Sveučilište u Rijeci
University of Rijeka
http://www.uniri.hr

Polytechnica: Journal of Technology Education, Volume 2, Number 1 (2018)
Politehnika: Časopis za tehnički odgoj i obrazovanje, Volumen 2, Broj 1 (2018)

Politehnika
Polytechnica
http://www.politehnika.uniri.hr
cte@uniri.hr

Stručni članak
Professional article
UDK 004.42
004.43

Introducing a Dataflow visual programming language for understanding program execution*

Marin Aglić Čuvić
Faculty of Science
University of Split
Ruđera Boškovića 33, 21000 Split
marin.aglic.cuvic@pmfst.hr

Abstract

Regardless of the programming experience, the understanding of the program execution is mandatory if a programmer is to write a code. Therefore, it is vitally important for novice programmers to construct correct mental models of the execution of the notional machine. To this end, many program visualizations have been developed over the last years. However, novice programmers often focus on learning the syntax of a programming language rather than getting to grips with the programming itself. Dataflow visual programming languages (DFVPL) allow us to build programs by connecting blocks with arcs. In this paper we present our own DFVPL that exhibits a high level of responsiveness to user inputs and enables the user to control the execution of the program.

Keywords: *dataflow; dataflow visual programming; learning programming; program visualizations*.

1. Introduction

In order to learn programming, novices are often required to learn the syntax and semantics of a programming language and develop an understanding of how programs are executed at the level of abstraction provided by the language itself (Hidalgo-Céspedes, Marín-Raventós, Lara-Villagrán, 2016), (Gomes, Mendes, 2007). Given the fact that students usually find learning to program overwhelming for them, they tend to focus on learning the syntax rather than learning to program. Furthermore, novices often gain partial or incomplete understanding of programming concepts, which impedes the acquisition of new knowledge and is likely to result in students failing to solve their programming tasks. Researchers have been dealing with programming misconceptions for decades, simultaneously developing new

** Paper is accepted for the 20th CARNET Users Conference CUC, 2018, Šibenik, Croatia*

programming languages and tools that would facilitate the process of learning programming.

Scratch is probably the best-known programming language developed for this purpose. Since it is a block-based programming language, where students drag and drop blocks into their appropriate slots, the syntax of the language has been almost disregarded (Maloney et al., 2010). This allows students to focus on building programs rather than learning the syntax. However, university students quickly outgrow or get bored of Scratch, and express their interest in an industry level programming language.

Apart from programming languages, researchers have developed program visualizations that graphically depict how programs are executed. The purpose of these systems is to provide novice programmers with the correct mental model of the *notional machine*. The notional machine is an abstract "construct formed from concepts provided by the programming language" (Čuvić, Maras, Mladenović, 2017) which gives a sufficiently detailed insight into the program execution (Sorva, 2013). A mental model of a notional machine is a mental representation of that machine held by a programmer.

Just like the aforementioned programming language Scratch, Dataflow Visual Programming Languages (DFVPL) are block-based languages connected by arcs (also called wires) where data flows between blocks (Johnston, Hanna, Millar, 2004), (Hils, 1992). A program in a DFVPL has a graph-like structure. Similarly to Scratch, DFVPLs almost completely eliminate the need for learning the language syntax. Furthermore, they make it simple for the user to see the transformation of data as it flows, in the manner that is similar to how program visualizations depict the state of variables. Since DFVPLs make it easy to construct programs, there have been numerous applications of DFVPLs, such as the construction of user interfaces, image processing, music, graphics, general-purpose programming and education (Hils, 1992).

Although these languages simplify the construction of programs, they do not allow the user to examine how these complex functions are actually implemented. Furthermore, only a small subset of them, e.g. Show and Tell (Hils, 1992), enable novice programmers to learn programming. We believe that some of the key features of program visualizations included in a DFVLP may facilitate the process of learning to program. Therefore, we present a DFVLP that i) allows the user to control the execution of the language while visualizing which expressions are currently executed, ii) explicitly displays the control-flow of a program, and iii) provides blocks that are conceptually similar to Python instructions.

2. Program visualizations and education

During the process of writing a program, programmers usually consult their mental model of the notional machine in order to understand and draw inferences about the behaviour of the program (Sorva, 2013). However, when it comes to novice programmers, these mental representations are often faulty and incomplete, thus preventing them from solving their programming tasks. They are often formed intuitively based on analogies and prior experience with similar systems. The problem is that even though it is simple to construct a mental model, changing it takes much more effort (Schumacher, Czerwinski, 1992). Therefore, it is necessary for teachers to provide students with a correct mental model prior to learning programming. This is the reason why program visualizations have been developed.

Program visualization is a term that generally refers to the use of graphical elements for depicting the execution of the notional machine (Hidalgo-Céspedes, Marín-Raventós, Lara-Villagrán, 2016). Since the notional machine is composed of the concepts related to a programming language, each language may have its own notional machine. In his literature review (Sorva, Karavirta, Malmi, 2013), Sorva identified forty-six different program visualizations that

appeared from 1979 to 2013. This list was expanded by J. Hidalgo-Céspedes (Hidalgo-Céspedes, Marín-Raventós, Lara-Villagrán, 2016) who added the ones emerging in the period between 2013 and 2016. Some of the best-known program visualizations are UUhistle (Sorva, Sirkia, 2010), Online Python Tutor (Guo, 2013) and Jeliot 3 (Moreno, Myller, Sutinen, 2004).

UUhistle is a visualization system that not only visualizes program execution, but also allows users to assume the role of the machine and simulate program execution (Sorva, Sirkia, 2010). Online Python Tutor is a web-based visualization system that currently supports eight programming languages (counting Python 2 and Python 3 as two different languages), but it could be embedded into other web pages as well (Guo, 2013). Moreover, Online Python Tutor has a live programming mode which updates the graphical elements of the visualization as the user types the code, thus allowing users to observe the changes in program behavior in real-time. Jeliot 3 is a visualization system that was developed many years ago. Its goal is to facilitate the learning of both procedural and object-oriented programming (Moreno, Myller, Sutinen, 2004).

All program visualizations enable the users to write and visualize their own code, step by step through program execution, and to show the current state of variables during program execution. These are the core features that permit the system to visualize program execution and therefore allow for the user's basic interactivity with the system. This is increasingly important if we take into consideration the fact that novices might want to return to a certain point in a program and repeat some steps. For this very reason, these features were incorporated in our DFVPL.

It is worth noting that it is necessary to keep the visualizations as simple as possible and to avoid having too much animations. As Moreno observed in reference to Jeliot 3, too much repetitions might reduce an animation to a "movie of moving boxes" (Moreno, Joy, 2007), where students no longer think about what is happening with the notional machine and may miss out on their meaning.

3. Dataflow visual programming languages

Dataflow visual programming languages (DFVPL) have been studied for more than three decades (Gauvin, Paquet, Freiman, 2015). DFVPLs are in fact block-based languages where blocks (also known as nodes) are connected by arcs (or wires). Therefore, a program in a DFVPL is a directed graph through which data flow between blocks and each block has a function that may allow for the transformation of the received data (Johnston, Hanna, Millar, 2004), (Hils, 1992). However, it is possible to have blocks that accept no input data and those that produce no output data.

Different DFVPLs have been developed for various application domains. In these DFVPLs, blocks provide functions that are specific to the intended use of the DFVPL. This is quite similar to the concept of a notional machine. For example, Orange3 ("Orange3", 2018) is a DFVPL that provides block for statistical analysis and machine learning. Hence, blocks perform high level functions such as training neural networks or displaying data in a dataset. This allows a simpler way of constructing programs that, in the case of Orange3, is analysing the data. Other application domains of DFVPLs have been already mentioned in the introductory part of this paper. In his paper, Hils (Hils, 1992) grouped DFVPLs according to their application domain and the number of design alternatives. In regard to DFVPL design alternatives, we will only give a brief summary of i) modes of execution and ii) level of liveliness, as these are the most significant for this discussion.

3.1. Modes of execution

The execution of nodes in a DFVPL may be either data-driven or demand-driven (Hils, 1992). When it comes to data-driven execution, the execution of nodes starts as soon as the data on the input nodes become available. In this execution mode, the data flow downstream. In accordance with the terminology (Hils, 1992), downstream nodes are those that are found by following the node's

output arcs, while upstream nodes are defined as those that are found by going backwards via node's input arcs. Conversely, in demand-driven execution, the execution of a certain node requires data from another node's output arc. If needed, the node in question requests data from upstream nodes through its input arcs and waits until the data become available. Once the data are available, the node sends them through its output arcs to other nodes.

In a data-driven execution mode, all nodes are executed, even though some of the computations are not used (Hils, 1992). By contrast, in a demand-driven execution mode, only those nodes whose data is requested are executed. However, demand-driven execution is more complex and it is used less often.

3.2. Levels of liveliness

Liveliness is measured using a four-level scale (Hils, 1992). At the "informative" level, which is the first one, visual representations are used as documentation for the program. The second level is termed "informative and significant". At this level, visual representations of the program are executable. In comparison with the second level, the third one is enhanced by responsiveness, which means that the program is executed each time the user enters input data or edits the program. Finally, the fourth level refers to the systems that are "live". At this level of liveliness, the system continually updates its display to show the new data that are being processed, as well as results.

4. A Dataflow Visual Programming Language for novice programmers

At the Faculty of Science in Split, we have developed a DFVPL prototype aimed at helping novice programmers understand program execution. The DFVPL is built with JavaScript and it is completely web-based.

When designing our DFVPL we wanted to i) have a system that can be used to demonstrate Python programs ii) allow the user to control the execution of program in a step-by-step fashion iii) have a system that has a high level of liveliness.

In what follows, we will discuss our design decisions.

4.1. The case of Python

Python is a general-purpose programming language that is often used in introductory programming courses at universities due to its simple syntax. Since the language is widely used in introductory programming courses, we wanted our DFVPL to be conceptually similar to Python. One of the problems that we faced was related to the implementation of simple branching statements into our language. Branching as it is done in Python does not apply well to the dataflow model used by our language since there is no data flowing in an *if-else* statement (tertiary operators excluded). Furthermore, dataflow languages typically use blocks, such as merge and switch, for branching (due to space limitations, see reference for details). Therefore, our solution was to implement special control-flow blocks that correspond to *if*, *if-else*, *elif* and *elif-else* statements. Since these control-blocks do not allow data-flow, they must be connected to variables or other nodes that might produce data, or the ones whose data can be inferred while parsing.

Furthermore, the way certain blocks function has been adjusted so as to simulate Python statements, e.g. print. When the print block is used, it will print out the result in a special program output area. Print can also have an arbitrary number of inputs which are then printed in accordance with their y-position in the workspace.

4.2. The Graph Engine

In designing our DFVPL, our second aim was to enable users to control the execution of the program in a step-by-step fashion that is visually enhanced, thus allowing for the display of blocks that are currently being executed. This is something that is typical of program visualizations in which currently executed lines are pointed to

or highlighted. This is not characteristic of DFVPLs. Furthermore, when dealing with control structures such as *if-then-else* statements, we wanted to draw users' attention to the fact that a certain branch will not be executed. Program visualizations make this possible by taking into account which branch is going to be executed when calculating the total number of steps for the user.

This lead to the development of a component we call the Graph Engine (GE). The Graph Engine is a central component of the DFVPL that consists of a parser, execution service and background execution service.

The input to the parser is a program that the users create. The parser then transforms the graph into a key-value pair data structure. Each key is a step number, and the value contains all of the nodes that will be executed in a certain step. Therefore, the number of keys determines the number of steps. In case a *for* loop is found during parsing, the appropriate number of times the parser embeds of the loop body into the resulting Map.

During parsing, when a branching node is discovered, the background execution service is called. The task of the background execution service is to mark arcs as either inactive or active, depending on which arcs are going to be executed by the user. Nodes that are downstream from the inactive arcs are ignored by the parser.

Finally, the execution service is a component of the GE that executes the nodes when the user steps through the program. If the user steps forward through the program, the execution service executes only the next step (the previous being already executed). In case of going backwards through the program, default values are restored to the nodes following the new step.

4.3. The liveliness of the system

As discussed earlier in this paper, there are four levels of liveliness. Our DFVPL reparses and re-executes the program each time the user enters a new value or modifies the program. The execution service re-executes the program up to the current execution step at which the user is currently. In addition to that, all of the visuals and variable values are updated accordingly. Taking into account the previous discussion, it is fair to conclude that our DFVPL falls into the third level of liveliness at least.

5. Conclusion

In this paper we provided a brief insight into notional machines and program visualizations and illustrated the importance of constructing a correct mental model of the notional machine. Afterwards, we discussed Dataflow Visual Programming Languages (DFVPLs) and some of their features. Finally, we presented the DFVPL that has been developed at the Faculty of Science in Split. Its main features are as follows: i) blocks are functionally very similar to Python statements, ii) the system is at a high level of liveliness, and iii) the system supports some of the features typical of program visualizations.

References

Čuvić, M. A., Maras, J., Mladenović, S. (2017). Extending the object-oriented notional machine notation with inheritance, polymorphism, and GUI events. *Proceedings of 40th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2017*.

Gauvin, S., Paquet, M., Freiman, V. (2015). Vizwik - visual data flow programming and its educational implications. In S. Carliner, C. Fulford & N. Ostashewski (Eds.), *Proceedings of EdMedia 2015--World Conference on Educational Media and Technology*, 1594-1600. Montreal, Quebec, Canada: Association for the Advancement of Computing in Education (AACE).

Gomes, A., Mendes, A. J. N. (2007). Learning to program-difficulties and solutions. International Conference on Engineering Education - ICEE 2007, Coimbra, Portugal.

Guo, P. J. (2013). Online python tutor: Embeddable web-based program visualization for cs education. *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, 579-584.

Hidalgo-Céspedes, J., Marín-Raventós, G., Lara-Villagrán, V. (2016). Learning principles in program visualizations: a systematic literature review. 2016 IEEE Frontiers in Education Conference (FIE), 1-9. IEEE.

Hils, D. D. (1992). Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1), 69-101.

Johnston, W. M., Hanna, J. R. P., Millar, R. J. (2004). Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36(1), 1-34.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E. (2010). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, 10(4), 1-15.

Moreno, A., Joy, M. S. (2007). Jeliot 3 in a Demanding Educational Setting. *Electronic Notes in Theoretical Computer Science*, 178, 51-59.

Moreno, A., Myller, N., Sutinen, E. (2004). Visualizing Programs with Jeliot 3. *Proceedings of the working conference on Advanced visual interfaces*, 373-376.

Orange3 (2018). Retrieved 30. 04. 2018. from https://orange.biolab.si.

Schumacher, R. M., Czerwinski, M. P. (1992). Mental Models and the Acquisition of Expert Knowledge. In: Hoffman R. R. (eds), *The Psychology of Expertise*, 61-79. Springer, New York.

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13(2), 1-31.

Sorva, J., Karavirta, V., Malmi, L. (2013). A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education*, 13(4), 1-64.

Sorva, J., Sirkia, T. (2010). UUhistle: a software tool for visual program simulation. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, 49-54.