Pak. J. Engg. Appl. Sci. Vol. 23 July, 2018 (p. 55–65)

Towards a Universal Framework for Visual Programming Languages

Muhammad Idrees, Faisal Aslam*, Khurram Shahzad, Syed Mansoor Sarwar

Punjab University College of Information Technology (PUCIT), University of the Punjab, Lahore, Pakistan

* **Corresponding Author:** Email: faisal.aslam@pucit.edu.pk

Abstract

A Visual Programming Language (VPL) can help programmers quickly develop robust programs using simple drag-and-drop of visual elements, without worrying about the syntactic details of a programming language. In contrast to the textual programming languages, a VPL is usually designed for a specific domain such as to teach programming to beginners or to develop engineering models. Therefore, it is highly likely that numerous VPLs will be developed in future for different tasks and domains. Presently, each new VPL being developed is either created from scratch, or in some cases a newly developed VPL has used codebase of only a single existing VPL. Consequently, significant effort is required for developing a new VPL. This paper highlights the need of a universal framework to drastically reduce the time and effort required to develop a new VPL, and to enhance reusability of an existing VPL codebase. The framework offers a layered approach to VPL development. The layered approach offers an opportunity to generate a VPL layer by combining components from the corresponding layers of existing VPLs while writing minimal new components of the layer when required.

Keywords: Visual programming languages, VPL, VisFra, VPL evolution

1. Introduction

Typically, computers are programmed using programming languages that require typing code manually; such programming languages are called Textual Programming Languages (TPLs). Writing code in a textual language entails understanding the programming concepts and strictly following the syntax of that programming language [1]. That is, each line should be written carefully and even a minute syntactic mistake can result in generating multiple compilation errors. Due to the inherent difficulty in writing code using TPLs [2, 3], programing has been limited to few experts who have mastered the art of textual programming over several years. In order to make programming accessible to masses, it is desirable to significantly reduce the effort of typing a program so that a programmer can focus on the logic of a program to solve the problem at hand rather than wasting resources on the intricacies of the programming language syntax [4]. To this end, many Visual Programming Languages (VPLs) have been introduced.

A VPL allows a programmer to develop the logic for a program by simply dragging-and-dropping a visual element on a canvas and subsequently connecting that visual element with other elements [5, 6]. A visual element hides the syntactic complexities of the programming language from the programmer while the connections between elements represent the logic of the program.

A new VPL is usually developed keeping in mind the needs of a specific domain. For instance, since 2011 more than 15 domain-specific VPLs have been developed [7-32]. This trend may continue in future, leading to the development of numerous VPLs, each designed to fulfill the need for performing a specific task in a domain. Presently, each new VPL being developed is either created from scratch, without using any of the existing VPLs' codebase, or, in some cases, a VPL has used codebase of only a single existing VPL. Consequently, significant effort is required for developing a new VPL. To address this issue, this paper makes the following main contributions:

- Using a rigorous process, we choose 40 VPLs, developed from 1983 to-date and analyze the relationships between these VPLs. We have identified three types of relationships between these 40 VPLs, which include based-on, similar-to, and refers-to relationships.
- Given the insights developed from the analysis of relationships between VPLs, we have divided the existing VPLs into two generations. Form the analysis of relationships, we deduce the design limitations of the existing VPLs of the two generations.
- Finally, building on the lessons learned from the capabilities of existing VPLs, we envision

the next generation of VPLs. The third generation (i.e., future) VPLs should be based on a universal framework, which supports a layered-based approach to the development of a VPL. The use of our proposed framework will significantly reduce the time to develop a third generation VPL and will make it easier to reuse.

The rest of the paper is organized as follows. In Section 2, we outline the three types of relationships that a VPL may have with other VPLs. Section 3 divides VPLs in two generations and discusses the shortcomings of each generation. Section 4 presents the third generation VPLs and lists the salient features of the layered universal framework. Finally, we conclude the paper and present future work in Section 5.

2. Relationships between VPLs

One of the contributions of this paper is to identify the evolution of VPLs over the years, in order to understand the relationships between them. The understanding of relationships will help us to ascertain whether a VPL has influenced other VPLs and when a VPL codebase has benefited the development of other VPLs. We propose that three types of relationships are possible between a pair of VPLs. These relationships are based-on, similar-to, and refers-to. In general, a relationship between two visual languages L_1 and L_2 is represented by $L_1 \rightarrow L_2$. This implies that L_2 has a relationship with L_1 and L_1 precedes L_2 in terms of date of origin. The three types of relations between VPLs are formally described below:

- **Based-on**: A relationship between two VPLs is called based-on when L_2 is either developed using the codebase of L_1, or in the publication of L_2 it is explicitly declared that L_2 is based on L_1. A solid line between L_1 and L_2 represents this relationship. For example, BlockPy is based-on Blockly because BlockPy uses the codebase of Blockly and BlockPy also explicitly acknowledges that "*BlockPy owes much of its power to Blockly*" [7]. Thus, we can say that BlockPy—Blockly. Of the three types of relationships between VPLs, based-on is the strongest form of relationship between VPLs, among the three types of relationships.
- **Similar-to**: A relationship between two VPLs is called *similar-to*, if the both VPLs look the same and provide a similar programming interface. A thin line between L_1 and L_2 represents this relationship. For example, Pencil Code is similar-to Scratch due to several reasons including both VPLs are block based, and have various similar visual elements to represent loops, conditions, and drawing procedures. Thus, we can say that Pencil Code—Scratch. We have identified this relationship with the consent of at least two independent programmers who have developed multiple programs in both VPLs. Similar-to relationship is a weaker type of relationship than based-on.
- **Refers-to**: The weakest type of relationship between two VPLs is called *refers-to*. This relationship represents the fact that the publication of L_2 has referred to L_1, thus indicating that the creators of L_2 were aware of the existence of L_1. This relationship between L_1 and L_2 is represented by a dotted line. For example, Programinate has referred-to relationship with FLINT as the publication of Programinate has made a reference to FLINT. Thus, we can say that Programinate⋯FLINT.

3. Evolution of VPLs in Two Generations

In this section, we first provide an overview of the systematic and rigorous protocol to identify a comprehensive set of VPLs. Subsequently, we outline the procedure adopted to reliably mark all possible relationships between the VPLs. Finally, we classify these VPLs into two generation based on the relationships and deduce the design limitations of each generation of VPLs.

3.1 Protocol of Identifying VPLs

We employed a systematic and rigorous protocol to identify a comprehensive set of VPLs [8]. The protocol includes formally defined inclusion criteria, a systematic procedure to apply the criteria, and a cross-validation of the characteristics of the VPLs under consideration. According to the criteria, a VPL is shortlisted if:

a) it allows implementing basic programming constructs such as if-condition and loops,

b) these constructs can be implemented using drag-and-drop, instead of typing textual code, and

c) enough information about the VPL is available to evaluate the strengths of its various characteristics.

The procedure entails generating of a list of candidate VPLs and screening them to identify the VPLs that meet the inclusion criteria. As a result

of the application of this procedure, 40 VPLs were identified [9-81]. Table 1 lists these 40 VPLs with their dates of origin and domains.

Finally, to *cross-validate* the characteristics of each VPL five researchers independently reviewed the documentations of these VPLs and, whenever possible, developed visual programs in these VPLs to extract their various characteristics. The protocol to identify VPLs and gather their characteristics via cross-validation is depicted in Fig. 1.

3.2 Identifying Relationships between VPLs

After identifying 40 VPLs using the protocol summarized above, we employed a systematic procedure to mark the three types of relationship among these VPLs.

Table 1: VPLs and their purpose

Sr. No.	Year	VPL	Domain/Purpose
First Generation of VPLs			
1	1983	Marten (Prograph)	General purpose
2	1986	DRAKON	Teach Programming basics
3	1986	LabVIEW	Data acquisition and visualization
4	1991	Agent Sheets	Kid's education
5	1992	BACCII/BACCII++	Teach Programming basics
6	1992	Analytica	Analytical modeling
7	1996	EToys	Kids programming
8	1998	Alice	Turtle graphics, Games
9	1999	FLINT	Teach Programming basics
10	2001	The SFC Editor	Teach Programming basics
11	2001	SIVIL	Teach Programming basics
12	2004	Raptor	Teach Programming basics
13	2004	Larp	Teach Programming basics
14	2004	Visual Logic	Teach Programming basics
Second Generation VPLs			
15	2005	Iconic Programmer	Teach Programming basics
16	2005	Scratch	Kids games, Animations
17	2006	B#	Teach Programming basics
18	2006	Microsoft VPL	Robotics
19	2006	Lego Mindstorms Software	Robotics
20	2008	StarLogo TNG	Turtle graphics
21	2009	Progranimate	Teach Programming basics
22	2009	devFlowcharter	Teach Programming basics
23	2009	GameSalad	Games, Animations
24	2009	Kudo	Turtle graphics, Games
25	2010	App Inventor for Android	Teach Programming basics
26	2011	Flowcharts Interpreter	Teach Programming basics
27	2011	Snap	Kids Games
28	2011	Stencyl	Kid's education, Games
29	2011	Touch Develop	General purpose, Mobile
30	2012	Blockly	General purpose
31	2013	CODE	Kids programming
32	2013	Open Roberta	Robotics
33	2013	MBlock	Robotics, IoT
34	2013	Pencil Code	Turtle graphics, Education
35	2013	Dynamo	3D modeling, Art
36	2014	Flowgorithm	Teach Programming basics
37	2014	Tynker	Kid's education, Games
38	2015	VIPLE	Robotics, IoT
39	2015	Beetle Blocks	Turtle graphics 3D
40	2017	BlockPy	General purpose, Scientific

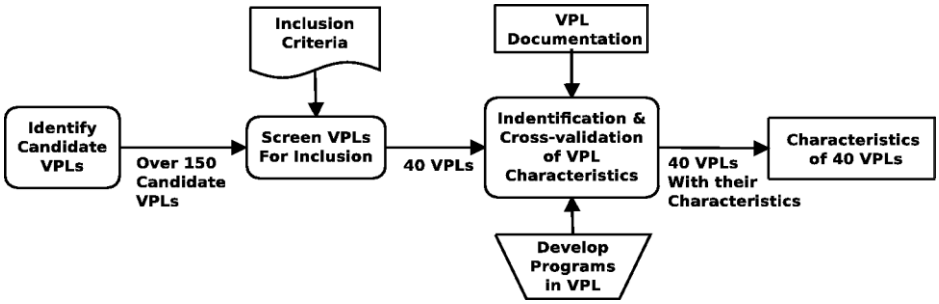


Fig. 1: Summary of the Protocol used to identify VPLs and gather their characteristics

The understanding of these relationships helped us to ascertain whether a VPL has influenced other VPLs and when a VPL codebase has benefited the development of other VPLs. The relationship marking procedure consists of two steps. In the first step, we formed a team of two researchers with expertise in using VPLs. Each researcher worked independently to perform two tasks. That is, locate the documentation and publication of each VPL and, whenever possible, develop example programs in a VPL. Based on the documentation and the development experience, each researcher marked a relationship between VPLs. In the second step, each researcher reviewed the relationship markings as well as the underlying rationale of the other researcher. In conclusion, the relationships included in this study were developed on a mutual consensus, based on the interpretation of the available literature and the experience of using VPLs.

3.3 Evolution of VPLs

As a result of the marking procedure discussed above, we identified 50 relationships between the VPLs. These include 10 based-on, 3 similar-to, and 37 refers-to relationships. The list of all the relationships is presented in Table 2.

The table lists all the individual relationships between VPLs, however the evolution of VPLs could not be understood without depicting these relationships in the context of time. To this end, in Fig. 2 we have sorted all the VPLs with respect to the date of their origin and drawn the relationship between them. In the Fig. 2, a small solid circle represents a VPL and a line between VPLs depicts relationships between them. Note that a solid circle is placed corresponding to the year of origin of a VPL and VPLs are sorted with respect to the time of their origin.

The depiction of relationship provides valuable insights about the evolution of VPLs. We observe that there is only a single based-on relationship till 2004. In contrast, from 2005 onwards multiple VPLs have developed several based-on relationships between them. Using this insight, we define two generations of VPLs. The first generation includes all the VPLs developed till 2004 whereas the second generation contains VPLs developed from 2005 onwards. These two generations of VPLs are represented by different color schemes in Fig. 2. In the following subsections, we reflect on the two generations of VPLs.

Table 2: Relationships of VPLs

Relationship Type	Relationships between VPLs
Based-on	**Marten (Prograph)** → Lego Mindstorms Software; **Scratch** → Snap, Stencyl, MBlock; **Microsoft VPL** →VIPLE; **Snap** → Blockly, Beetle Blocks; **Blockly** → CODE, Open Roberta, BlockPy;
Similar-to	**BACCII/BACCII++** → Iconic Programmer; **FLINT** → Raptor; **Scratch** → Pencil Code;
Refers-to	**LabVIEW** → Dynamo; **BACCII/BACCII++** → FLINT, Raptor, Scratch, Programinate; **EToys** → Scratch; **Alice** → Scratch, StarLogo TNG, GameSalad, App Inventor for Android, Touch Develop, Kudo, CODE; **FLINT** → The SFC Editor, Raptor, Programinate; **The SFC Editor** → Programinate; **Raptor** → Programinate; **Visual Logic** → B#, Programinate; **Iconic Programmer** → Programinate; **Scratch** → Kudo, GameSalad, Touch Develop, Blockly, CODE, Open Roberta; **B#** → App Inventor for Android; **Microsoft VPL** → Lego Mindstorms Software; **Lego Mindstorms Software** → Kudo, App Inventor for Android; **devFlowcharter** → GameSalad; **Kudo** → CODE; **App Inventor for Android** → Touch Develop, Pencil Code; **Blockly** → Pencil Code; **CODE** → Pencil Code;

3.3.1 First Generation of VPLs

Fig. 2 shows that the first generation VPLs were developed independently of each other. This can be inferred from the fact that VPLs of the first generation have mostly *refers-to* relationship between them and only a couple of *similar-to* relationships. This indicates that the development of a new VPL of the first generation did not benefit from the codebases of existing VPLs. We argue that there are two possible reasons for this scarcity of strong relationships between VPLs. These are, a) lack of modularity in the codebase of VPLs, and b) the absence of proper documentation about the underlying design of the VPLs. For instance, no VPL of this generation has provided easy to reuse components, which a VPL being developed could import in order to reduce the time and effort of its development. One exception is LabVIEW, whose codebase was used in the development of Lego Mindstorms Software. However, it should be noted that the same organization (i.e., National Instruments) was involved in the development of Lego Mindstroms Software and LabVIEW [77]. Thus, we argue that this based-on relationship does not truly reflect that Lego Mindstroms has benefited from LabVIEW due to modularity of its software.

3.3.2 Second Generation of VPLs

Fig. 2 also shows that many second generation VPLs have *based-on*, the strongest form of relationship, with other VPLs of this generation. We argue that the emergence of abundant relationships of the strongest form was caused by two factors: 1) the improved modularity in the design and code of these VPLs, and 2) the user-friendly documentation of the VPLs in this generation. Scratch and Blockly are the two prominent modular-VPLs in this generation. These VPLs provide useful pointers for the developers' community to extend and/or reuse their functionality. For instance, Blockly has a set of visual blocks supporting a wide range of functionalities. Thus, a developer interested in creating his own VPL could import some of the blocks provided by Blockly to his VPL by writing a few lines of code. Furthermore, Blockly also provides a developers' tool to create new blocks or modify the functionality of its existing blocks. This enabled creation of many languages that have based-on relationship with Blockly. Although, the VPLs of the second generation advanced significantly from the first generation, however, the VPLs of second generation still benefited from existing VPL codebase in a limited way. In the next section, we explain the limitations of the second generation VPLs and envision future VPLs.

4. The Third Generation of Future VPLs

The VPLs of the second generation advanced significantly from the first generation. However, the VPLs of second generation had many limitations. In this section, we present these limitations and suggest how these limitations could be overcome in the third generation of VPLs.

4.1 Need to learn codebase of VPLs

Unlike the first generation of VPLs, the VPLs of the second generation, such as Scratch and Blockly, had structured their codebase into modules. Due to this underlying structure, a developer of a new VPL does not necessarily require complete understanding of an existing VPL codebase to reuse it. Instead, the developer can change code of a few selected modules and write new modules to develop a new VPL. This implies that the developer has to partially understand codebase of an existing VPL depending upon the modules he/she wants to reuse in developing the new VPL. For example, Blockly uses JavaScript and HTML. Thus, one has to first learn JavaScript and HTML, and then partially understand Blockly codebase before reusing it. We argue that understanding code of an existing VPL, even partially, still hinders rapid development of new VPLs.

Ideally, a new VPL developer should be able to reuse an existing VPL codebase and write his VPL's new components, without writing a single line of code in a textual language. This can be accomplished in two steps. First, identify all the tasks that need to be performed in order to create a new VPL. Second, provide a graphical user interface to accomplish those tasks via drag-and-drop. Although, one still has to manually type some of the properties and names but a VPL development can be done without writing code in any textual programming language. In this paper, we propose creation of a universal framework, VisFra, which defines all the tasks that need to be accomplished in order to create a VPL. Once this has been achieved, one can write a GUI to perform the required tasks using drag-and-drop and create a new VPL.

4.2 Incompatibility of VPLs

Incompatibility between VPLs is another key reason that hinders the reuse of VPLs in developing a new VPL. For instance, two modular VPLs, which were either developed using two different TPLs or provide different kind of modules, cannot be reused simultaneously in the creation of a new VPL, due to their incompatibility. This may be the reason why there does not exist any one-to-many correspondences between the second generation VPLs, that is, the reusability is restricted to one-to-one correspondences (see Fig. 2).

We argue that if multiple compatible VPLs are provided then a new VPL can reuse many existing VPLs simultaneously, creating one-to-many correspondences between VPLs and starting a new era of reusability. Thus, the VPLs developed using VisFra, will be mutually compatible.

4.3 Dependency of VPLs

There are several dependencies associated with VPLs including, operating system on which they run, the user interfaces provided by them, and the textual language(s) in which the textual code of the VPLs are generated. This means that changing one of the dependencies requires significantly rewriting the codebase for the VPL. For instance, in order to reuse a VPL in Linux that was originally developed for Windows requires significant rewriting of code. Similarly, using VPLs that translate visual programs into Java code requires significant rewriting of codebase if it is be reused to translate code in C++. The presence of these dependencies creates new challenges to the reusability of VPLs.

To that end, another objective of VisFra is to make VPLs independent of operating system, GUI, and the textual language used to develop it. More specifically, VisFra will employ a modular and layered approach to divide each task of VPL development into parts and layers. Hence, for instance a VPL developer may be able to generate code in multiple TPLs just by changing the specific layer of a VPL provided by VisFra for the TPL support, leaving rest of the VPL code unchanged.

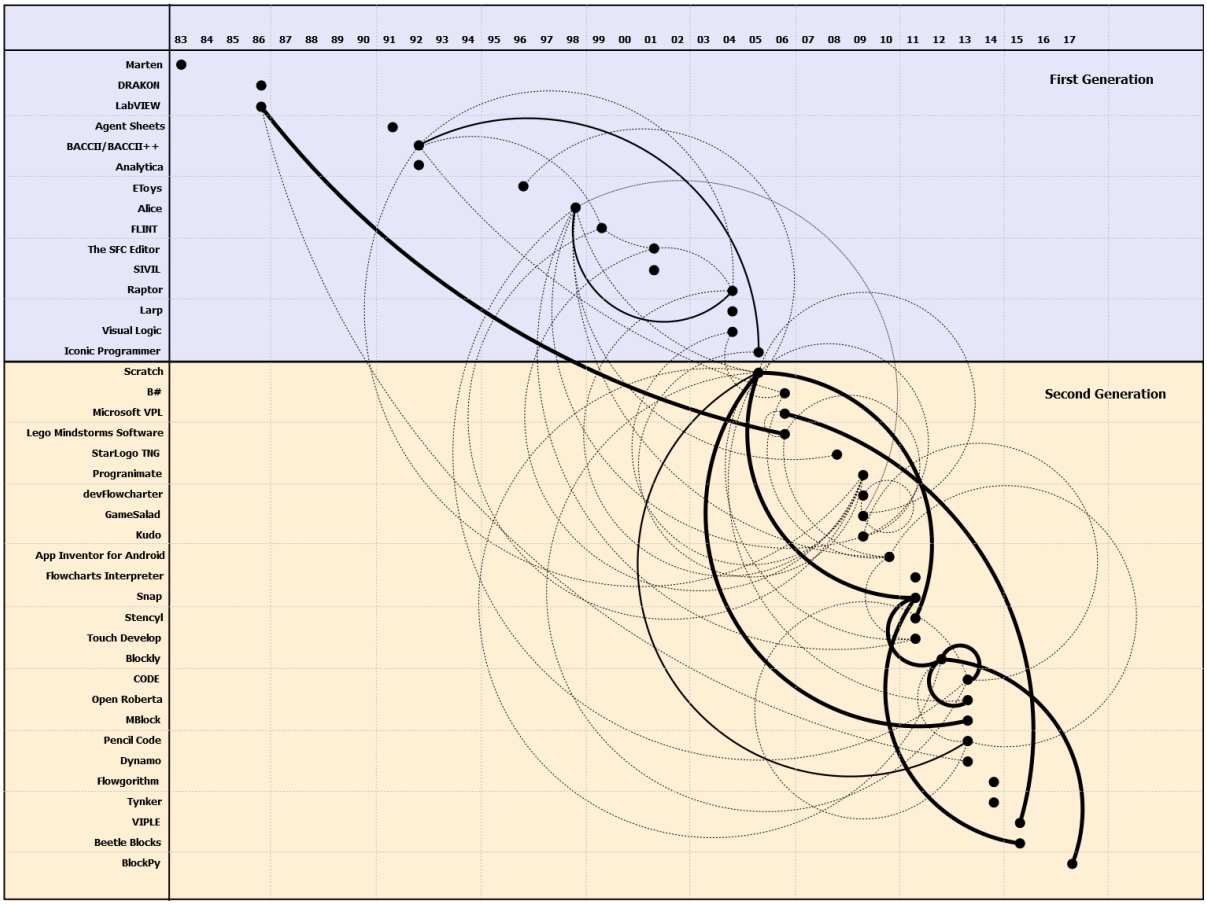


Fig. 2: The three types of relationship between VPLs

Fig. 3 shows a typical VPL developed using VisFra. This third generation of VPL has n layers. Each layer has multiple blocks at a maturity level represented by that layer. The next (higher) layer builds on the blocks of the previous (lower) layer.

5. Conclusion

Fig. 4 shows the evolution of VPLs in three generations. In the first generation, a VPL was composed of a single layer and a single block. To reuse code of such a VPL one had to adopt more-or-less the whole codebase and change it to develop a new VPL. For example, Lego Mindstorms Software was developed by adopting codebase of LabVIEW software, only. This kind of based-on relationship can only exist if some of the team members involved in the development of the previous VPL are developing the new VPL while using their knowledge of the codebase. Therefore, in the first generation almost all the VPLs (except one) have either refers-to or similar-to relationships.

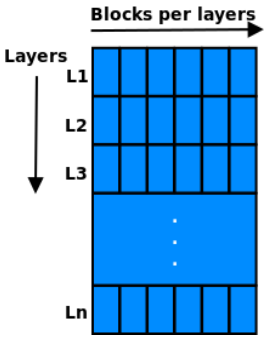
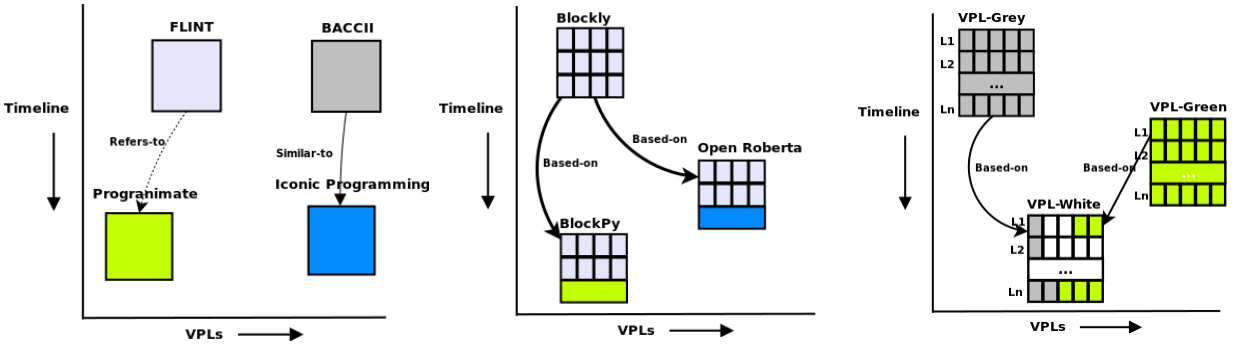


Fig. 3: A typical VPL of the third generation

Figure 4b, shows that VPLs of the second generation provide blocks, leading to easy to reuse codebase of VPLs in this generation. However, in the second generation VPLs, there are three shortcomings. Firstly, reusing a second generation VPL requires understanding some of its code and skills to program in a TPL, thus hindering the reusability of that VPL. Secondly, the second generation of VPLs does not provide layers, therefore a functionality is not confined to a specific layer designated to it. Hence any change in their underlying structure, requires a major revamping in different parts of such a VPL. Finally, second generation of VPL are mutually incompatible as their design is not based on a common framework. Hence, a new VPL being developed cannot reuse multiple incompatible existing VPLs codebase.

Fig. 4c, shows that a third generation VPL could reuse the codebase of several existing VPLs, if these existing VPLs were developed using the same universal framework, VisFra. VisFra provides layers hence confines each functionality to a specific layer designated to it. The layer approach enables making a major change in an existing VPL easy, without effecting other layers providing different functionalities. Finally, VisFra enables creating a new VPL using drag-and-drop of different components thus eliminating the need to write code in any TPL. The VPL-white in Fig. 4c uses many of the components of VPL-gray and VPL-green at different layers of its development.

We are currently developing the universal framework VisFra. It defines ten different layers at different maturity levels. Based on this framework, we plan to reproduce selected VPLs and produce new VPLs to demonstrate its effectiveness. The details of the framework will be discussed in a future paper.



a) VPLs of the First Generation **b) VPLs of the Second Generation** **c) VPLs of the Third Generation**

Fig. 4: Evolution of VPLs Generations

6. References

[1] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B. D. & Wilusz, T. (2001, December). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In Working group reports from ITiCSE on Innovation and technology in computer science education (pp. 125-180). ACM.

[2] Jenkins, T. (2002, August). On the difficulty of learning to program. In Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences (Vol. 4, No. 2002, pp. 53-58).

[3] Chang, C. K. (2014). Effects of using Alice and Scratch in an introductory programming course for corrective instruction. Journal of Educational Computing Research, 51(2), 185-204.

[4] Knuth, D. E., & Pardo, L. T. (1980). The early development of programming languages. In A history of computing in the twentieth century (pp. 197-273).

[5] Chao, P. Y. (2016). Exploring students' computational practice, design and performance of problem-solving through a visual programming environment. Computers & Education, 95, 202-215.

[6] Weintrop, D., & Wilensky, U. (2017). Comparing block-based and text-based programming in high school computer science classrooms. ACM Transactions on Computing Education (TOCE), 18(1), 3.

[7] Bart, A. C., & Kafura, D. (2017, March). BlockPy Interactive Demo: Dual Text/Block Python Programming Environment for Guided Practice and Data Science. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (pp. 639-640). ACM.

[8] Muhammad Idrees, Faisal Aslam, Syed Mansoor Sarwar, and Khurram Shahzad. "Contextual Ranking of Visual Programming Languages." Manuscript submitted for publication.

[9] Flowchart interpreter. http://vardanyan.am/fi/. Accessed: 2016-08-10.

[10] Rizvi, M., & Humphries, T. (2012, October). A Scratch-based CS0 course for at-risk computer science majors. In Frontiers in Education Conference (FIE), 2012 (pp. 1-5). IEEE.

[11] Harvey, B., Garcia, D. D., Barnes, T., Titterton, N., Miller, O., Armendariz, D. & Paley, J. (2014, March). Snap!(build your own blocks). In Proceedings of the 45th ACM technical symposium on Computer science education (pp. 749-749). ACM.

[12] Rouly, J. M., Orbeck, J. D., & Syriani, E. (2014, October). Usability and suitability survey of features in visual ides for non-programmers. In Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools (pp. 31-42). ACM.

[13] Stencyl.-http://www.stencyl.com/. Accessed: 2016-08-16.

[14] Horspool, R. N., & Tillmann, N. (2013). TouchDevelop: programming on the go. Apress.

[15] Tillmann, N., Moskal, M., de Halleux, J., Fahndrich, M., & Xie, T. (2012, April). Engage your students by teaching computer science using only mobile devices with touchDevelop. In Software Engineering Education and Training (CSEE&T), 2012 IEEE 25th Conference on (pp. 87-89). IEEE.

[16] Touch develop. https://www.touchdevelop.com/. Accessed: 2016-08-16.

[17] Touch develop export. https://www.touchdevelop.com/docs/export-toapp. Accessed: 2016-08-16.

[18] N Fraser et al. Blockly: A visual programming editor. URL: https://code. google. com/p/blockly, 2013.

[19] Marron, A., Weiss, G., & Wiener, G. (2012, October). A decentralized approach for programming interactive applications with javascript and blockly. In Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions (pp. 59-70). ACM.

[20] Code dot org. https://code.org/about. Accessed: 2016-08-16.

[21] Kalelioğlu, F. (2015). A new way of teaching programming skills to K-12

students: Code. org. Computers in Human Behavior, 52, 200-210.

[22] Ketterl, M., Jost, B., Leimbach, T., & Budde, R. (2016). Tema 2: Open Roberta-A Web Based Approach to Visually Program Real Educational Robots. Tidsskriftet Læring og Medier (LOM), 8(14).

[23] mblock. http://www.mblock.cc/. Accessed: 2016-08-16.

[24] Bau, D., Bau, D. A., Dawson, M., & Pickens, C. (2015, June). Pencil code: block code for a text world. In Proceedings of the 14th International Conference on Interaction Design and Children (pp. 445-448). ACM.

[25] Open source graphical programming for design. http://dynamobim.org/. Accessed: 2017-09-01.

[26] Flowgorithm. http://www.flowgorithm.org/. Accessed: 2016-08-10.

[27] Díaz, M., & Luis, J. (2016). REMGRAFEE TOOL: Herramienta para el estudio de los sistemas basados en reglas mediante grafos RETE.

[28] Kumar, D. (2014). Digital playgrounds for early computing education. ACM Inroads, 5(1), 20-21.

[29] García-Peñalvo, F. J., Rees, A. M., Hughes, J., Jormanainen, I., Toivonen, T., & Vermeersch, J. (2016, November). A survey of resources for introducing coding into schools. In Proceedings of the Fourth International Conference on Technological Ecosystems for Enhancing Multiculturality (pp. 19-26). ACM.

[30] Viple. http://neptune.fulton.ad.asu.edu/VIPLE/. Accessed: 2016-08-16.

[31] Chen, Y., & De Luca, G. (2016, May). VIPLE: visual IoT/robotics programming language environment for computer science education. In 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 963-971). IEEE.

[32] Romagosa Carrasquer, B. (2016). From the turtle to the beetle.

[33] Bart, A. C., Tilevich, E., Shaffer, C. A., & Kafura, D. (2015, October). Position paper: From interest to usefulness with BlockPy, a block-based, educational environment. In Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE (pp. 87-89). IEEE.

[34] Chin, J. M., Chin, M. H., & Van Landuyt, C. (2013). A String Search Marketing Application Using Visual Programming. e-Journal of Business Education and Scholarship of Teaching, 7(2), 46-58.

[35] S Mitkin. Drakon: The human revolution in understanding programs. http://drakon-editor.sourceforge.net/DRAKON.pdf, 2011.

[36] Travis, J., & Kring, J. (2007). LabVIEW for everyone: graphical programming made easy and fun. Prentice-Hall..

[37] Johnson, G. W. (1997). LabVIEW graphical programming. Tata McGraw-Hill Education.

[38] Wells, L. K., & Travis, J. (1997). LabVIEW for everyone: graphical programming made even easier. Upper Saddle River, NJ: Prentice Hall PTR.

[39] What is labview? http://www.ni.com/en-lb/shop/labview.html. Accessed: 2017-09-01.

[40] Repenning, A. (2000). AgentSheets®: An interactive simulation environment with end-user programmable agents. Interaction.

[41] Agentsheets. http://www.agentsheets.com/. Accessed: 2016-08-16.

[42] Calloni, B. A., & Bagert, D. J. (1994, March). Iconic Programming in BACCII vs. Textual Programming: which is a better learning environment?. In ACM SIGCSE Bulletin (Vol. 26, No. 1, pp. 188-192). ACM.

[43] Calloni, B. A., Bagert, D. J., & Haiduk, H. P. (1997, March). Iconic programming proves effective for teaching the first year programming sequence. In ACM SIGCSE Bulletin (Vol. 29, No. 1, pp. 262-266). ACM.

[44] Lawhead, P. B., Duncan, M. E., Bland, C. G., Goldweber, M., Schep, M., Barnes, D. J., & Hollingsworth, R. G. (2002, June). A road map for teaching introductory programming using LEGO© mindstorms robots. In ACM SIGCSE Bulletin (Vol. 35, No. 2, pp. 191-201). ACM.

[45] Kay, A. (2005). Squeak Etoys authoring & media. Viewpoints Research Institute.

[46] Conway, M., Audia, S., Burnette, T., Cosgrove, D., & Christiansen, K. (2000,

April). Alice: lessons learned from building a 3D system for novices. In Proceedings of the SIGCHI conference on Human Factors in Computing Systems (pp. 486-493). ACM.

[47] Cooper, S., Dann, W., & Pausch, R. (2000, April). Alice: a 3-D tool for introductory programming concepts. In Journal of Computing Sciences in Colleges (Vol. 15, No. 5, pp. 107-116). Consortium for Computing Sciences in Colleges.

[48] Alice. http://www.alice.org/. Accessed: 2016-08-10.

[49] Bhargava, H. K., Sridhar, S., & Herrick, C. (1999). Beyond spreadsheets: tools for building decision support systems. Computer, 32(3), 31-39.

[50] Sfc website. http://watts.cs.sonoma.edu/SFC/. Accessed: 2016-08-10.

[51] Ziegler, U., & Crews, T. (1999, March). An integrated program development tool for teaching and learning how to program. In ACM SIGCSE Bulletin (Vol. 31, No. 1, pp. 276-280). ACM.

[52] Materson, T. F., & Meyer, R. M. (2001). SIVIL: a true visual programming language for students. Journal of Computing Sciences in Colleges, 16(4), 74-86.

[53] Carlisle, M. C., Wilson, T. A., Humphries, J. W., & Hadfield, S. M. (2005). RAPTOR: a visual programming environment for teaching algorithmic problem solving. Acm Sigcse Bulletin, 37(1), 176-180.

[54] Nikunja Swain, P. E. (2013). Raptor-A vehicle to enhance logical thinking. Journal of Environmental Hazards, 7(4), 353-359.

[55] Raptor, a flowchart-based programming environment.http://raptor.martincarlisle.com/. Accessed: 2016-08-10.

[56] Perrin, E., Linck, S., & Danesi, F. (2012). Algopath: A new way of learning algorithmic. In in The Fifth International Conference on Advances in Computer-Human Interactions.

[57] Logic of algorithms for resolution of problems (larp). http://larp.marcolavoie.ca/en/default.htm. Accessed: 2016-08-10.

[58] Visual logic. http://www.visuallogic.org/. Accessed: 2016-08-10.

[59] Chen, S., & Morris, S. (2005, June). Iconic programming for flowcharts, java, turing, etc. In ACM SIGCSE Bulletin (Vol. 37, No. 3, pp. 104-107). ACM.

[60] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... & Kafai, Y. (2009). The magazine archive includes every article published in Communications of the ACM for over the past 50 years. Communications of the ACM, 52(11), 60-67.

[61] Fincher, S., Cooper, S., Kölling, M., & Maloney, J. (2010, March). Comparing alice, greenfoot & scratch. In Proceedings of the 41st ACM technical symposium on Computer science education (pp. 192-193). ACM.

[62] Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: urban youth learning programming with scratch (Vol. 40, No. 1, pp. 367-371). ACM.

[63] Scratch. https://scratch.mit.edu/. Accessed: 2016-08-10.

[64] Koorsse, M., Cilliers, C., & Calitz, A. (2015). Programming assistance tools to support the learning of IT programming in South African secondary schools. Computers & Education, 82, 162-178.

[65] Greyling, J. H., Cilliers, C. B., & Calitz, A. P. (2006, July). B#: The development and assessment of an iconic programming tool for novice programmers. In Information Technology Based Higher Education and Training, 2006. ITHET'06. 7th International Conference on (pp. 367-375). IEEE.

[66] Microsoft vpl. https://msdn.microsoft.com/enus/library/bb483088.aspx. Accessed: 2016-08-16.

[67] Klassner, F., & Anderson, S. D. (2003). Lego MindStorms: Not just for K-12 anymore. IEEE Robotics & Automation Magazine, 10(2), 12-18.

[68] Lawhead, P. B., Duncan, M. E., Bland, C. G., Goldweber, M., Schep, M., Barnes, D. J., & Hollingsworth, R. G. (2002, June). A road map for teaching introductory programming using LEGO© mindstorms robots. In ACM SIGCSE Bulletin (Vol. 35, No. 2, pp. 191-201). ACM.

[69] Resnick, M. (1996, April). StarLogo: an environment for decentralized modeling and

decentralized thinking. In Conference companion on Human factors in computing systems (pp. 11-12). ACM.

[70] Wang, K., McCaffrey, C., Wendel, D., & Klopfer, E. (2006, June). 3D game design with programming blocks in StarLogo TNG. In Proceedings of the 7th international conference on Learning sciences (pp. 1008-1009). International Society of the Learning Sciences.

[71] Scott, A., Watkins, M., & McPhee, D. (2008, July). Progranimate-A Web Enabled Algorithmic Problem Solving Application. In CSREA EEE (pp. 498-508).

[72] Scott, A., Watkins, M., & McPhee, D. (2008, April). E-Learning For Novice Programmers; A Dynamic Visualisation and Problem Solving Tool. In Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on (pp. 1-6). IEEE.

[73] Progranimate. http://www.progranimate.com/. Accessed: 2016-08-10.

[74] Sourceforge devflowcharter. http://devflowcharter.sourceforge.net/. Accessed: 2016-08-16.

[75] DeQuadros, M. (2012). GameSalad Beginner's Guide. Packt Publishing Ltd.

[76] Roy, K., Rousse, W. C., & DeMeritt, D. B. (2012, October). Comparing the mobile novice programming environments: App Inventor for Android vs. GameSalad. In Frontiers in Education Conference (FIE), 2012 (pp. 1-6). IEEE.

[77] MacLaurin, M. (2009, April). Kodu: end-user programming and design for games. In Proceedings of the 4th international conference on foundations of digital games (p. 2). ACM.

[78] MacLaurin, M. B. (2011, January). The design of Kodu: A tiny visual programming language for children on the Xbox 360. In ACM Sigplan Notices (Vol. 46, No. 1, pp. 241-246). ACM.

[79] Roberts, R. (2011). Google App Inventor. Packt Publishing Ltd.