Visual programming language for modular algorithms

Rudolfs Opmanis, Rihards Opmanis

Institute of Mathematics and Computer Science University of Latvia, Raina bulvaris 29, Riga, LV-1459, Latvia
rudolfs.opmanis@gmail.com, rihardso@latnet.lv

Abstract: *In this paper we present a way how to describe modular algorithms visually using building block approach. We are using graph drawings to represent algorithm flow. Presented algorithm description language allows to develop algorithms faster, see intermediate results, execute the whole algorithm or just part of it and to run it without setting up complex build environment. In theoretical computer science flow charts are widely used to describe algorithm visually, but it is not possible to execute them. As a result in order to execute it programmer needs to transfer graphical algorithm description into some general textual programming language which is error-prone and time consuming process. Interactive graphical programming environments are available for learning computer programming, but since they are not extendable and provide only limited number of language constructions so they are not applicable for implementing advanced algorithms. We use graph recognition and graph generation algorithms as a use case to demonstrate possible uses of the presented visual programming language.*

Keywords: visual programming language, block diagram, data-flow programming, modular algorithm.

Introduction

Visual computer science alone there are many examples when visual representation of information is preferred over textual. One of most obvious examples is database schema which almost always is shown in form of a ER diagram, but not in textual description. That could be explained by the nature of data, because in database schemas we usually have tables that are linked together with many relationships. That kind of data is hard describe in organized, sequential fashion as text description would demand and it is easier to present drawing to the user and let him/her to decide the order in to traverse the whole data model.

Algorithm that is meant to be executed by a computer can be written in sequential form, mainly because processor of a computer can execute instructions only in determined sequential fashion so we believe that this is the reason why textual programming languages are so popular. Nature of computer algorithm description is sequential enough to be logically described in textual form while still being understandable by programmers.

To implement algorithm in textual programming language in addition to thinking about algorithm itself programmers need to know pretty complicated syntax rules of the language and think about how or organize code so that is easy to test, maintain and extend. While implementing algorithms for graph layout and optical graph recognition we faced several issues.

The second chapter defines type of algorithms that could be developed with proposed visual programming language, but in the third chapter we describe the problem that we faced while working on graph layout and recognition algorithms with textual programming languages. In the fourth chapter we demonstrate our proposed visual programming language.

Nowdays when almost everyone has a personal computer only small part of computer users can write even the simplest “Hello world” program. Classic textual programming languages and programming itself is difficult to learn because it requires skills that many people do not have (Lewis, 1987). It is also a lot harder for software developers to predict all possible scenarios and design complex system that will work and give good results with all possible sets of options that user can provide. So we have to find ways how delegate some part of programming to computer users or users with specific non-programming knowledge (Whitley, 1997). Almost every computer user knows how to work with spread-sheets and how to make simple calculations – visual programming language enables even bigger possibilities for these users to make more complex calculations without learning complex computer programming but using only simple statements to specify some things like where to get input and how to present the results.

Visual programming language is expression tool for visual programming. Visual programming refers to any system that allows the user to specify a program is a two dimensional fashion (Myers, 1990) (Myers, 1986).

Two-dimensional displays for programs, such as flowcharts and even the indenting of block structured programs, have long been known as helpful aids in program understanding (Smith, 1977)

Visual programming languages are designed to make programs from ready-made sub-programs, functions or blocks. Typically visual programming languages create data flow graph – thus why it is called dataflow programming (Bragg, 1994).

Big advantage for visual programming languages is that it is easy to modify program data flow so it is easier to test and develop new version of specific block because it is easy to give input to both of them and then compare results.

Definitions

In this paper we consider only one subclass of algorithms that we call modular. Algorithm is modular algorithm if it can be divided into sub-algorithms (modules) that can be executed in fixed order one after another. For example graph layout algorithm that first positions nodes on the plane, then routes edges and lastly positions all labels next to their owner objects could be considered as modular algorithm, because it can be divided in three sub-algorithms. The first sub-algorithm would be node positioning algorithm, then the second sub-algorithm would be edge routing algorithm, but the last one is label positioning algorithm. Each of these three algorithms uses output from the previous one, but they do not depend on some internal properties of each other. With modular algorithms if we have multiple options for some sub-algorithm we can easily swap them and easily tailor result of the whole algorithm to our needs. For example in previously mentioned graph layout algorithm we could have two implementations of edge router sub-algorithm one would route edges as line segments directly connecting source and target nodes, but the other edge router could route edges with bends to ensure that there are no node-edge overlaps. If we have these two implementations we can use either one of them and graph layout algorithm output also changes depending on selected algorithm.
On the other hand we do not consider insertion sort or binary search algorithms to be modular because they do not have natural division of sub-algorithms that could be executed in specified order.
We call implementation stable if it produces the same result for equal input data. This means that running implemented algorithm with equal data we expect that execution order of all statements will be the same. This is property of implementation is important to guarantee the outcome of algorithm. Most typical causes of non-stable implementations of non-random algorithms are caused by iterations over unsorted collections like sets of maps.

Problem statement

While developing algorithms for image processing and data visualization we observed that we spend huge amount of time for implementing algorithm prototypes rather than coming up with ideas for new solutions. In algorithm implementation phase we organized already implemented algorithms in general structures so that they could be easily reused, but still linking together all necessary sub-algorithms in the implementation of theoretically plausible algorithm required significant amount of time. So it was necessary to come up with solution that would allow us to connect pre-existing sub algorithms faster.
Another issue that we faced was that every next iteration of algorithms prototype was evolution of the previous version with rather minimal changes. Moreover it may as well be that the next version not better than the previous version so it is necessary to run both of them simultaneously on the same set of input data to measure which is better. This problem added requirement to be able to keep several almost identical algorithm implementations in parallel, be able to run them at any time and also to quickly see the differences between any two of them. As developed algorithms got complicated it became obvious that working with long textual documents is not the most convenient solution. After observation that all of our prototyped algorithms are modular it became apparent that visual programming values might solve these problems.
During our research for existing visual programming languages we found out that the best fit is KNIME (Berthold, 2006) programming language which was really close to our needs, because it is easily extensible, it already has well developed tool support based on Eclipse RCP (Eclipse RCP, 2013), however the main obstacle of using it for our needs to quickly prototype various modular algorithms were its unstable behaviour. Unstable in this case means that running program on equivalent input data it can produce different output data, but in KNIME it was possible to run defined algorithm and observe changing order of executed sub algorithms. There were also some other less important reasons that prevented using KNIME modular environment: its rather difficult to integrate algorithm developed in KNIME into classical programming languages and also to use it forces to translate all problems into domain of data mining or statistics, because the basic data transfer object between sub algorithms is table.
Because existing visual programming languages didn't provide all required solutions it was decided to create our own data-flow visual programming language that would allow us to quickly develop prototypes, define stable algorithms, load/save algorithm definitions and represent them in intuitive manner.

Architecture

Design of our programming language is based on block diagrams that are vastly used in theoretical computer science. The problem with blocks in block diagrams is that there are fixed number of block types, but contents of each block is not formalized. Complexity of each block in block diagram can be arbitrary detailed it can be just a simple assignment operator or complex algorithm call. To remove this problem when user can create blocks of arbitrary complexity it was decided to allow only those blocks, which we know how to execute i.e. the ones that have implemented specified interfaces and that are registered in language. We don't fix the set of available blocks, but rather define way how they should be implemented and registered in order to use them. Fig. 3 illustrates single building block in our visual programming language. All required inputs are located at the left side of block representing rectangle, and are documented by tooltip text that is shown when mouse hovers over

each of the inputs. Similarly all produced outputs are positioned on the right side of the block and also provides description in tooltip of each input. Blocks name is shown in the centre of the building block, therefore the user can easily follow algorithm flow and more detailed information is provided only by request.

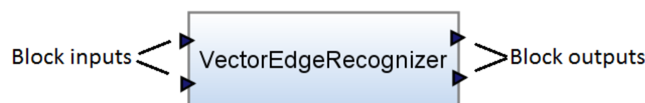

Fig. 3. **Single building block.**

Blocks are linked together by two types of relations. The first type of relation between block is execution order. The second type of relation between blocks is input-output relation. Input-ouput relation is implemented by the common memory that can be accessed from each block. Every block can announce required values and properties that it produces. For each of these inputs and outputs each block instance provides sets of string keys for its inputs and outputs. If the key of some block's required input property is the same as output key of some other block then it means that the first block uses output of the second block. Visually this key equality is represented as line between respective input and output properties. Fig. 4 illustrates two blocks that are linked together. The rectangular block to the left generates a tree graph, but the block to the right can write graph to the file. Thin solid line between "TreeRandomGeneratorBlock" and "TextFileGraphOutputBlock" shows that "TextFileGraphOutputBlock" will use something that is generated as output in "TreeRandomGeneratorBlock" (in this case generated tree graph) as an input. Thicker dashed line on the other hand shows that "TextFileGraphOutputBlock" will be executed right after "TreeRandomGeneratorBlock". Ellipse in the right side is not a block that will be executed, but shows that "TreeRandomGeneratorBlock" has a constant input value for one of required input properties. In this case it shows that "TreeRandomGeneratorBlock" is configured to generate a random tree graph with 100 vertices.

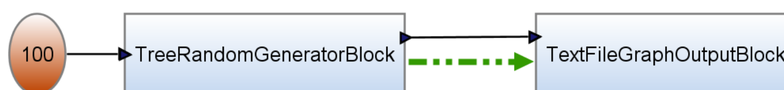

Fig. 4. **Linked building block system.**

More complicated algorithm that is programmed using our visual programming language can be seen in Fig. 5 where one version of opical graph recognition algorithm is visible. According to our experiments with algorithm prototyping his is typical look of any modular algorithm.

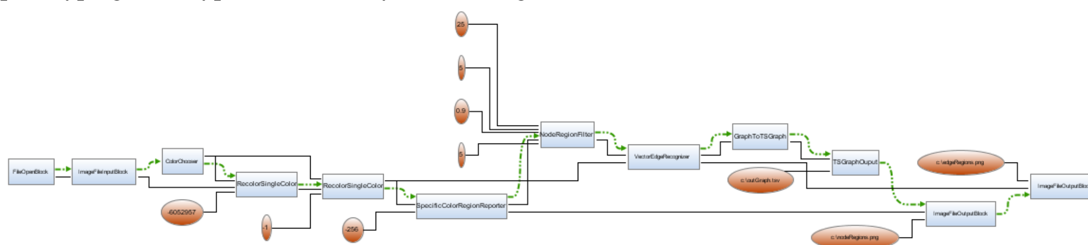

Fig. 5. **More complicated algorithm description.**

Our visual programming language has following features and benefits when compared to traditional textual programming languages:

Interfacing with traditionally written code

When it is necessary to interface visually defined code with some program that is written in conventional programming language such as Java then it simply requires four steps:

1. Read project definition from file or build it with API.
2. Set default values input values
3. Run project
4. Request produced output values from common memory.

Ability to design interactive algorithms faster. By using blocks that presents dialog to the user it is really easy to gather user feedback and adjust future algorithm flow based on entered values. This option might be particularly important to decide if produced results up until now match with desired results. For example when we consider graph layout algorithms it might be useful to allow the user to fine tune automatically positioned node positions before performing automatic edge routing.

Design modular algorithms. Exchanging one block with another requires only two steps. The first step is to create the substitute block, but the second is to reconnect attribute links and control flow path.

Maintain several versions of configurations in parallel and easily compare them. While prototyping it is necessary to maintain some two or more versions of algorithm implementations so that we can compare them, by running on various test data. With textual programming languages it we would need to maintain several classes and use some text file comparison tools to locate the differences between them, but in case of our visual programming language we can perform bipartite matching algorithm between graph objects in both comparable algorithms and visually indicate differences. Our matching allows to draw attention to really significant differences in algorithm structure, but not every single little change in source code like changes in javadoc, additional new line symbols or other coding style improvements.

No need for compilation. Major advantage of our visual programming language over textual programming languages like Java is that our algorithm can be executed without any compilation so compile and run cycle is reduced and prototyping can be done faster.

Implementation details are hidden from the user so developers can use higher abstraction objects. This allows the user to work with higher level concepts and better comprehend the overall picture. This also enables less experienced users to configure complex algorithms, because only general knowledge of each block is necessary.

Easily extendable by implementing general interfaces everyone can extend available set of blocks therefore tailor programming language to any domain where modular algorithms are natural.

Show configuration problems visually. This feature is available in all major IDEs then it shows compilation errors on-the-fly therefore allowing to minimize number of unsuccessful compilation attempts and also visually locating the error. Our visual programming language supports equal functionality. There are certain very simple rules that allows us to show either warning or error message for any block. For example it is considered an error if in control flow there is any block that doesn't have any value or connection associated to its input. This would mean that particular input value is not initialized and it can cause crash in blocs evaluation process.

Building block approach forces to divide implementation in sub blocks and it makes testing easier because it is possible to test each block separately and build algorithms from well tested components.

Our visual programming language allows to look and work with an algorithm from different viewpoints. By default we present algorithm definition in the complete mode (**Fig. 5**) when we illustrate everything that is used to define algorithm: all blocks, execution flow, links connecting produced outputs and required inputs and predefined constant values used for block inputs. It is possible to switch to reduced mode to look separately on control flow (Fig. 6), data flow (Fig. 7) or just block connections (Fig. 8).

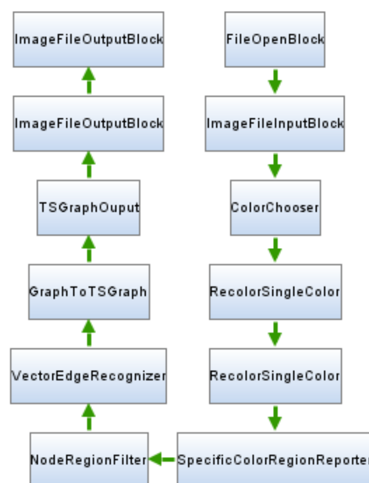

Fig. 6. **Algorithm in reduced mode showing only control flow.**

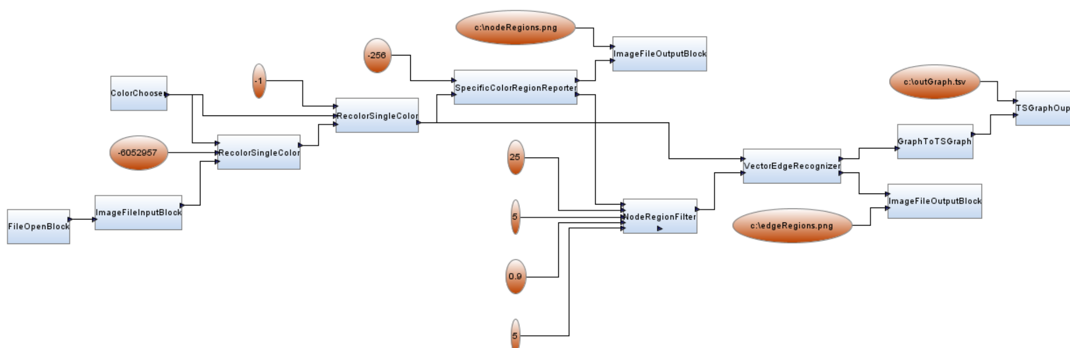

Fig. 7. **Algorithm representation in reduced mode showing data flow.**

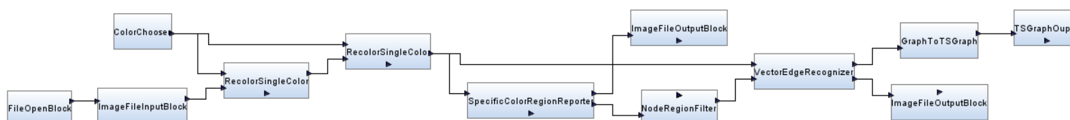

Fig. 8. **Algorithm representation in reduced mode showing just block connections.**

Conclusion

We observed that in this paper presented visual programming language significantly increased speed of our algorithm development and allowed to organize various parallel versions of algorithm prototypes while avoiding from copy-paste errors. In future we should do in-depth usability studies with experienced and non-experienced computer users to measure how and if the presented visual programming language reduces development time

Acknowledgements

Supported by ERAF project 2010/0318/2DP/2.1.1.1.0/10/APIA/VIAA/104

References

Berthold, M.R., Cebron, N., Dill, F., Fatta, G.D., Gabriel, T.R., Georg, F., Meinl, T., Ohl, P., Sieb, C., Wiswedel, B., 2006. Knime: The Konstanz Information Miner. Technical Report (http://www.knime.org/).

Bragg, S. and Driskill, C., 1994. Diagrammatic-graphical programming languages and DoD-STD-2167A, Proc. IEEE Autotestcon, Sept. 1994, pp. 211-220.

Myers, B.A., 1986. Visual programming, programming by example, and program visualization: a taxonomy, Proceedings of the SIGCHI conference on Human factors in computing systems, p.59-66, April 13-17, 1986, Boston, Massachusetts, United States

Myers, B.A., 1990. Taxonomies of visual programming and program visualization, Journal of Visual Languages and Computing, v.1 n.1, p.97-123, March, 1990

Lewis, C. and Olson, G.M., 1987. Can Principles of Cognition Lower the Barriers to Programming?, in *Empirical Studies of Programmers*, Vol. 2, Ablex, 1987.

Smith, D.C., 1997. Pygmalion: A Computer Program to Model and Stimulate Creative Thought. Basel, Stuttgart: Birkhauser, 1977.

Whitley, K.N., 1997. Visual programming languages and the empirical evidence for and against. Journal of Visual Languages and Computing, 8 (1997), pp. 9–142.

http://www.eclipse.org/home/categories/rcp.php