



GryphonSharp v1

Anton Sukachev, a.sukachev@lancaster.ac.uk
School of Computing and Communications
Lancaster University

Coordinator: Abe Karnik (a.karnik@lancaster.ac.uk)
Research group: Human Computer Interaction

Contents

1	Artefacts of the research	2
2	Introduction	2
3	Background	9
3.1	First abstract language	9
3.2	Object-Oriented Programming	10
3.3	First Visual Programming Environments	11
3.4	Minecraft and Redstone	11
3.5	Transpilers, an upgrade to existing languages	11
3.6	Blockly and general-purpose VPE	11
4	Motivation	11
5	System Design	12
6	Language Design	12
6.1	Object-Oriented-Programming Design	13
6.2	Blockly and OOP	13
6.3	StarCraft II inspired Behaviour Oriented Design	13
6.4	Node-like Shader Editor Design	15
6.5	Unreal Engine Blueprints inspired Design	15
6.6	Stateless Node Editor	15
7	Implementation	15
7.1	Intermediate GryphonSharp Representation	15
7.2	Incomplete Components	15
7.2.1	Language Server	15
7.2.2	G#-to-C# Transpiler	17
7.2.3	Visual Node Editor	18
8	Discussion	19
9	Appendix	19

1 Artefacts of the research

2 Introduction

Since the first computers, there has existed a strong need for a high-level programming language that would allow users to quickly and efficiently code applications and systems that they need for their businesses, personal projects or even games.

Motivation of research. Textual languages are generally flawed. A typical source code of a program written in any textual language contains a lot of unnecessary information that is essential for computer to understand what programmer wants to achieve, but otherwise useless to the coder. These key features of any textual language are commonly referred to as 'Syntax of a Language'. A syntax is a set of special characters and keywords that help programmer declare their intent by expressing their algorithm through syntax.

Let us use an example, a programmer might want to create a simple program that outputs, first, a sum of 2 numbers, then a multiplication of those 2 numbers. To implement this algorithm, a programmer needs to declare the two numbers they want to add up, then he must assign those numbers to variables, a number per a variable. At this point, there is already a redundancy with said variables. Existence of variables is vital in textual computer source file, it plays a role of a 'symbolic link' to a data. Consequently, for a variable to be referenced, it must be addressed by a declared name, refer to example listing 1.

```
int a = 12;
int b = 9;
int sum = a + b;
Console.WriteLine(sum);
int mult = a * b;
Console.WriteLine(mult);
```

Listing 1: Sum and Multiply

For any textual language this syntax is essential to writing a program. However, naming variables is quite abstract and there is no rules, unlike languages' syntaxes, on variables' names. Naming conventions were introduced and change from project to project, from company to company. In fact, variable naming in data science, became so absurdly difficult, people had to start educating programmers to invent proper names for their variables.¹² In contrast, visual languages make it much easier to deal with variables because there is no need to declare names, unless it's a field of an object. Refer to an example of the same program using visual representation in figure 1.

¹<https://towardsdatascience.com/data-scientists-your-variable-names-are-awful-heres-how-to-fix-them-89053d2855be>

²<https://betterprogramming.pub/useful-tips-for-naming-your-variables-8139cc8d44b5>

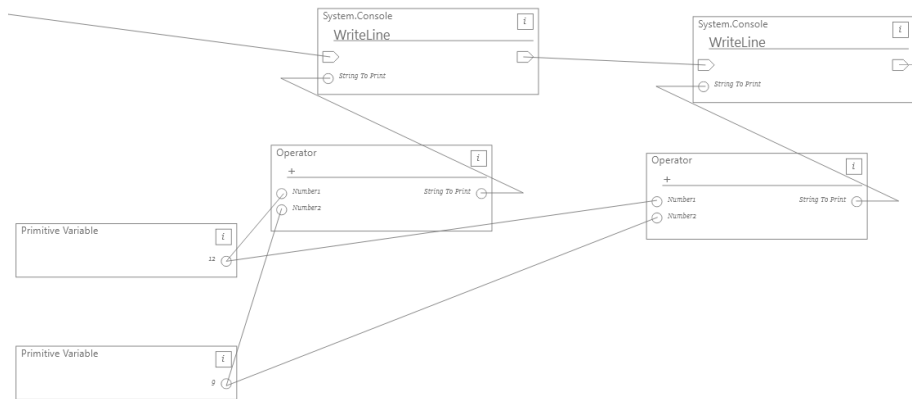


Figure 1: Example Visual Graph

Those are:

- **Fragile base class**

The problem comes from the ability to make changes to base class, which require all inheriting classes to be inspected for resulting faulty behaviours. For example here is example code for parent and child class in listing 2.

```
public class Parent{
    public virtual void DisplayToConsole(string message)
    {
        Console.WriteLine(message);
    }
    public virtual void LogToFile(string message){
        // TODO: write to file
    }
}
public class Child : Parent{
    public override void DisplayToConsole(string message)
    {
        LogToFile(message);
    }
}
```

Listing 2: Fragile Parent Class

The example program above is pretty straight forward - the Child class wants to log all requests to file instead. However, there is a problem of a fragile class in this program. Once Parent class was requested to also display message to console, so just 1 line modification was performed in the Parent class, which now completely halts the program, by locking it in the infite loop, see listing 3.

```
public class Parent{
    public virtual void DisplayToConsole(string message)
```

```

    {
        Console.WriteLine(message);
    }
    public virtual void LogToFile(string message){
        DisplayToConsole(message);
        // TODO: write to file
    }
}
public class Child : Parent{
    public override void DisplayToConsole(string message)
    {
        LogToFile(message);
    }
}

```

Listing 3: Fragile Parent Class Backfired

This seemingly harmless and thoughtful change to Parent class, suddenly breaks all existing programs that derive from this class if they decide to override the behaviour of DisplayToConsole with LogToFile.

- **The diamond problem**

The data model of OOP is based on inheritance. There is an example with fruits and decay. Naturally, all fruits slowly decay, they might have different rate and the rate might be affected by the environment. Although this might sound reasonable, there is another example with aluminum and decay. Aluminum is a metal, which will also decay over time. Just like the fruits, it has a different rate and slightly different conditions before it begins to decay. Furthermore, the decay operation, or 'function', is slightly different to that of a fruit. Despite that, both aluminum and fruit decay over time. The diamond problem, however, takes this matter a step further and asks a fundamental question 'is this inheritance even legal?' The following example in listing 4 will address legality of inheritance.

```

public class Student{
    public int StudyYear;
    public bool CanOpenDoor(int roomId){
        switch (roomId){
            case 1:
                return true;
            case 9:
                if (StudyYear > 2)
                    return true;
                return false;
            default:
                return false;
        }
    }
}

public class Professor{
    public bool CanOpenDoor(int roomId){
        switch (roomId){
            case 1:
            case 2:
            case 5:
            case 9:
                return true;
            default:
                return false;
        }
    }
}

```

Listing 4: Inheritance Problem

In Student-Professor example, is it safe to say that a professor is a subclass of student? But if professor is child of a student class, the extra data such as 'StudyYear' would be obsolete and any methods that the class might have which are meant exclusively for students would somehow need to be locked.

- **Encapsulation**

Another problem with OOP is encapsulation. The following example in listing 5 illustrates the issue.

```

public class Student{
    private List<int> OpenedDoors;
    public void OpenDoor(int doorId){
        // opens a door
        OpenedDoors.Add(doorId);
    }
    public List<int> GetOpenedDoors(){
        return OpenedDoors;
    }
}

```

Listing 5: Encapsulation Problem

When requesting a list of opened doors through method `GetOpenedDoors`, list of items is returned. Despite the variable being private, the returned list is, in fact, a reference to that list. This means that modifications to this returned list, which is a private variable in the class, will also affect class's variable `OpenedDoors`.

There are also C# specific problems which we also go through:

- **Little 'useful' accessibility modifiers**

In C# there are multiple accessibility modifiers. All of modifiers, along with their access privileges, are displayed in figure 2.

Caller's location	public	protected internal	protected	internal	private protected	private
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

Figure 2: C# Access Modifiers

Unlike Java, C# doesn't have a modifier to restrict access to a certain namespace, in fact, the only namespace-like restriction could be achieved through declaration of another assembly, in other words creating another C# project and explicitly referencing this project. This is not at all useful, especially when projects are tied together and some access restrictions is absolutely necessary to keep codebase organized. Meanwhile Java addresses this problem by introducing package, which behaves like namespace. When the package is declared, it is enforced in the filesystem, meaning a file.java under folder some/folder must have a 'package some.folder' declaration in the source file, otherwise the entire java project will result in compilation error. Java offers fewer access modifiers but with greater impact due to existence of packages, those along with access levels can be seen in figure 3.

Access Levels

Modifier	Class	Package	Subclass	World
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
<code>private</code>	Y	N	N	N

Figure 3: JavaAccess Modifiers

- **Not enforced namespaces**

As previously stated, Java's namespaces, which are called packages, are enforced in the filesystem. When creating item.java file under com/apple folder, Java compiler will force 'package com.apple' declaration. C#, however, doesn't enforce any namespaces onto the filesystem, which, consequently, creates a messy source code environment. Furthermore, unlike in Java, source code files in C# are not enforced at all, meaning a single file can contain the codebase of the entire project, declaring namespaces one by one and classes within those namespaces, like in the example listing 6.


```

namespace Root{
    public class Clazz{
        // some data

        // some methods
    }
}
namespace System.Numerics{
    public class Clazz{
        // some other data

        //some other methods
    }
}

```

Listing 6: Example File thisisok.cs

Having ability to declare namespaces out of place, randomly and for any namespace hurts readability of the code. This makes a single namespace be composed by multiple files, quite literally, hidden across the project and may be even across assemblies.

- **Partial Classes**

Another problematic feature of the language is partial classes. When partial class is declared, this means it is can be completed somewhere else in a different file, possibly within a different folder.

- **Multiple implementations of a Solution**

When there is a problem, there is an algorithmic solution. However, in case of C#, the same solution can be implemented a thousand ways, but not because of the Turing-completeness of the language, but rather the design of the language. In the following listing 7.

```

public class Clazz{
    public int _Id;
    public int Id {get => _Id;}
    public int GetId(){
        return Id;
    }
    public int GetId1() => _Id;
}

```

Listing 7: Example C# File notok.cs

In the above example, the problem is the following: 'Retrieve a field value _Id'. The solution of the problem is a simple one - simply get the field value. However, an '_Id' variable can be retrieved using 1 of 4 ways, using syntax of the language, in other words, the solution can be implemented in 4 ways. When the complexity of the problem grows and the complexity of the

solution grows exponentially fast, the implementation should be limited to help programmer stay on course to the solution instead of creating extra overhead (in form of system design decisions) by providing extra implementation tools which impose extra decisions to be made by the programmer. Consider the example in listing 8.

```
public classClazz
{
    public int SomeVar { get => SomeVar - 1; private set {
        SomeVar = value; } }
    public void SetSomeVar(int value) =>
        SomeVar = ++value;
}
```

Listing 8: Simple get/set Example C# File reallynotok.cs

A simple getter/setter variable example is quite difficult to read and understand what the program is trying to achieve instantly. As the codebase grows and evolves, it becomes increasingly difficult to read the code.

In conclusion, features and design decisions in the list above, make C# virtually impossible to navigate through without a proper IDE such as VSCode, Visual Studio, Eclipse, Rider or others.

This research has a few distinctive objectives it attempts to address:

- A set of questions in the section Motivation.
- Proposes a design for creating simple programs through G# language.
- Presents implementation of G# VPE.

// Aims of this research

3 Background

In this section, we will discuss how programming languages became what they are now, later, how they became abstract to accommodate extensibility. Subsequently, we will discuss Object-Oriented Language C# which is the language to which G# will transpile. It is important to understand why visual environments started to appear and what purpose did they serve. Finally, we will acknowledge and briefly discuss existing visual programming environments Blockly and

3.1 First abstract language

The first ever abstract language appeared around 1969-1973 and it rapidly replaced assembly language which offered 'one-at-a-time' instruction in a semi-readable for human format. The C Programming Language quickly became a

legend among programmers, especially those working on a very first UNIX operating system[15]. The C programming language introduced a wide range of abstract paradigms that programmers could utilize to swiftly develop readable programs where they can orient freely and extend to their needs. Some of the examples of such paradigms include: struct, enum and array indexing with the following syntax:

```
variable [2]
other_variable [0]
```

Listing 9: C array indexing

Programs that would otherwise require hours of revision and coding could be written in a matter of minutes, if not seconds. After C language, a major abstract language at a time, was standardized by ANSI, it became the primary language for most computing systems.

Despite being a standard for most, if not all, computing systems, the language was platform-specific, meaning different platforms would have specific implementations, hence compiled executables could only be run on the platforms for which executables were explicitly compiled. That way executables compiled for Windows would not run on MacOS/Linux Unix systems.

Unlike most, C language offers naked address referencing, meaning any given number could be converted into address in memory and backwards. This low level access to actual computer memory is strongly tied to memory leaks, critical security leaks, and even unwanted overflows that, if left unchecked, could violate kernel space from user space, which in turn could trigger kernel panic.[11] Not only extremely low-level access that the C language provides poses severe vulnerability risk to the entire operating system, but it also requires significantly advanced memory management systems written by the applications' developers. In contrast, languages such as Java or C# have built-in memory management that automatically frees memory when instances no longer accessible. For these reasons C language simply did not offer enough abstraction, security, and flexibility to aid development of complex programs.[16]

3.2 Object-Oriented Programming

Object-Oriented Programming, henceforth OOP, was primary focus of early stages of research because it is a paradigm that is used by a target language C#. Expressing OOP data model, even in native, textual format proves to be troublesome. There are a few design flaws which heavily contribute towards rejecting OOP data model.

For this research, a custom language has to be developed to accommodate all the needs of the visual language. This visual programming paradigm is the primary focus of this research. In the remaining subsections every design of the language will be discussed and explained in-depth. By the end of each subsection, there should be a clear understanding why those decisions came to be and also why the design was rejected.

3.3 First Visual Programming Environments

Visual Programming Environments (henceforth VPE) were introduced for educational purposes first appearing in 2003 for Intel's Computer Clubhouses where young could learn computer technologies. Original designs of scratch were solely intended for young by adding visual programmable elements that manipulate media on screen.[6] Scratch UI uses sturdy design principles that achieve it's ultimate goal i.e. Single Window with multiple tabs ensures there is always visible primary components such as scene or code.

3.4 Minecraft and Redstone

Minecraft is a computer game that was introduced in 2009. Minecraft opened a new genre - Sandbox. Sandbox genre, effectively, allowed players to manipulate their environments and shape them visually how they see fit. Game introduced first few 'quests' that introduced player to the game, which then left player to explore world on their own. Minecraft is extremely successful, but not for a straightforward 'because of this and that' reasons. [1] Why Minecraft game that has nothing to do with teaching and programming[1] (at the very least initially, it was designed for self-entertainment by creating stories using creative tools that the game provided[2]) has a communities which focus heavily on implementing complex computing systems (even entire CPU architectures in Minecraft using nothing else but plain single-thread Redstone).[13]

3.5 Transpilers, an upgrade to existing languages

Transpilers take source code and compile into other source code. This is 'transpilation'.[4]

3.6 Blockly and general-purpose VPE

Blockly was a good take on general-purpose VPE. It uses transpiler principles to interpret visual representation of a code.[8] It support wide range of target languages, however, currently, without specifically designed environment, Blockly, out-of-the-box, only supports few paradigms: functions, variables, arithmetic operations, flow control (loops, if statements) and a few more basic primitives like lists and color.³ Blockly, natively, is only a set of theories and findings that build on top of preexisting VPE 'Scratch'. [14] Despite it having rich and extensible documentation, Blockly is just a set of tools for creating VPEs.[18, 3]

4 Motivation

How C achieved it's purpose? Why has it became so popular?

How C represented basic programming constructs (for, while loops, if statements)? When and why it fails?

³<https://developers.google.com/blockly>

Why Scratch became so popular? How Scratch aims to achieve their aims?

Scratch is limited by Scratch VM.

Transpilers are interesting take on simplifying development. Any attempts to generate code from Visual Programming Environments - refer to Blockly.

// merge into introduction

5 System Design

Previously mentioned transpilers were 'the next step' towards higher languages that might provide task-specific abstraction level and paradigms for easy development of said tasks.[10] For example, many game studios that started working in limited language such as Lua had to either rewrite their entire game on a different engine under different programming language. However, since introduction of transpilers that use higher-level or simpler languages to transpile to low-level or complex languages, one of the most popular examples (in addition to mentioned Typescript-Javascript transpiler) being CSharp.lua⁴ which takes c# code and transpilers it into Lua code retaining most, if not all original language features such as reflection, polymorphism, encapsulation etc.

In this project, it was decided to utilize offline transpiler. This decision is motivated by strong presence of high-level heavily optimized languages such as Java, Javascript, C#, Python, C++ and others. In contrast, the other choices were: Compiler and Interpreter. Interpreter is an algorithm which uses calls and translates them one by one into bytecode, all of this 'interpretation' is happening at runtime and therefore creates huge overhead because every instruction the interpreter executes, essentially has to be looked up before it can be executed. Look up of interpreters is *usually* capped at RAM latency, rather than CPU cache latency, meaning look up of instructions happens in RAM in worst case scenario. Compiler both JIT and AOT require significant investment of time to design and implement from scratch. For purposes of this research, creating custom G# to machine code compiler would be infeasible and would simply go outside of this project's scope.

6 Language Design

Currently there exist over 10 programming paradigms (in order of popularity): Object-Oriented Programming, Functional Programming, Imperative Programming, Procedural Programming, and others. This research focuses on the Object-Oriented Programming because the target language is C#, which is Object-Oriented. However, in the next section it will be discussed why such language design was

⁴<https://github.com/yanghuan/CSharp.lua>

6.1 Object-Oriented-Programming Design

6.2 Blockly and OOP

A popular visual programming solution for real-world problems is Blockly, a visual programming environment based on Scratch. Example of Blockly is in figure 4.

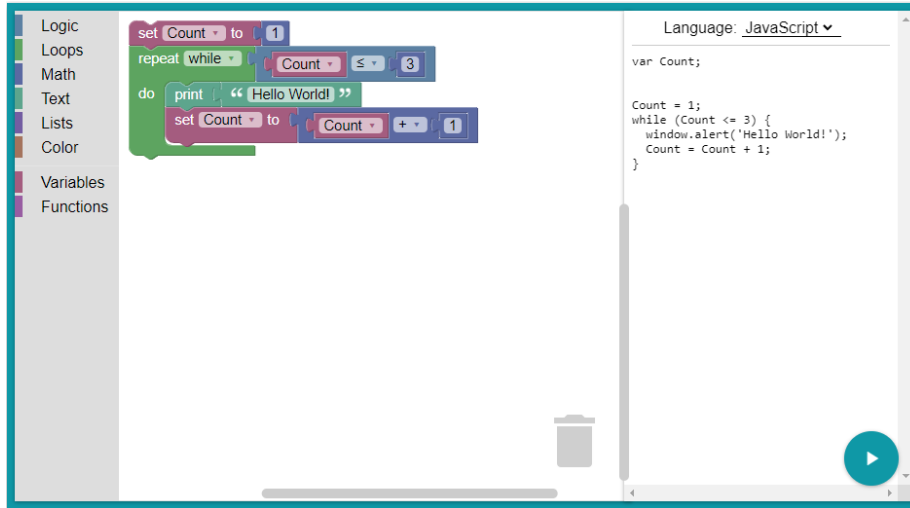


Figure 4: Galaxy Editor

A few articles [8, 14] pointed out some wrongs with Blockly, also, consequently, suggesting how some of the problems could be achieved. One of the problems is data, both variables and entire objects do not fit in block-based model that Blockly offers.
[17] [7]

6.3 StarCraft II inspired Behaviour Oriented Design

Initially, the design was inspired by how StarCraft II's units behaviour builds up in the game. In the game StarCraft II, it is possible to create custom scenarios, called 'maps', using Galaxy Editor. The core gameplay of StarCraft II is also built using Galaxy Editor but with custom signing algorithm to make compiled scenario have 'Made by Blizzard' signature. Galaxy Editor screenshot is in figure 5. In the background is the primary editor, where the core elements of the map can be modified such as terrain, unit placements, regions for triggering events etc. In front is the data editor, editor that the research is interested in. A close up of the editor and one of unit's behaviour lists is in figure 6.

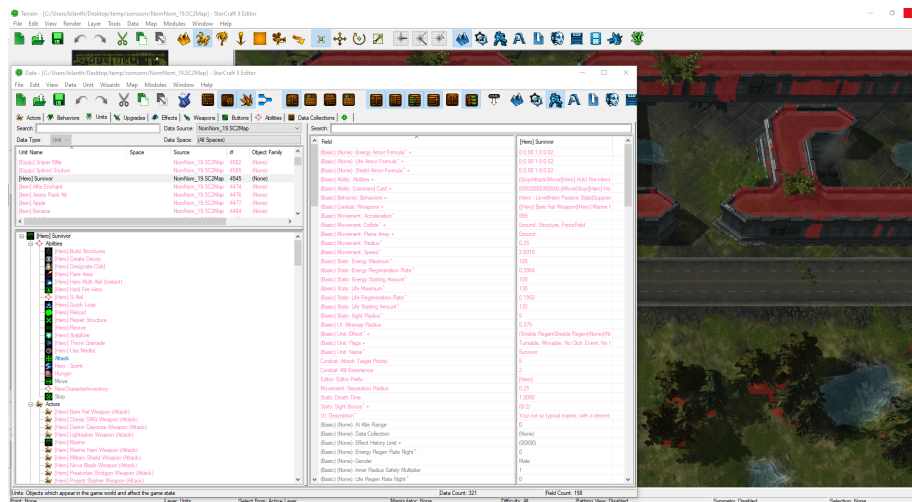


Figure 5: Galaxy Editor

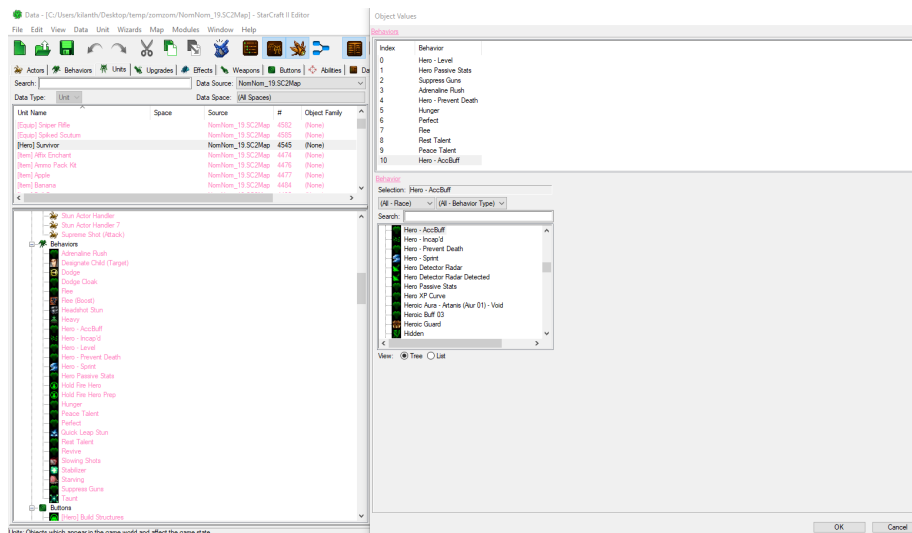


Figure 6: Galaxy Data Editor

What is interesting is how any unit in the game - a player's primary character, enemy unit, or otherwise a neutral merchants that sells items have an array of behaviours that control unit's parameters, override their functionality or store custom data. For example, a unit might have a 'veterancy behaviour' enabling it to receive experience. Upon reaching a certain threshold, the unit will gain a new level, which might provide extra combat parameters or give an extra ability - all of it is highly configurable. This kind of behaviour-enabling design is

what allows objects to have multiple behaviours that would otherwise contradict OOP's data model.

Research into Behaviour-Oriented-Design, henceforth BOD, [9] has been already conducted and provided some results 'good potential', although, the prototype was 'less-advanced' due to time constraints.

6.4 Node-like Shader Editor Design

// write up[12, 5]

6.5 Unreal Engine Blueprints inspired Design

// write up

6.6 Stateless Node Editor

// write up[14]

7 Implementation

7.1 Intermediate GryphonSharp Representation

Intermediate G# Representation (henceforth G# IR) is a JSON-structured file containing code, data and metadata for both Transpiler and Node Editor.

7.2 Incomplete Components

Following subsections will include incomplete or unfinished components of this project. Generally, just like in the proposal of the GryphonSharp, the research was attempting to produce: Language Server (GryphonSharp-Overwatch), G#-to-C# Transpiler (GryphonSharp-Transpiler), and Visual Node Editor (GryphonSharp-vscode). Every component is detailed and documented in the following subsections.

7.2.1 Language Server

Originally, Language Server, codenamed 'Overwatch', was planned to be implemented with VSCode language server protocol.⁵ The Overwatch language server handles multiple tasks listed below.

- **Type-Safety Enforcer**

When user tries to connect nodes in the node editor, types of connectors are evaluated through Overwatch server request. Overwatch, being connected to Omnisharp server, will evaluate validity of type parsing and

⁵<https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>

return boolean with TRUE value if parsing type is 'legal' for the connector, otherwise it will return FALSE, which will instruct node editor to highlight connection is 'illegal'. Highlight can either allow connection with red-error cross across illegal connection (can be seen in figure 7) or prevent connection altogether and raise an error through message box (can be seen in figure 8).

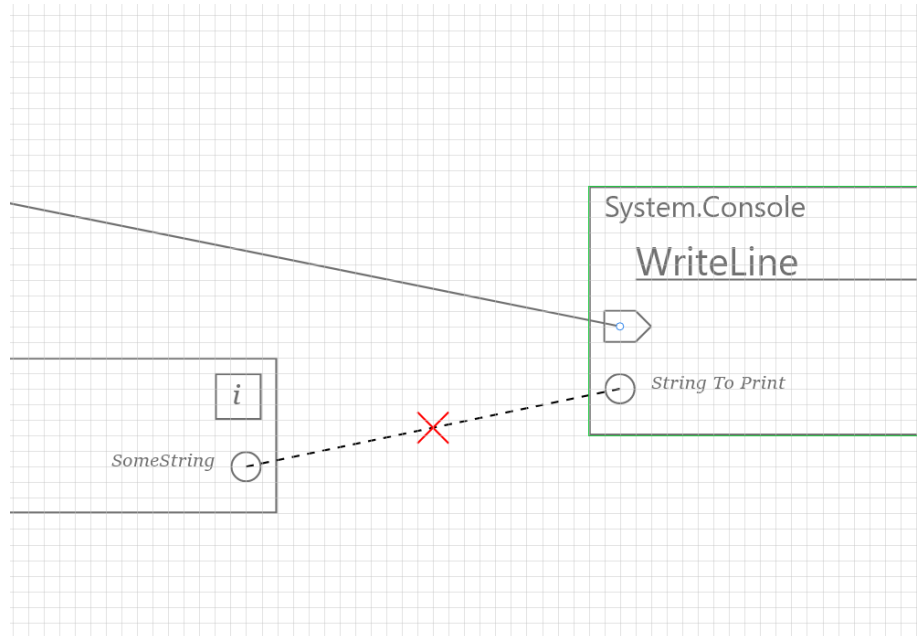


Figure 7: Illegal Connection

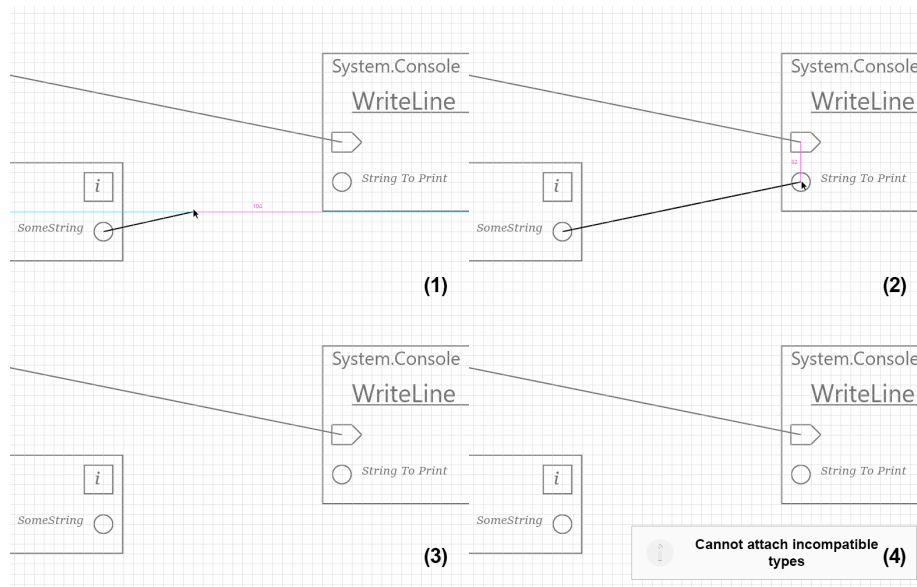


Figure 8: Illegal Connection

- **Type Discovery**

When user attempts to parse a type into a empty space in the node editor, this will prompt user with function suggestions that match the parsed type.

- **Logic Evaluation**

When user constructs loops without exist clauses, such as `while(true)` loops or implement strange if statements, such as `if(false)`, this will do a basic evaluation of the logic through existing tools in Omnisharp server.

- **Autocompletion**

When user types in a name of a function, variable, class or any other language construct, this will ask Omnisharp for Intellisense suggestions, which are then parsed into Node Editor for display.

- **Source Watcher**

When user makes changes to any of the files, Node Editor will notify VSCode of any workbench changes. These workbench changes are also parsed into language server. When the changes are made to source files or project files, language server will invoke `G#-to-C#` transpiler task.

7.2.2 G#-to-C# Transpiler

This is the actual transpiler that will convert G# IR into C# source code by the means of CodeDOM Microsoft package. CodeDOM allows construction of

source code in readable format and supports all features of C# language that are available to developers. CodeDOM is a powerful execution graph builder that supports C# and IronPython transpilation. However, it can be defined, through a transformer factory, to translate into any other language according to the language's rules.

7.2.3 Visual Node Editor

Node Editor (in code is referred to as NE) is the primary focus of this research. Visual design of the editor is explained in-depth in section Language Design. In this section, however, only the technical side of editor will be discussed. Visual Node editor is an extension for a popular Electron app "Visual Studio Code"⁶. Extension is written in typescript ES2015 and has 2 primary parts within the extension: Editor and Extension (sometimes referred to as simply frontend for Editor and backend for Extension).

Frontend of NE is an html page covered by a Konva canvas. Konvajs canvas is a high-performance canvas for drawing nodes. It has powerful API and has some very promising demos, some of which are: fully functional Google-like spreadsheets, react-konvajs which is used by some of high-end applications such as Polotno.⁷ User can manipulate nodes in the editor which will be written into a *.gs file upon saving. G# (or *.gs) files contain JSON intermediate representation (IR) of G#. This JSON contains a lot of metadata for both NE and transpiler. Furthermore, JSON is plaintext serializable data structure, meaning it can be used seamlessly with code transformation tools, version control tools, and even modified by hand. Frontend of NE is written in Typescript, unlike extension it has slightly different compilation and deployment pipelines. Frontend has to be compiled 'for browser', since, as stated above, VSCode is only a chromium that displays webpage, but has a backend that allows it to interact with operating system. Unfortunately, TypeScript doesn't support compilation for Browser runtime, which means the code will have to be bundled. This projects takes advantage of Rollup bundler, which compiles Typescript code into ES6, but the transforms it through iteration of dependency tree. Source files, along with any referenced NodeJS libraries, bundled into a single *.js file that can be instantly executed by NE frontend. The second part of the NE is backend. Backend is written in Typescript as well and can be natively run by NodeJS runtime. Due to limitation and old technology used in VSCode, backend code is still compiled into deprecated CommonJS code. Extension code is then injected into a developer instance of VSCode. For release versions, code is bundled and executed as a module when added.

⁶<https://code.visualstudio.com/>

⁷<https://polotno.dev/>

8 Discussion

This project was heavily time-constrained and therefore was left in a work-in-progress state.

9 Appendix

References

- [1] Youngkyun Baek, Ellen Min, and Seongchul Yun. Mining educational implications of minecraft. *Computers in the Schools*, 37:1–16, 01 2020.
- [2] Grady Booch. From minecraft to minds. *IEEE Software*, 30:11–13, 03 2013.
- [3] Jean Bresson and Carlos Agon. Musical representation of sound in computer-aided composition: A visual programming framework. *Journal of New Music Research*, 36:251–266, 12 2007.
- [4] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation, 11 1998.
- [5] Enrique Coronado, Fulvio Mastrogiovanni, Bipin Indurkha, and Gentiane Venture. Visual programming environments for end-user development of intelligent and social robots, a systematic review. *Journal of Computer Languages*, 58:100970, 06 2020.
- [6] Mustafa Can DEMİRKIRAN and Fatma TANSU HOCANIN. An investigation on primary school students’ dispositions towards programming with game-based learning. *Education and Information Technologies*, 02 2021.
- [7] Tomáš Effenberger, Jaroslav Čechák, and Radek Pelánek. Difficulty and complexity of introductory programming problems. In *3rd Educational Data Mining in Computer Science Education Workshop*, 2019.
- [8] Neil Fraser. Ten things we’ve learned from blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 49–50, 2015.
- [9] Swen Gaudl, Simon Davies, and Joanna J. Bryson. Behaviour oriented design for real-time-strategy games: An approach on iterative development for starcraft ai. In *Foundations of Digital Games Conference 2013 (FDG 2013)*, pages 198–205. Foundations of Digital Games, May 2013. Foundations of Digital Games 2013 ; Conference date: 14-05-2013 Through 17-05-2013.
- [10] V. V. Gribova and A. S. Kleshev. Ontology paradigm of programming. *Automatic Documentation and Mathematical Linguistics*, 47:180–187, 09 2013.
- [11] Paul A. Karger and Andrew J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *1984 IEEE Symposium on Security and Privacy*, pages 2–2, 1984.
- [12] JAMES D. KIPER, ELIZABETH HOWARD, and CHUCK AMES. Criteria for evaluation of visual programming languages. *Journal of Visual Languages & Computing*, 8:175–192, 04 1997.
- [13] Addie Matteson. A diy computer, plus minecraft. *School Library Journal*, 63(10):20, 2017.

- [14] Erik Pasternak, Rachel Fenichel, and Andrew N. Marshall. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B B)*, pages 21–24, 2017.
- [15] Dennis Ritchie. The development of the c language.
- [16] Joachim W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2:247–261, 09 1977.
- [17] David Weintrop, David C. Shepherd, Patrick Francis, and Diana Franklin. Blockly goes to work: Block-based programming for industrial robots. In *2017 IEEE Blocks and Beyond Workshop (B B)*, pages 29–36, 2017.
- [18] Kirsten N. Whitley, Laura R. Novick, and Doug Fisher. Evidence in favor of visual representation for the dataflow paradigm: An experiment testing labview’s comprehensibility. *International Journal of Human-Computer Studies*, 64:281–303, 04 2006.