construct the CFG.

For the UVS, they provide a default set up for different types of nodes. For example, for the for node, there are three potential outputs: body, index, and exit. This output status is recorded at the edge of the UVS graph. After gaining a more in depth understanding of the framework’s set up, I realized the method I should take to acquire knowledge of each node’s branching type. For example, if a node is from a non-branching node, it has no branching type. However, if a node is connected by an edge from a for-loop node, and the edge is type “body”, then this node’s branching type is loop-body. With that information, I adjust the graph representation into an adjacency list representation where for each node, I have a parameter where it contains all of its child nodes and its child nodes’ branching type.

Algorithm 1 Algorithm for UVS CNF Generation

```
Change data of nodes and edges in a adjacency list form
Input: Starting node, the Control Flow Graph object, and last node
while The node's child node is not empty do
    if The node is an If node then
        Create a after-if-node
        if It has two child nodes then
            Call this recursive function on both nodes
            Add the nodes and edges on the CFG object by connecting them to the after-if-node
        else
            Call this recursive function on one node
            Add node and edges on CFG object by connecting them to the after-if-node
        end if
    end if
    if The node is an For-loop or While-loop node then
        For body node, call recursive function and add edges to loop back to the loop node
        For the exit node, simply call the recursive function on it and continue
    end if
    if The node is a Switch node then
        Create a after-switch-node
        for each of the child node do
            Call the recursive function on the node
            Then after the node, add edge to connect back to the after-switch-node
        end for
    end if
    Move to the next node
end while
```

Another information I need to acquire for the graph is where to start. To acquire the starting nodes, I loop through the edges and find the nodes that don’t have any other nodes connected to them. Then I filter out three implicit starting nodes: ”Unity.VisualScripting.Start”, ”Unity.VisualScripting.Update”, ”Unity.VisualScripting.FixedUpdate” among those starting nodes. It is worth mentioning that it is possible for a UVS file to not have a starting node. Then this file won’t be process.

With the starting nodes, and the branch type of every node, I created a recursive function that starts with the starting node and then walks through the graph. If a node is not a branching node, it will only have one output, and the program will continue without doing anything. However, if the node is a branching node, the program will go through every single of its child nodes and check their branch type. Then with the different branch type, I would create a control flow graph based on the node type and branch type. The overview of

the logic can be seen at *Algorithm 1*.

For the Blueprint, I perform a similar logic to acquire the control flow graph. The database of Blueprint doesn't explicitly label for-loop nodes and while-loop nodes. To acquire the loop node, I have to acquire it from the pin information of a particular node. Therefore, there is an exact step to label the loop-node with the edge's information. For the recursive function, the logic is overall similar beside that I have combined the for-loop and while-loop as one type of node called loop-node.

C. Cyclomatic Complexity Calculation

After acquiring the CFGs, it is trivial to calculate the CC with the formula mentioned in Section 1. After calculating the number of nodes, and edges in the CFGs, I calculated the number of end nodes by finding nodes with no output edges. Then I calculate the CC = Edges - Node + 2 * (EndNodes).

IV. Analysis

Given the time limitation of the research, I only analysed the data on its cyclomatic complexity.

Type	Total Graphs/Programs Analyzed	Average CC
UVS	67	8.746
Blueprint	292	3.610
Unity C#	43	4

TABLE I
TABLE FOR THE CYCLOMATIC COMPLEXITY RESULT

For the UVS, I used the files I had extracted at the point I started this research with the total number of 133 UVS files. Among those files, some of them doesn't have the implicit starting nodes that I have mention in section 3. Overall, I am able to calculate the CC on 67 graphs with an average CC of 8.746

For the Blueprint, I randomly selected 1000 graphs from the database. While performing the CC calculation, I faced many difficulties. The first thing I discovered is that there are various Null values in the blueprint dataset I am using. They Null could even exist in nodes of edges which I have to skip through every graph that contain a Null edge node. The next thing I discovered is that there is a type of node called the Knot node, which could generate a loop in the graph and create an infinite loop for my recursive function. Moreover, this Knot node doesn't contain functional purpose in the graph, but only act like a supporting tool to make the development more convenient. Therefore, I also skip all the graph that contain a Knot node. The last challenged I faced is that in the database, I discovered some rare edge cases where there is an inherit loop in the graph that is not related to the Knot node. For example, graph with uuid: "ba1396f7-2df2-4235-baca-233a96e0d05d" has a loop between a SpawnActorFromClass node with a CallFunction node. After filtering out all those graphs, I analyzed a total number of 292 graphs, with a total CC of 1054, and have an average CC of 3.610.

For the Unity C#, given the challenged I faced on the compilation issue, I have to manually parse the file to find equivalent branching logic for the C# files. Due to this major limitation, I only randomly selected 10 files, and then received a total CC of 174 with 43 programs. The average CC per program for Unity C# is 4.000.

Among all the programming languages, UVS has the highest average CC with 8.746. Between the two VPLs, UVS has twice higher of a CC compared to the Blueprint with only 3.610 per graph. This demonstrated that UVS has a significantly bigger maintainability issue compared to the Blueprint, which could be an explanation for the lack of usage of the UVS compared to the Blueprint. The Unity C# on the other hand, has a slightly higher CC compared to the Blueprint but also is less than half of the CC of UVS. Given that they are part of the same framework, this could be due to the textual format allow for a more modular design with more sub-functions with little cyclomatic complexity, making the average CC to be lower.

V. Future Work

The first future work that could be developed upon this project, is to figure out a way to systemically calculate the CC for Unity C# file without manually translate the original raw files into a compilable format. Due to the limitation of time, I am unable to come up with a solution to this issue. This could greatly increase the number of Unity C# code being analyzed as a comparison to the UVS and Blueprint.

For the further development of this project, more static analysis tool could be applied to this work to provide a more comprehensive analyzation of the maintainability of the VPLs. Given that existing static analysis tools such as Line of Code are not applicable to the VPLs, more research on the metric for the analyzation for the VPLs are also needed.

VI. Conclusion and Threat to Validity

With data collected through WoC and online dataset, I am able to perform a static analysis on the Unity Visual Scripting code, Unreal Engine Blueprint, and Unity C# codes with cyclomatic complexity as the metric. For the UVS, I have calculated an average CC of 8.746 per graph with a total of 67 graphs. For the Blueprint, I have calculated an average CC of 3.610 which is the lowest among the three with a total 292 graphs analyzed. For the Unity C#, the average CC is 4 with 43 functions analyzed. The UVS has the highest CC making it the worst in maintainability among those three languages.

The biggest threat to validity is for the testing. For the UVS control flow graph generation, I am able to create UVS files from the Unity on my local machine to test the branching statements given its output file is human readable. For the Unity C#, I am also able to write test in C# for the testing of CFG generation. However, for the translation from original C# code to compilable C# code with equivalent branching logic, currently there is no way to test it. For the Blueprint, I am not able to write Blueprint code for testing given that the output of the code is not humanly readable and the sources of Blueprint files are from the dataset only. Currently for the Blueprint, I am only using visualization of existing Blueprint graphs I generated and manually comparing with the CFG I generated to make sure the parser for CFG is working properly.

REFERENCES

[1] ENG, K., AND HINDLE, A. Under the Blueprints: Parsing Unreal Engine's Visual Scripting at Scale. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)* (2025), IEEE.

[2] JOST, B., KETTERL, M., BUDDE, R., AND LEIMBACH, T. Graphical programming environments for educational robots: Open roberta-yet another one? In *2014 IEEE International Symposium on Multimedia* (2014), IEEE, pp. 381–386.

[3] KALEMKUŞ, J., AND KALEMKUŞ, F. The effects of designing scientific experiments with visual programming language on learning outcomes. *Science & Education* (2024), 1–22.

[4] KUHAIL, M. A., FAROOQ, S., HAMMAD, R., AND BAHJA, M. Characterizing visual programming approaches for end-user developers: A systematic review. *IEEE Access 9* (2021), 14181–14202.

[5] MA, Y., BOGART, C., AMREEN, S., ZARETZKI, R., AND MOCKUS, A. World of code: an infrastructure for mining the universe of open source vcs data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (2019), IEEE, pp. 143–154.

[6] MATHUR, B., AND KAUSHIK, M. Critical evaluation of maintainability parameters using code metrics. *International Journal of Computer Science and Information Security 14*, 7 (2016), 131.

[7] MOHAMAD, S. N. H., PATEL, A., TEW, Y., LATIH, R., AND QASSIM, Q. Principles and dynamics of block-based programming approach. In *2011 IEEE Symposium on Computers & Informatics* (2011), IEEE, pp. 340–345.