Hybrid Visual Programming Language Environment for Programming Training

Rumen Daskalov [1], George Pashev [1], Silvia Gaftandzhieva [1]

[1] *University of Plovdiv "Paisii Hilendarski", "Tsar Asen" 24, Plovdiv, Bulgaria*

Abstract **– The paper presents an approach to learning and an environment for working with a hybrid visual - text programming language with a special emphasis on the training for future programmers at an early stage. An overview of other visual programming environments and languages is made and the advantages of the hybrid visual - text approach offered in this article are highlighted. Emphasis is placed on the practical implementation of a proprietary environment for the development of sample programs in the author's hybrid visual programming language. An example of the use of the environment is presented and perspectives for its development are formulated.**

Keywords **– visual programming environments, hybrid visual - text approach, programming training.**

1. Introduction

In computer science, a visual programming language (VPL) is any programming language that lets users create programs by manipulating program elements graphically rather than by specifying them textually [1]. A VPL allows programming with visual expressions, spatial arrangements of text and graphic symbols, used either as syntax elements or secondary notation.

DOI: 10.18421/TEM102-63
https://doi.org/10.18421/TEM102-63

Corresponding author: George Pashev,
University of Plovdiv "Paisii Hilendarski", Bulgaria
Email: georgepashev@uni-plovdiv.bg

Received: 18 April 2021.
Revised: 15 May 2021.
Accepted: 20 May 2021.
Published: 27 May 2021.

Many VPLs [2] are based on the idea for boxes and arrows, where boxes or other screen objects are treated as entities, connected by arrows, lines or arcs which represent relations.

VPLs are not something new. Initially, the idea behind their development was to awake the passion for programming in children from an early age. For this reason, Feurzeig, Seymour and Cynthia Solomon designed the programming language Logo in 1967 [3]. Logo is famous for its use of turtle graphics, in which commands for movement and drawing produced line or vector graphics, either on a screen or with a small robot termed a turtle. Later, block-based VPL appeared (e.g. Scratch) [4], [5]. Scratch is a block-based VPL and website targeted primarily at children 8-16 as an educational tool for coding. Although Scratch is for beginners in programming, it is not as simple as functionality language. Other known VPLs for Scratch for Android are: Ardublock, GraspIO, ReactiveBlocks and AT&T Flow Designer.

Each of these VPLs is built on a text-based programming language (such as ActionScript, JavaScript, C #, Java, etc.), which the user does not need to know to compile a program from the blocks of the VPL.

Most VPLs are considered to be mainly for children's education or use by non-specialists. No approach is specially adapted to the initial training of future programmers.

Weintrop and Wilensky [6] review earliest found study for Hybrid Visual Programming Languages as mentioned in [7], which also reviews several other attempts for Hybrid approach, such as [8] and [9]. These Hybrid approaches, however, are very bound to specific well-known text programming languages and do not provide more language independent approaches. The usage of more language independent text in the blocks is an essential part of the education, because students that already have some experience in well-known text programming language will not be able to use their previous experience to outcompete other students with no previous experience in assignment in which this could be important. Moreover, usage of more universal Hybrid Visual-Textual notation would make development of cross-translation to multiple text languages easier.

A significant adaptation addressing some downsides reviewed could be made using a "semi-graphical" approach, in which students still have to describe a language independent "pseudocode" in some of the visual components of the VPL, which makes it a Hybrid VPL (HVPL). The last would allow a smoother transition from a visual environment to a subsequent classic text environment. On the other hand, cross-compiling code from VPL to more than one text programming language allows students to perform comparative analysis on the linguistic constructions of different well-known text programming languages. Using the hybrid visual-text approach teaches students algorithmic thinking while describing an algorithm as code.

This paper presents an author's environment for performing a VPL with a hybrid visual-text approach (HVTA). Based on a visual block chart editor, a cross-translator is created from the code generated by the visual editor to a language for immediate execution or compilation, (linking) and execution. The developed environment is suitable for teaching first year students in a higher education institution in Introduction to Programming Languages and can be used as an integrated environment for programming training through VPL.

2. Architecture of the Environment for Performing a VPL

The architecture (see Figure 1) consists of the following main components:

- **User interface** – it will contain a container with the graphical elements (shapes) needed for drawing a chart, a drawing area and a control panel;
- **Event module** – it will monitor for each event (change) in the drawing area, performing the appropriate checks and the necessary validations for the event;
- **Cross-translator (Interpreter)** – it will convert the chart into code in the programming language selected by the user;
- **Compiler** – it will compile and execute the program (if the cross-translator is set to C #).

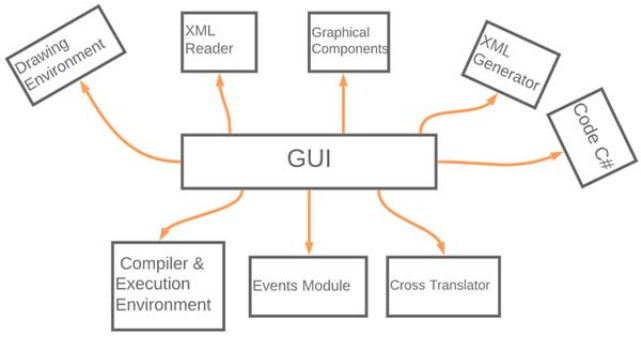

Figure 1. Architecture

The ***user interface*** includes a ***drawing environment, graphical elements and an XML generator***.

The ***drawing environment*** allows us to add an element (drag and drop), add element content, edit element content, remove an element, add a connection (connector) between two elements, change a connection between two elements and remove a connection (connector).

Figure 2 presents the ***graphical elements***. ***Start*** sets the start of the algorithm. ***Data*** accepts only primitive data types: *integer types* (sbyte, byte, short, ushort, int, uint, long, and ulong), *floating-point types* (float, double), *boolean type* (bool), *newline* (

), character (char) and string (string). When declaring a variable, the variable type and name are set. In this element, the "?" after "=" means that the value of the variable will be inputted from the console. Each variable is declared on a new line. Valid values that Data accepts are variable type, variable name, assignment character, and variable value or console input character. ***Decision*** works with the conditional statements if, if-else and while (the first two are default and while must be explicitly specified). All variables that will be used for the check have to be pre-declared in the Data element. ***Process*** performs all operations between the declared variables in Data and prints the results in the console. ***Sub-process*** calls a new method (opens another chart). A connector is used to connect two elements, showing the execution sequence. The values are entered automatically by the program, and the manually entered values are deleted automatically. The valid values of Connector are "true" and "false" (if the initial element is Decision). For all other elements, the Connector does not accept a value.

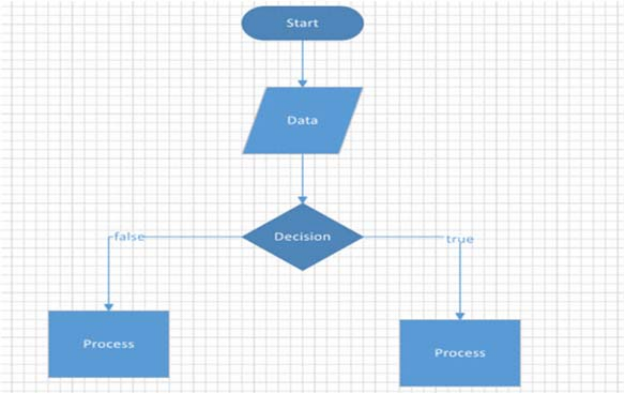

Figure 2. Graphical elements

When saving the block chart, an ***XML file*** is generated. This file contains all the information needed by the program to visualize the block chart from the file. Graphic elements are divided into two groups: connector and shape. Each graphical element in the diagram corresponds to a tag with attributes that represent its main characteristics.

The ***Events module*** track all events (changes) in the user interface drawing environment and consists of the following classes:

- *ConnectorContent* - monitors whether a connection needs content and fills it when needed;
- *ItemColor* - changes the color of the shape or connector when the connection is incorrect or missing;
- *ItemContent* - monitors the content of each shape in the chart to see if it meets the predefined criteria and if not change the text color;
- *Registrator* - shows only the Start element until it is placed on the drawing area.

The ***Cross-translator (Interpreter)*** contains language modules in which the content of shapes is parsed in a pre-selected programming language (CSharp, Java, Python) and 3 main classes:

- *Language* - sends the chart to the module which corresponds to the selected programming language;
- *Sequence* - arranges the shapes according to their connection sequence;
- *Content Retriever* - retrieves content from shapes.

Only when C # is selected as programming language, the ***Compiler*** can be called with the "Execute code" button and execute the source code.

3. Development of the Environment for Performing a VPL

An Integrated Development Environment (IDE) is a software application that consolidates the main tools which computer professionals use in the software development process. The IDE contains 3 main components: Source Code Editor, Compiler or Interpreter, and Debugger. The developer has access to all these components through a graphical user interface (GUI). Some IDEs target only a specific programming language and contain tools that match its fundamental programming style. Other IDEs (such as Microsoft Visual Studio, Eclipse, NetBeans, IntelliJ IDEA) are multilingual and allow additional modules to be installed at the user's request.

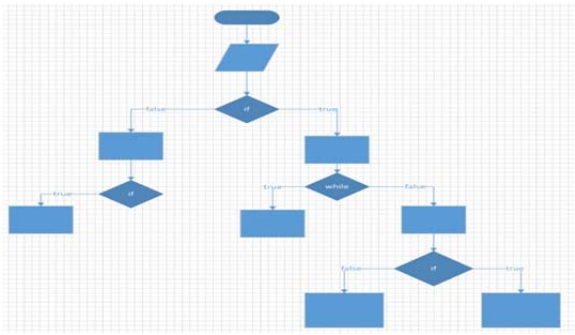

Figure 3. Nested conditional statement

For the development of the visual part of the program and a graphical user interface for desktop applications, the .NET Framework library (Windows Forms) and DevExpress components are used. The visual part is a component-oriented development in which each component solves a specific task. The chart itself is the body of a method, its components are separate parts of this method, and the sequence of connecting these components is the order in which they will be traversed. This way of connection allows using an unlimited number of nested conditional statements *if*, if-*else* and *while*, as a result of which the need for *else-if* is eliminated (see Figure 3).

The ***Events module*** does all real-time checks, monitors any changes in the drawing area. This module prevents the generation of program code if there are unconnected elements. The attempt to generate a code if there is an element in red activates a pop-up message on the screen informing that the generation cannot take place (see Figure 4). The last makes working with the program quite intuitive and thus reduces the probability of a compilation error.

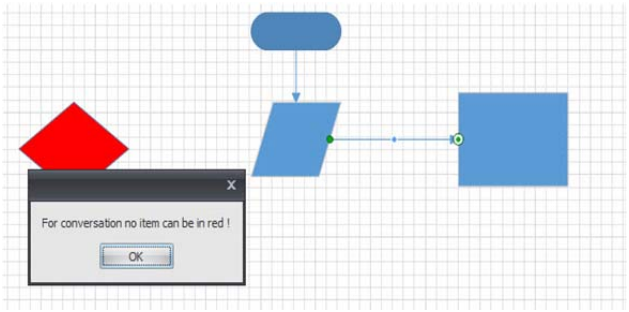

Figure 4. Nested conditional statement

When creating a new chart, its elements are numbered in this sequence in which they are added to the drawing area. Their arrangement and connection can be changed an unlimited number of times. For conditional structures, there is also no restriction or requirement on how and which part to be created first, which also applies to nested conditional structures. All this leads to a mismatch between the sequence in which they are added and the connection sequence according to which the code is generated. Since this is the most important unit on which the results of the execution of the converted code depend the *Sequence* class executes this task. This collection is submitted to a distribution function. After cloning the collection, the procedure for comparing connectors begins. When opening the program while the drawing area is empty, the only figure that can be placed is Start. This allows the program to "find" the beginning and "mark" it as a reference from which to start checking each shape and its number of outbound connectors. When the finite element search procedure finds one, it and its inbound connector are removed from the temporary collection. For conditional statement, the number of outbound

connections depends on the need to take action if the check is false.

The ***Interpreter*** is divided into 3 independent parts - *Data*, *Decision* and *Process*. Each part consists of two or more classes and has its own vocabulary in the form of constants. This helps to avoid the use of a database.

Data interprets the input data. Currently *Data* processes only primitive data types and their input from the console (Syntax: *<def> <variable-name> <=> <variable value> <as variable-type>*), as well as printing text in the console (Syntax: *<print > <text>*). All declared variables and their types are stored in a dictionary used in *Decision* and *Process*. Figure 5 shows examples of declaring variables and printing text.

```
print Въведете стойност за a: // ще се преобразува на: Console.WriteLine("Въведете стойност за a: ");
def a = ? as int              // ще се преобразува на: int a = int.Parse(Console.ReadLine());
def b = test as string        // ще се преобразува на: string = "test"
```

Figure 5. Examples

In case the type of the conditional statement is not declared, the default type is *if*. Figure 6 shows examples of conditional statements transformation.

```
c bigger (a+b) and a smoller b // ще се преобразува на: if(c>(a+b) && a<b)
if a same b or c different 9   // ще се преобразува на: if(a==b | c != 9)
if a same b and c not equal 9  // ще се преобразува на: if(a==b && c != 9)
while b smoller a              // ще се преобразува на: while(b<a)
                  ...........
```

Figure 6. Conditional statement transformation

Decision interprets the contents of the conditional *if* and *while* constructs. This is because the variables are pre-declared in *Data*, comparison operators as well as logical operators are interpreted here. If the conditional statement type is not declared, the default type is *if*.

Process interprets the commands indicating what to do after the data has been entered and processed, as well as in case it is in a conditional statement. *Process* also has its own class of constants (see Figure 7).

```
.....................
//Print line
    public const string PRINT = "print";
    public const string WRITE_LINE = "Console.WriteLine(";
    public const string WRITE_LINE_END = ")";


    //Replace
    public const string REPLACE = "replace()";
    public const string REPLACE_START = ".Replace(";
    public const string REPLACE_END = ")";
    public const string REPLACE_SEPARATOR = ", ";
    public const string WITH = "with";

    //Lenght
    public const string LENGHT = "lenght()";
    public const string LENGHT_START = ".Lenght";

    public const string START = "[";
    public const string END = "]";
    public const string END_LINE = ";";
    public const char QUOTE = '"';
    public const string EMPTY = "";
    public const string PLUS = "+";
    public const string SPACE = " ";
    public const string EQUAL_ARG = "=";
    public const string NEW_LINE = "\r\n";
.....................
```

Figure 7. ConstantsProcess.cs

Here all simple operations are performed, such as addition, subtraction, multiplication, division, value assignment and concatenation. The syntax in this case is: *<variable><=><variable1> <operation><variable2>*. Due to the fact that these are simple operations, there are no special requirements in addition to the variable to be declared in *Data* and not to be merged. Regarding the printing of a result in the console, the variable declared in *Data* must be in square brackets (Syntax: *<print> <[variable]>*). Examples of these operations are presented in Figure 8.

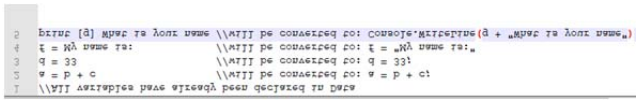

Figure 8. Data processing

Unlike the other 2 modules of the interpreter, *Process* also has a class *ProcessFunctions*, which contains various complex functions, e.g. *Replace* (Syntax: *<variable><=><[vatiable1]><replace> <[word/variable2]><with><word/variable3>*). The specified function's checks whether the variables in square brackets are declared, as *variable 2* or *variable 3* can be an undeclared string.

The ***Compiler*** contains only one class (*Compilator*), which does not use external libraries to compile the code. The code generated by the interpreter is the body of a method. For this reason, the compiler has its procedures - for creating namespaces, adding traditional console libraries, adding the necessary opening and closing brackets typical for C#, creating an empty main method in which a for-loop add the content created by the user and converted by the interpreter. During the compiling problem, a window with errors pops up, as a result of which the compilation became impossible. In case of success, a pops up window confirms the success of the compilation process.

4. Experiment

The environment is tested several types of statements in the Theory of Algorithms – simple conditional statement, nested conditional statement, and string. This section describes an example of a simple conditional statement *if* (see Figure 9).

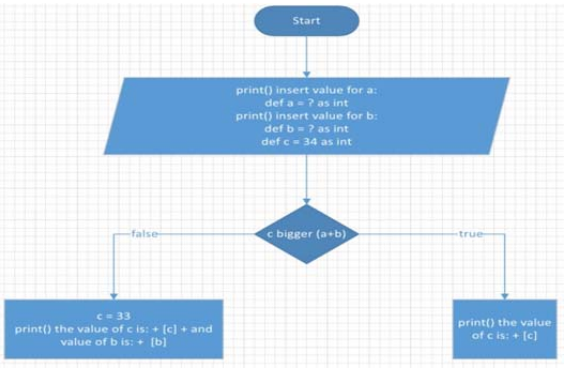

Figure 9. Simple Conditional Statement – Filling

When the user inputs values of variables, condition and the appropriate actions for fulfilled and unfulfilled condition, and click the Convert button, the code generation is activated. A window pops up with the generated code (see Figure 10). We can recognize the C# syntax in the window. If there are no syntax errors in the code, after clicking the *Execute code button*, the console window appears.

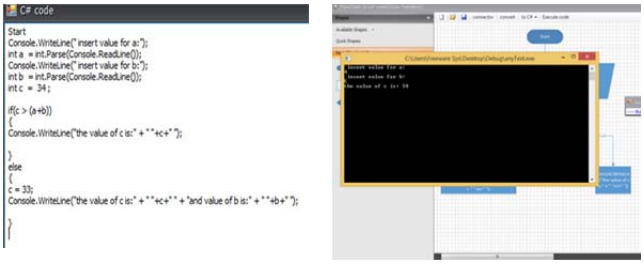

Figure 10. Simple Conditional Statement – Code generation & Console Windows

5. Training Tasks

The developed environment is suitable for teaching first year students in a higher education institution in Introduction to Programming Languages and can be used as an integrated environment for programming training through VPL.

Three groups of tasks can be formulated:

- **Group 1**. Training in algorithms synthesis and analysis by describing them in the hybrid visual-text language (VPL with pseudocode elements in some visual components) - This hybrid approach allows complex algorithms to be described more compactly and provides some opportunity for a visual representation. Once the student describes the algorithm through HVTL, he/she can immediately start the execution of the algorithm, see the result and verify it;
- **Group 2**. Transition from Diagram to Text – This allows students to gain greater linguistic confidence in learning a specific text programming language. In the presence of an already synthesized and tested algorithm in HVPL, it can be cross-translated into text programming language and studied by the student in greater detail how the individual constructions (including the pseudocode) correspond to the corresponding constructions in the text programming language;
- **Group 3**. Comparison of text programming language with each other - In the presence of an already synthesized and tested the algorithm in HVPL; it can be cross-translated to more than one text programming language and their language constructions can be compared with such specific examples. The approach described in this way provides training with examples.

For **Group 1**, an example assignment for student would be to develop a program in the HVPL in the environment and run it and analyze the results. Another sample assignment would be to have a given HVPL diagram and to try to guess what would be the output of this program.

For **Group 2**, an example assignment would be to try to translate the code in HVPL to a specific text programming language and then to invoke the Cross-Translator and compare both translations.

For **Group 3**, an example assignment would be to try to translate a given already generated code from HVPL from one text programming language to another. Then the student would be asked to invoke the Cross-Translator from the original HVPL to the target text programming language and to compare and analyze both translations.

6. Conclusion

Although the developed environment is a beta version, its behaviour during the tests proves that it is suitable for beginner training. Currently, there are three main shapes in which the user can input data, set conditions and actions if the conditions are met or not met. The beta version allows the generation of C# code.

The future work will focus on the development of the software application in several directions, e.g. implementing:

- capabilities for generation of Python and Java code;
- new elements in the VPL;
- support for other VPLs;
- concept of language paradigm and grouping of VPL and languages to which they are translated into paradigms;
- storing the data for VPL, the programming languages to which they are cross-translated, the way of execution (interpretation, compilation, etc.) of the source languages, and the paradigms in which they are grouped in Knowledge Base.

These improvements of the developed environment will make it a universal tool that can be used for training in other study disciplines, e.g. for Training in Computational Linguistics in order to create new VPL, programming paradigms, and etc.

Acknowledgements

The paper is supported within the National Scientific Program "Young scientists and Post-doctoral students" in accordance with Appendix No. 11 of Council of Ministers Decision No. 577 of 17 August 2018.

References

[1]. Plaza, P., Peixoto, A., Sancristobal, E., Castro, M., Blazquez, M., Menacho, A., ... & Lopez-Rey, A. (2020, April). Visual block programming languages and their use in educational robotics. In *2020 IEEE Global Engineering Education Conference (EDUCON)* (pp. 457-464). IEEE.

[2]. Kucukural, A., Garber, M., Yukselen, O., Turkyilmaz, O., Ozturk, A., Girard, I., & Martin, R. (2020). DolphinNext: A graphical user interface for creating, deploying and executing Nextflow pipelines. *Journal of Biomolecular Techniques: JBT*, *31*(Suppl), S25.

[3]. Rugescu, A. M. M. (2020). Logo: Creativity, Innovation and Visual Intelligence. *Journal of Industrial Design and Engineering Graphics*, *15*(1), 13-18.

[4]. Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, *10*(4), 1-15.

[5]. Lamb, A., & Johnson, L. (2011). Scratch: Computer Programming for 21st Century Learners. *Teacher Librarian*, *38*(4), 64.

[6]. Weintrop, D., & Wilensky, U. (2015, June). To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th international conference on interaction design and children* (pp. 199-208).

[7]. Noone, M., & Mooney, A. (2018). Visual and textual programming languages: a systematic review of the literature. *Journal of Computers in Education*, *5*(2), 149-174.

[8]. Federici, S. (2011, October). A minimal, extensible, drag-and-drop implementation of the C programming language. In *Proceedings of the 2011 conference on Information technology education* (pp. 191-196).

[9]. Kyfonidis, C., Moumoutzis, N., & Christodoulakis, S. (2015). Block-c: A block-based visual environment for supporting the teaching of c programming language to novices. In *9th International Conference" New Horizons in Industry, Business and Education"(NHIBE 2015): 27-29 August 2015 Skiathos Island, Greece* (pp. 160-166).