Kopenograms – Graphical Language for Structured Algorithms

Jiří Kofránek[1], Rudolf Pecinovský[2], Petr Novák[3]
[1]Charles University in Prague, Laboratory of Biocybernetics, Prague, Czech Republic
[2]University of Economics, Dept. of Information Technologies, Prague, Czech Republic
[3]HID Global, Prague, Czech Republic

Abstract - *In the OOP era of the present times, it tends to be forgotten sometimes that the design of rather complex algorithms may be ahead of us at the end of object analysis. Several graphical languages are available for their representation. Kopenograms are one of the clearest ways how to represent structured algorithms. They are an apt supplement of UML diagrams used to show algorithmic structures, and they have proven themselves as a very effective tool in programming classes.*

Keywords: Education, Graphical language, Algorithms, Programmiong methodology, UML

1 Introduction

Kopenograms are one of graphical ways of representing algorithms and data. The acronym ***KOPENOGRAM*** expresses the fundamental idea of this graphical representation: ***K**eep **O**ur **P**rogram in **E**mbedded **N**oted **O**blongs for **G**raphical **R**epresentation of **A**lgorithmic **M**odules.*

The original idea of graphical representation of algorithms emerged in the 80ies of the past century, in a discussion of three programmers (Kofránek, Novák, Pecinovský), as a reaction to the defects of existing ways of structured representation of algorithms and data structures (such as flowcharts [1], Jackson diagrams [2, 3], Nassi-Schneiderman diagrams [4] etc.). However, it became apparent soon that the chosen notation can also be used for visually clear demonstration of algorithm and data structuring in teaching. Named kopenograms (originally a temporary name composed of the surnames of the authors) [5] in the then Czechoslovakia and later in the Czech and Slovak Republics, these diagrams started to be used as a teaching aid both for the teaching of programming essentials, e.g. using the programming language Karel [6, 8], and also for the teaching of more advanced programming methods (8-15).

Block is a basic term in any kopenogram, and it means a certain part of the program. A block may be data declaration, class, procedure, function, statement, etc. Every block is shown as a rectangle.

The graphical language of kopenograms includes expression means for writing algorithmic structures as well as data structures. However, UML is now used as standard for writing data structures. Therefore in our contribution, we shall focus only on describing how those algorithmic structures are written for which no diagram is offered by UML that could display them, without "seducing" the developer to using non-structured statements.

2 Algorithmic Structure Blocks

Some blocks exhibit more complex inner structure.:

- **Header** is the upper part of the block divided from the rest of the block using a single or double line. If divided using a double line, it contains the name of the block; if divided using a single line, it contains the input condition of the block.
- **Footer** is the lower part of the block divided using a single line in algorithmic blocks. The footer denotes the end of the body of the cycle and contains the so called output condition of the cycle.
- **Qualification** is the left part of the block, separated using a single vertical line in algorithmic blocks; this part contains access operation for the components of a structured constant or variable.
- **Body** of the block is the remaining part of the block between any header and footer, right of any qualification.
- **Dividing bar** is a special part within the body of the block; this part may contain a condition causing leaving the body of the block prematurely.
- **Compound block** has a body with embedded blocks.
- **Block with several bodies** is a compound block that incorporates several bodies, immediately adjacent horizontally, and separated by single vertical lines.

2.1 Colors

Initially, kopenograms were designed so that the color provides additional information, similarly as colored highlighting of the syntax started to be used later. Colors were assigned to individual types of the blocks and their parts as follows:

- Yellow color is used to fill headers of procedures,

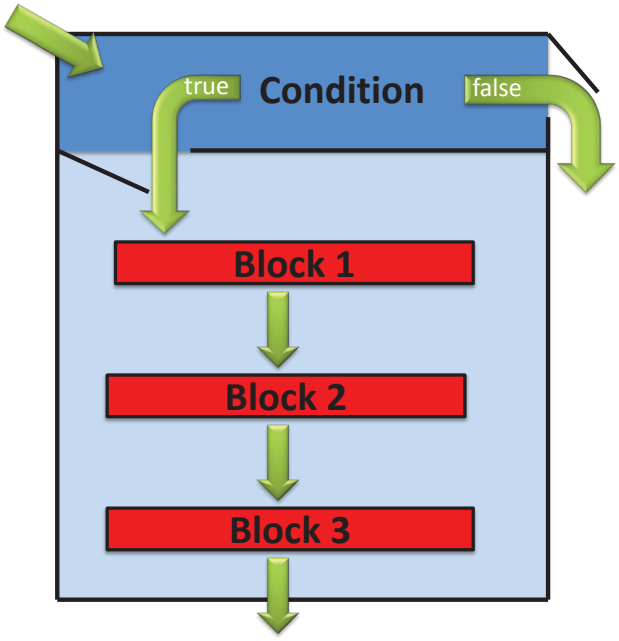

Figure 1: Evaluation flow in kopenogram. Evaluation is started from the upper left corner of the block. If the condition is true, the program continues downward; if false, the program continues to the right.

functions and methods. Yellow is also used to highlight recursive calls.

- Red color is used to fill blocks with actions with the exception of recursive calls, in which case yellow is used as mentioned above.
- Green color is used to fill headers, footers and dividing bars of cycles.
- Blue color is used to fill headers with conditions in conditional statements and switches. It is also used in dividing bars of blocks not used to represent cycles.
- Block qualifications are filled identically as statements, i.e. using red color.

In order to enhance clear arrangement, a lighter shade of the header color is used to fill also the inner area of the given block.

2.2 Comments

Any text written outside a simple block or outside the header, footer and qualification of compound blocks, respectively, is a **comment**. Connecting dashed lines may be used to express relationships of comments to appropriate blocks.

Arrows are a special type of comments; they represent input and output statements. The fact that any data should be input or output, respectively, can be represented by adding an arrow to the right side of a simple block, oriented to the right for the output or to the left for the input, respectively. Then it only remains to write in the block the list of data to be transferred. If the type of the used input/output device or file is to be emphasized, an appropriate schematic mark or name of the file can be drawn to the

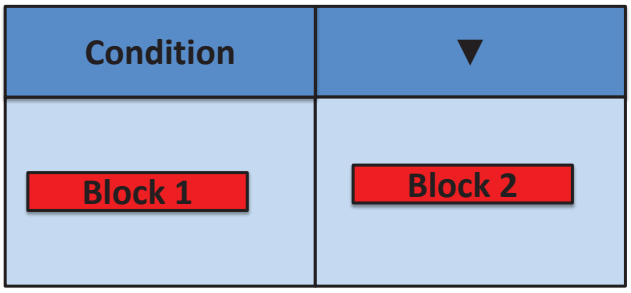

Figure 2: Block of simple alternative "if..then..else".

right of the arrow.

2.3 Compound Blocks

Embedded blocks in the body of a compound block are ordered from the top downward, which illustrates their sequential execution (see Fig. 1).

In compound blocks, the program "grows" toward the inside, thereby naturally limiting the size of defined subprograms. This urges the developer to design simple subprograms.

If any developer feels limited by this characteristic, an analogue of a connector in flowcharts may be used. An empty block can be used as such a connector, with its name put inside. The contents of such a block may be expanded on in another picture. However, in interactive electronic charts, the structure of such an empty block is expanded automatically upon clicking the empty block.

Attaching a qualification from the left to the block and inserting an identifier of any structured variable in the qualification, access operation to appropriate components of the variable can be expressed. These components then become immediately accessible from within the body of the block and all embedded blocks with no need of explicit qualification (analogue of the structure with in Algol, Pascal, Visual Basic, …).

Conditions that affect the run of the program are entered in the header, footer and dividing bars.

Compound conditional block of the type if … then … else can be expressed using a block with one header and two bodies where the condition in the header of the right body will always be true – see Fig. 2. Analogically, a general conditional statement can be represented using a block with several bodies – see Fig. 3.

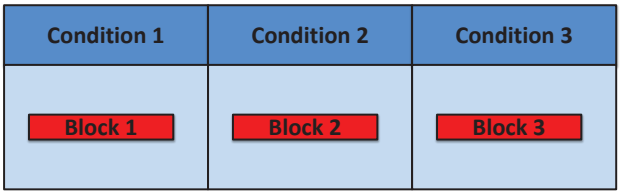

Figure 3: Block of general conditional statement.

Switch or case is a special case of the general conditional statement. Reduced notation according to Fig. 4 is established to eliminate the necessity of writing all condi-

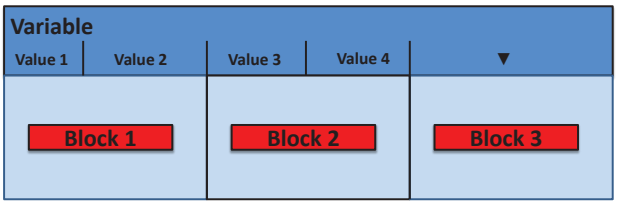

Figure 4: Block of switch statement.

tions in the header as "variable=value".

In some cases, all parts of a block with several bodies may not fit next to each other. If this is the case, individual parts of the block may be placed below each other and connected – see Fig. 5.

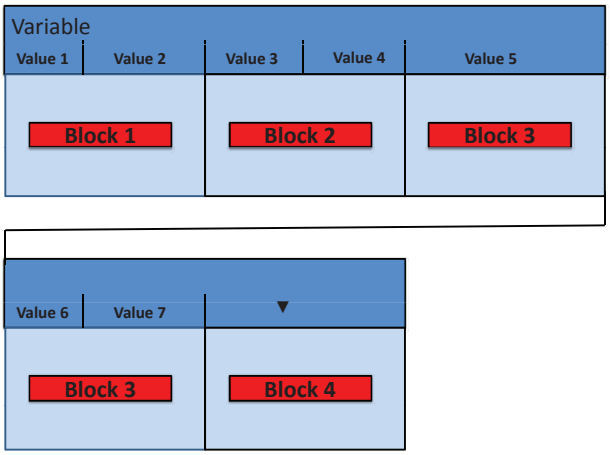

Figure 5: Order individual parts of the block when lack of space.

2.4 Evaluating Conditions

Headers, footers and dividing bars contain conditions that affect further evaluation procedure of the algorithm. The following general rule applies to evaluation of conditions: If the condition is true, the program continues downward; if false, the program continues to the right.

The header contains a condition that determines whether the body of the block will be entered. If true, the program continues downward to the body; if false, the program continues to the right by evaluating the next condition, and in the last condition on the right, by leaving the block and continuing with the subsequent block.

If the body of the block is divided using a dividing bar (see Fig. 6), its condition is evaluated identically. If true,

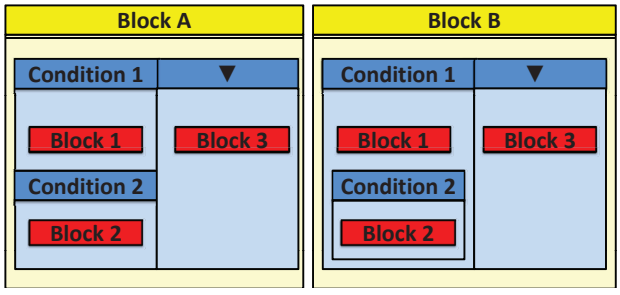

Figure 6: Dividing bar in the block. Algorithms written in blocks A and B are equivalent.

the program continues downward and executes the body of the block; if false, the program continues to the right and if it is the last condition, the block is left.

Blocks representing a cycle contain, besides a header, also a footer. In order to evaluate a condition in the footer, the cycle block can be understood as wound up on a roller. When the footer condition is true, the program does continue downward; however, continuing downward on a roller means evaluating the header again. If the condition is false, the program continues to the right, and if the last condition on the right is false, the cycle block is left (see Fig. 7).

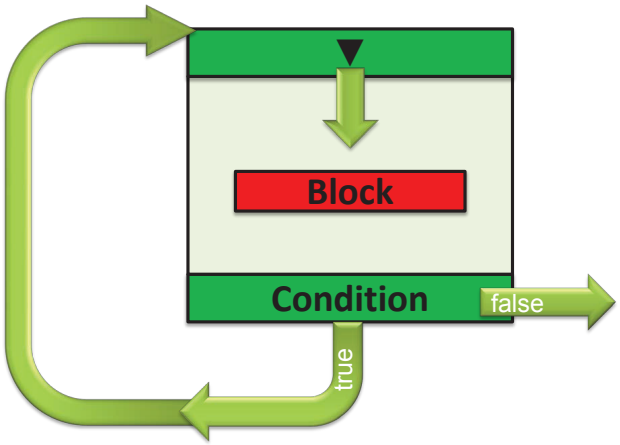

Figure 7: Evaluation flow in the cycle with condition in the footer.

Any condition that is always true should be marked using an arrow oriented downward. However, in the footer such a condition may also be marked using an upward arrow to make the arrangement clearer, given that it returns the program to the beginning of the cycle, thus to the point of header evaluation (see Fig. 8).

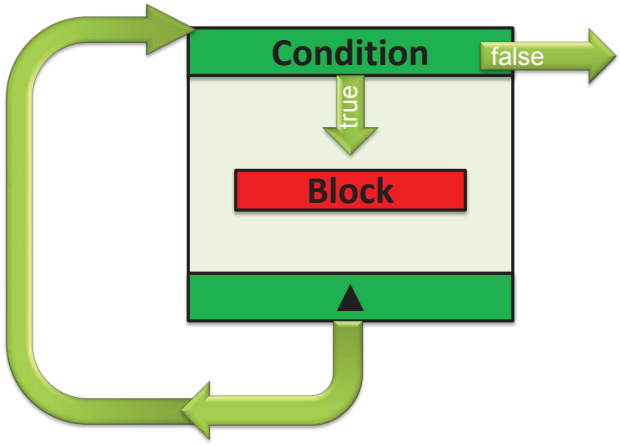

Figure 8: Evaluation flow in the cycle with condition in the header.

This simple way can be used to clearly express how various types of cycles are executed – see Fig. 9.

By dividing the block in several bodies, cycles with several bodies can be easily expressed, which are known from some programming languages. (see Fig. 10)

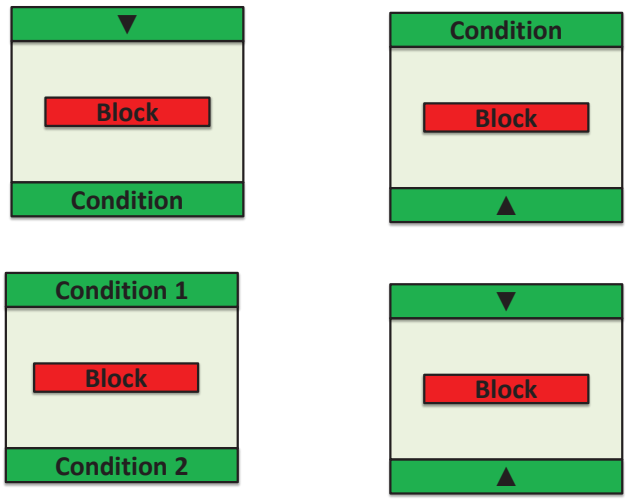

Figure 9: Basic types of cycles.

Input conditions in the header are evaluated identically as in a single-body cycle, i.e. the header is always evaluated from the upper left corner, irrespective of which of the bodies was gone through before. However, output conditions in the footer are evaluated only in the part adjacent to the given body (see Fig. 10). If several bodies share one output condition, such a condition can be represented as their shared footer – see Fig. 11.

In addition, upon adding embedded headers to a multiple-body block, a powerful algorithmic structure is achieved, which makes it possible to arrive at a simple and well-arranged representation of a number of algorithms structured only with difficulty before – see Fig. 21.

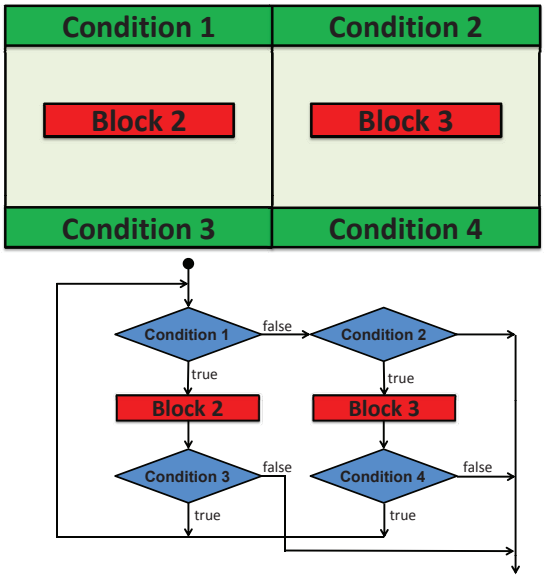

Figure 10: Cycle with several bodies.

2.5 Compound Conditions

Conditions expressed in the header, footer and dividing bar can also exhibit complex structures. A structured condition can be represented as a table whose cells include the tested conditions (see Figs. 12 – 14). These conditions are evaluated based on the following rules:

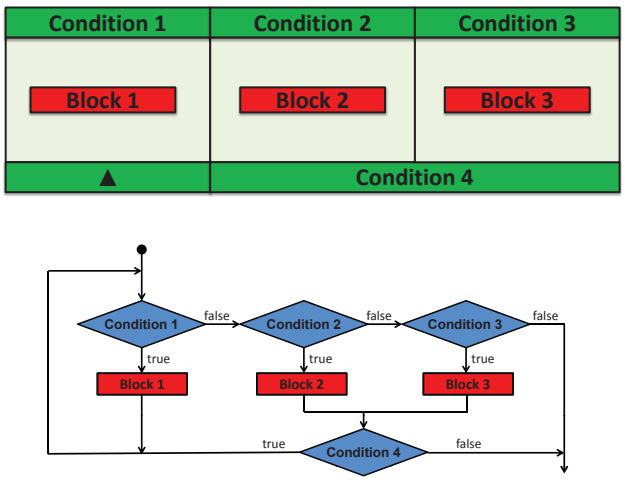

Figure 11: Cycle with several bodies with shared output footer

- Evaluation is started from the upper left corner.
- If the condition is true, the program continues downward.
- If the condition is false, the program continues to the right.
- An empty condition (empty rectangle) is run in the direction in which it was entered (if entered from the left, the rectangle will be left to the left; if entered from above, the rectangle will be left in the downward direction).
- A condition always true is represented using a downward arrow; in the footer, it may also be represented using an upward arrow to enhance the clarity of visual arrangement.
- A condition always false is represented using an arrow to the right.
- If any header or footer is left to the right due to evaluation of the conditions, it means that the whole block is left, as well, and program control is passed on to the next part of the program.
- If a header is left in the downward direction, the body is executed. If a footer is left in the downward direction, evaluation of the header will be the next step (see the analogy with a roller cycle described above).
- If the cycle block is divided in several parts, this division logically continues also in the footer, and evaluation of the footer starts from the upper left corner below the appropriate body. The program returns to the start of the block if the footer is left in the downward direction. The cycle is left if the footer is left to the right or if a bar is encountered while evaluating the conditions, which divides the cycle to individual parts corresponding to different bodies – see Figures 10 and 20.
- Evaluation of the footer in cycles with several bodies is started in the upper left corner again.

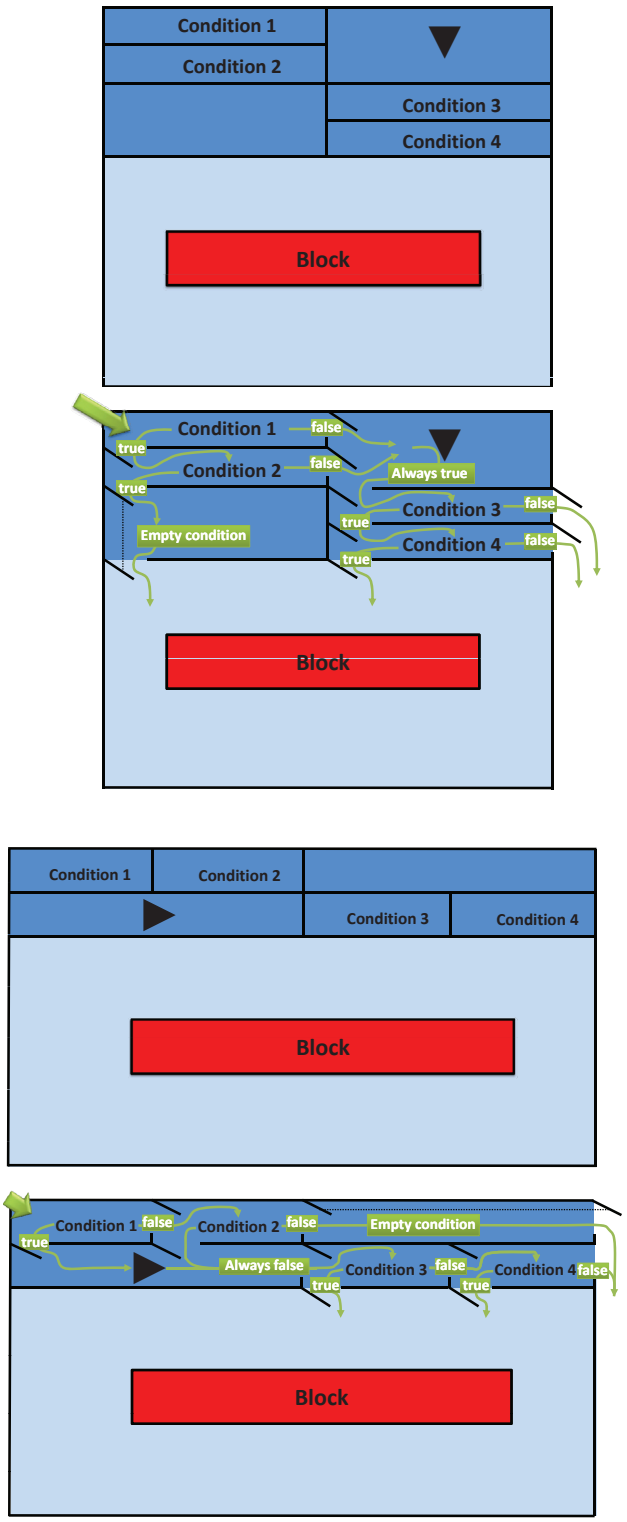

Figure 12: Evaluation flow in structured conditional blocks. Evaluation is started from the upper left corner of the blocks. If the condition is true, the program continues downward; if false, the program continues to the right. A condition always true is represented using a downward arrow; a condition always false is represented using an arrow to the right. An empty condition (empty rectangle) is run in the direction in which it was entered - if empty rectangle entered from the left, the rectangle will be left to the left; if entered from above, the rectangle will be left in the downward direction.

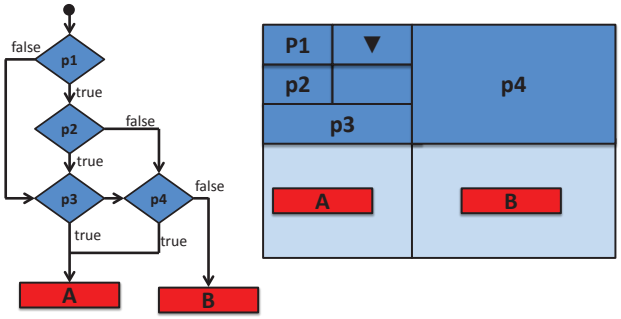

Figure 13: Intricately structured condition in header.

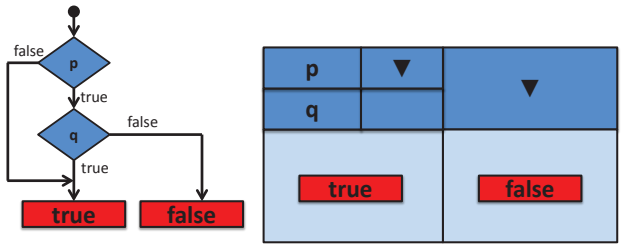

Figure 14: The algorithm evaluation of implication p → q using compound structured conditions.

The cycle is left only if, upon evaluating the conditions, the footer (and thus the whole cycle) is left to the right. If the program falls through the footer in the downward direction, the cycle is not left, and thus the cycle keeps running and the program continues by evaluating the conditions in the header (see Figures 10 and 21).

2.6 Non-Structured Statements

In theory, a structured algorithm is defined as one that observes the following rules:

- Algorithm is formed by a linear sequence of blocks where each has only one input and one output.
- Where a decision is to be made in the program, an equivalent of the block *if ... then ... else* is used.
- Where a part of the code is to be repeated, one of the cycles while or *repeat ... until* is used.

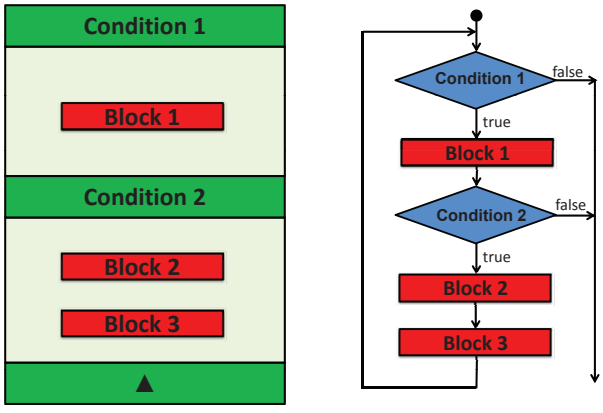

Figure 15: Dividing bar in the cycle.

While it can be proven that every algorithm can be written observing the rules above, in some cases its structure may be easier to understand if some of the rules is violated. Therefore the majority of programming languages offer syntactic structures for such operations.

That is why kopenograms allow for representing typical statements that violate the purity of its structure for the sake of its clear arrangement.

2.6.1 Break

Premature leaving of a block can be represented using a dividing bar (Fig. 6 and 15). However, if multiple blocks are to be left at the same time, the graphical representation in Figs. 16 and 17 can be used.

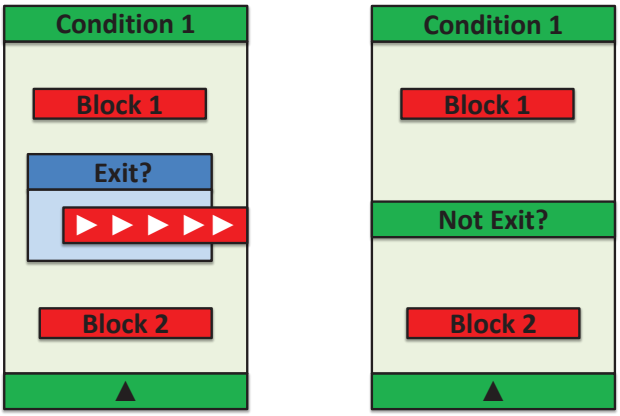

Figure 16: Early exit cycle (using the "break"). However, the use of dividing bar and inversion condition is clearer.

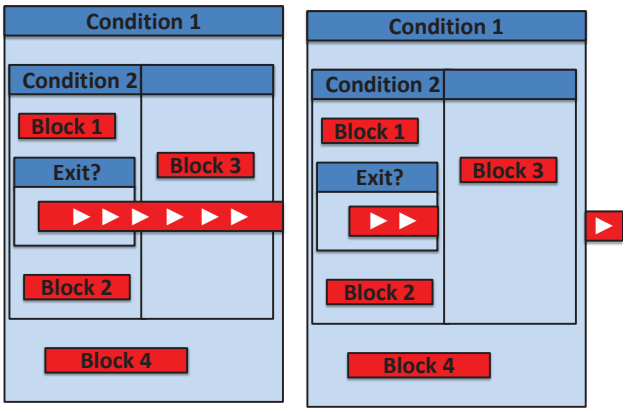

Figure 17: Two equivalent forms of graphical representation of leaving inner block.

2.6.2 Continue

Continue is shown as a classic statement whose name is represented by an upward arrow (Fig. 18). The program then continues by evaluating the header of the innermost cycle.

2.6.2 Return

The statement of premature termination of a subprogram can be represented similarly as break, and/or as a classic statement named return.

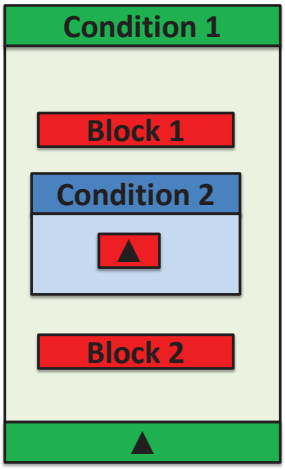

Figure 18: The "Continue" statement in the cycle.

2.6.2 Exceptions

In order to represent a block with an expected exception and also a block that is used to handle such a raised exception, only color is used (see Fig. 19):

- The header of the block with an expected exception and also of the subsequent block to handle the exception is filled using white color.
- The body of the block with an expected exception is filled using violet color.
- The body of the block where such a raised exception is handled is filled using orange color, similarly as any body of a block executed irrespective of whether an exception was or was not raised (the block finally).

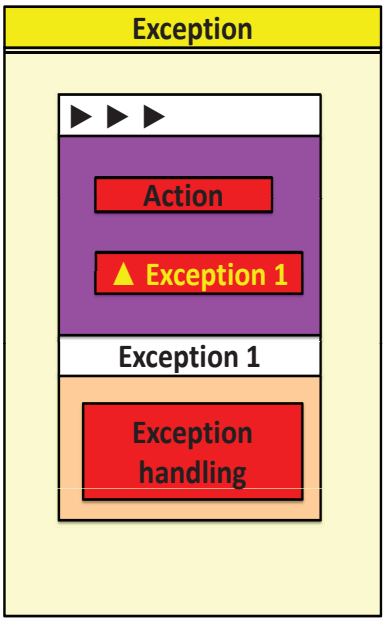

Figure 19: Throwing an handling the exceptions.

2.6.2 Goto

The general statement goto represents considerable violation of the structure, and therefore it is also strongly

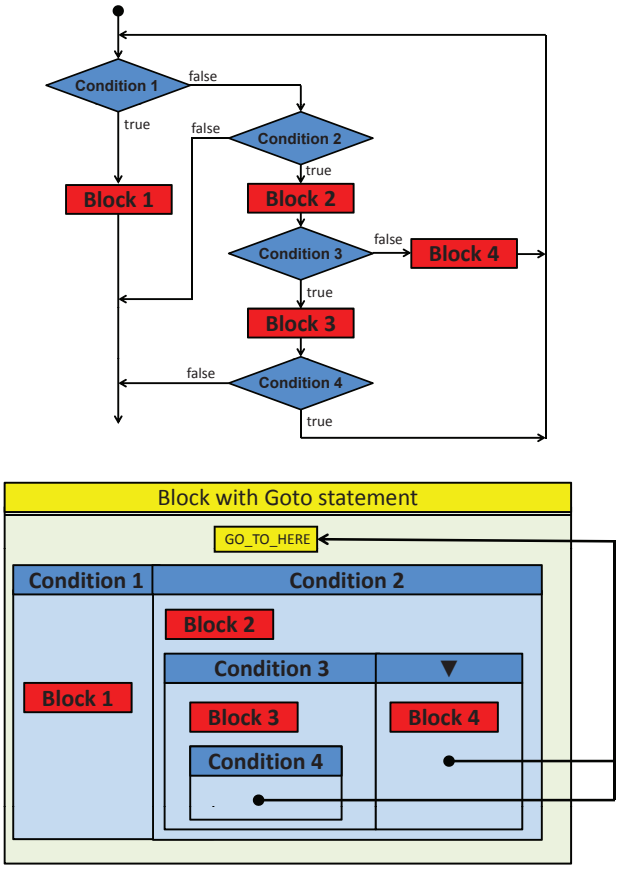

Figure 20: Goto statement in kopenogram.

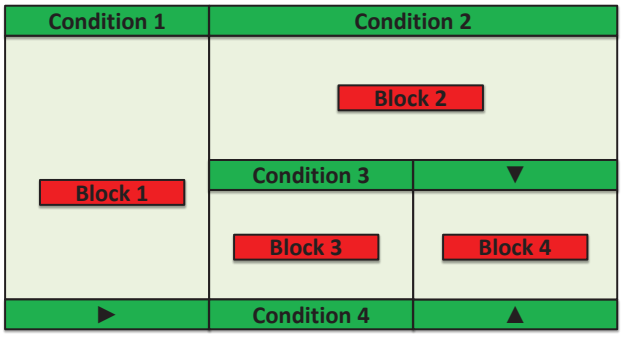

Figure 21: The general algorithmic block with embedded headers and multiple bodies. This block performs the same algorithm as a block in Figure 20.

highlighted. This statement is shown as an arrow oriented from the point of the jump to the right, outside of the block, and then up or down and finally to the left of the block with the target label (Fig. 20).

3 Conclusion

Kopenograms are a handy tool for clear graphical representation of the structure of algorithms, and they have found long-term application particularly in teaching programming classes.

These are a convenient supplement of UML diagrams used to represent algorithmic structures.

Kopenogram specification is published on [16].

4 Acknowledgement

This paper describes the outcome of research that has been accomplished as part of research program funded by GrantAgency of the Czech Republic Grant No. GACR P403-10-0092 and by the grant FR—TI3/869.

5 References

[1] ISO (1985). Information processing -- Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts. International Organization for Standardization. ISO 5807:1985.

[2] Jackson, M. A.: Principles of Program design. Academic Press, 197

[3] (Jackson, M. A.: System Development. Prentince Hall, 1983

[4] Nassi, I., Schneiderman, B.: Flowchart techniques for structured programming, ACM SIGPLAN Notices, Vol. 8 Issue 8, August 1973. ACM, ISSN: 0362 1340

[5] Kofránek, J.; Novák, P.: Kopenograms - graphical method of representation of programs (in Czech) Kopenogramy – způsob grafické reprezentace programů. In Moderní programování – Vinné 1987, Dům techniky ČSVTS, díl 4, Žilina, str. 11–161. 1987.

[6] Pattis, R. E., Karel The Robot: A Gentle Introduction to the Art of Programming.John Wiley & Sons, 1981. ISBN 0471597252

[7] PC-Karel : interpret of KAREL language for PC [online]. c2004 [cit. 2012-04-12]. Available from: <http://pckarel.sweb.cz/pckarel.html>.

[8] Pecinovský, R.: Fundamentals of Algorithmics I (in Czech) 602 ZO Svazarmu, 1985.

[9] Pecinovský, R.: Fundamentals of Algorithmics II (in Czech) 602 ZO Svazarmu, 1985.

[10] Pecinovský, R., Kofránek, J.: Simple data types (in Czech) 602 ZO Svazarmu, 1985.

[11] Pecinovský, R., Kofránek, J..: Structured data types (in Czech) 602 ZO Svazarmu, 1986.

[12] Pecinovský, R., Kofránek, J..: Searching and sorting (in Czech), 602 ZO Svazarmu, 1986.

[13] Pecinovský, R., Kofránek, J.: Dynamic data structures (in Czech), 602 ZO Svazarmu, 1986.

[14] Pecinovský, R., Kofránek, J.: Modular programming (in Czech) Modulární programování, 602 ZO Svazarmu, 1987.

[15] Pecinovský R., Ryant I.: Programming of of parallel processes (in Czech) 602 ZO Svazarmu, 1987.

[16] www.kopenogram.org

Email to corresponding author: kofranek@gmail.com