ijecs open access www.ijecs.in
International Journal Of Engineering And Computer Science
Volume 13 Issue 06 June 2024, Page No. 26219-26228
ISSN: 2319-7242 DOI: 10.18535/ijecs/v13i06.4847

Ball-Maze Block Diagram for Visualizing Logic Flow and Standardizing Code Structure

Eik Fun Khor

Institute of Technical Education
College West
1 Choa Chu Kang Grove
Singapore 688236

Abstract
Both flowchart and pseudocode are popular tools to describe program algorithms. However, they have their strengths as well as weaknesses. Flowchart visualizes logic flow effectively but sometimes it may not be conveniently converted to program code for novice programmers. On the other hand, pseudocode format is closer to program code but may not be as easily visualized as flow chart. To address the issue, this paper proposes an alternative graphical representation of logic flow, called ball-maze block diagram, which takes advantages of both flowchart and pseudocode in a single representation format. Besides describing the graphical representation of the diagram, this paper also demonstrates systematic conversion from the diagram to standard code structure to ease code development. The generated code is structure-friendly regardless of the complexity of logic flow. Step-by-step guides are given on practical problem to demonstrate the usefulness of the proposed methods.

Keywords: graphical representation, block diagram, programming, code development, computer science.

1. Introduction

In today's programming, there exist various diagramming tools to graphically describe computer programs. Unlike the representations used in other design disciplines, new diagrams for programming are continually being developed, so it is possible to observe the effect of successive generations of diagram conventions, or even influence the development of future generations of design notation [1]. Some diagram types to model program solutions can be found in Unified Modelling Language [2]. In addition, some other examples are introduced separately, such as Sketchpad [3,4], LabVIEW [5], Scratch [6] and so forth. In general, the diagramming tools can be classified into 2 main categories – Data Flow Diagrams (DFDs) and Control Flow Diagrams (CFDs) [7-12]. Both of them serve different objectives in programming applications. DFDs' origins can be traced back at the Ph.D. thesis of Sutherland [13] where a light-pen and a TX-2 computer are used to create a visual programming language, on top of the SKETCHPAD framework. While data flow diagrams represent the flow of data, operands or information within a program, a control flow diagram shows the logical flow of operations, i.e. what operations to be performed, in what order, and under what circumstances. While programming with DFDs is found more user-friendly for programmers, it normally requires sophisticated conversion software tool working in the background to translate the diagram to source code or directly to machine code for targeted platform specified by developer. In this paper, we focus on the graphical representation of control flow in a program, i.e. CFDs, where programmers can learn to develop their own source code in standard structure from the diagram in a simple and systematic way without the reliance on additional software tool.

Flowchart is schematic representation of a process [14]. As stated in New World Encyclopaedia [15], it was introduced by Frank Gilbreth to members of American Society of Mechanical Engineers (ASME) in 1921 as the presentation "Process Charts—First

Steps in Finding the One Best Way." It is also known as control flow diagram. Although it is the earliest diagramming tool, it is still remained as well-known diagramming tool. In today's computer science education, it is still found useful and widely applied, especially by beginners, to plan logical flow of a program before learning to develop source code for the program. It is capable of showing the overall logic flow of instructions from one process to another, including the branching and looping in the logic flow. Nevertheless, sometimes the novice programmers have difficulty to translate some logic flows to source code, especially when the original flow is unstructured and/or involve nested branching and looping in different combinations.

Pseudocode is another type of representation of a program algorithm. It uses a combination of natural language and programming language in written format. Since it allows users to formulate their thoughts into computer algorithm without the need to follow exact coding syntax, it serves as intermittent step towards the development of the actual code. As it is very much similar to program code, it is found easier than flowchart for user to write source code. Also, compared to flowchart, it requires less space to develop as it can be in written text format in our own way as there are no fixed rules. However, the logic flow of instructions in pseudocode cannot be easily visualized by users, especially novice programmers. Besides, pseudo code is less appropriate than flowchart to explain the flow to people with no programming background.

Knowing that flowchart and pseudocode have their own strengths and weaknesses as compared to each other, this paper introduce a new diagramming tool to combine the advantages of the two. The proposed tool, is call Ball-Maze (BM) block diagram. It is a versatile graphical representation of program so that user can visualize the logic flow of the program effectively and at the same time, facilitate users to write source code in more direct and simple way. The main purpose is to assist users who have difficulty to develop source code that involve complex control flow structure, such as the nested branching and looping and their combinations in various ways. It will be shown that with the proposed BM block diagram, the generated code structure is standard and systematic. The method does not require conversion tool from diagram to source code. In addition, it is suitable for users who want to have direct control on the structure of code to be developed.

This paper is organised as follow. Section 2 introduces the basic concept of BM block diagram. Section 3 defines the basic building blocks for the diagram while Section 4 covers the flow representation for the diagram. Section 5 suggests, although not compulsory to the users, a systematic coding method to facilitate users to develop source code effectively from BM block diagram. The full implementation process from developing BM block diagram to generating source code is explained step-by-step in an application example in section 6 before conclusion is drawn.

2. Basic Concept of Ball-Maze (Bm) Block Diagram

In BM block diagram, program execution is modelled by a ball rolling along pathways in a maze. The maze is constructed by a number of maze blocks and the pathways that interconnect the maze blocks. The ball rolls from one maze block to another through the pathways. The pathway is unidirectional. There are 2 types of junctions on the pathways, 1) branch - junction where one pathway split into multiple pathways (single-in-multiple-out) that can be controlled , and 2) join - junction where multiple pathways are combined into single pathway (multiple-in-single-out).

While the ball rolls inside a maze block, operations where the ball passes by are evoked. The ball can roll into a maze block at one end and exit at the other. Sometimes there are multiple exits out from a block. After exiting the block, the ball will follow the pathway that link from one block to another to enter the following block, which can be either another maze block or the same one where it was exited from. By re-arranging the maze blocks and re-connecting the blocks to one another, the overall layout of the maze can be altered accordingly.

Fig. 1 illustrates an example of a maze which consists of 4 maze blocks, i.e. blocks 1 to 4. The ball begins at "start" position of the maze. It first enters block 1 through the entrance "Entry 1". When it comes to "branch A", it can be controlled which exit to take, i.e. $Exit_{1,1}$ or $Exit_{1,2}$. If the ball take $Exit_{1,1}$, it will follow a pathway that lead to "Block 2", and subsequently to "Block 3" and "Block 4". On the other hand, if it take $Exit_{1,2}$, it will bypass "Block 2" and "Block 3" and reach Block 4 directly. In either way, it will finally enter "Block 4" and roll through "Entry 4" before exiting from the maze.

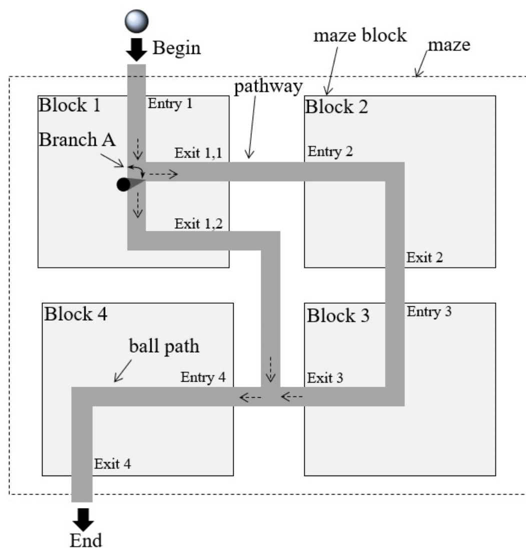


Fig. 1 Rolling ball and maze blocks

Since the diagram is modelled after the ball maze, the program structure in the diagram is analogous to the layout of the maze block and the interconnections among the blocks while program execution is analogous to the process of rolling ball in the maze.

3. Basic Building Block

After showing the overall ball maze concept, this section focus on the detailed structure of the building block of the maze and its simplified model, the main graphical symbol of BM block diagram.

First, let's take a look on the basic mechanism of a maze block as shown in **Fig. 2**. In general, it consists of 3 primary parts:

i) Entry to the block,
ii) Jobs, i.e. jobs P_1 to P_3, to be executed in pre-defined sequence from top to bottom, and
iii)Exits from block, i.e. $Exit_1$ & $Exit_2$.

The maze block is "activated" when the ball enters the block, which is part 1) of the block. When it rolls through the jobs in part 2), the jobs are executed. Each job is executed one at a time and in sequence where it passes by. In this case, the order of the jobs to be executed is P_1, P_2 and followed by P_3. The ball will stay at the current job where it is being executed and it only moves on to the next job when the current job is completed. When the ball comes to exit part (part 3) in this example, there are two possible exits. In the mechanism shown, which exit to take depends on the position of the control horn at the junction. If the horn is at position A, the ball will exit from the block through Exit 1. Otherwise (horn at position B), Exit 2 will be taken. The horn position is governed by the selection criterion C. If the criterion C is TRUE, control horn will be switched to position A, else, at position B. The subsequent block to be activated depends on which block the ball rolls to and this depends on the layout of the pathway that inter-connects the blocks together.

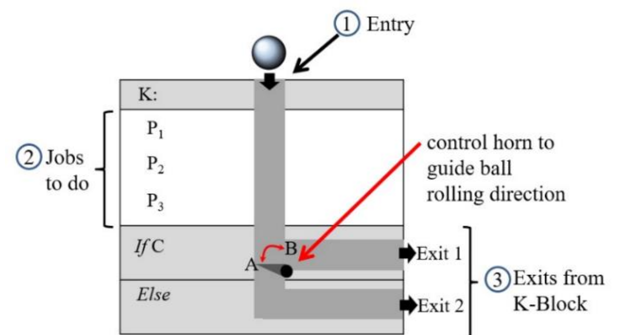


Fig. 2 Rolling ball model of maze block

Fig. 3 depicts the simplified representation of a maze block for use in the diagram. It applies the same working principle of maze block explained in **Fig. 2**.

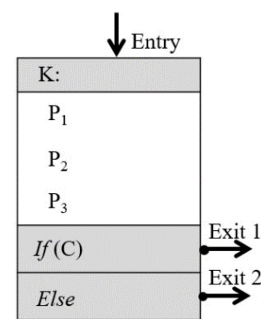


Fig. 3 Simplified model of maze block

Note that the block symbol is simplified from conceptual maze blocks in the ways that, (i) the layout of pathways inside the block is removed with the same working principle remained and, (ii) the pathways from one block to another are replaced by flow lines for ease of drawing.

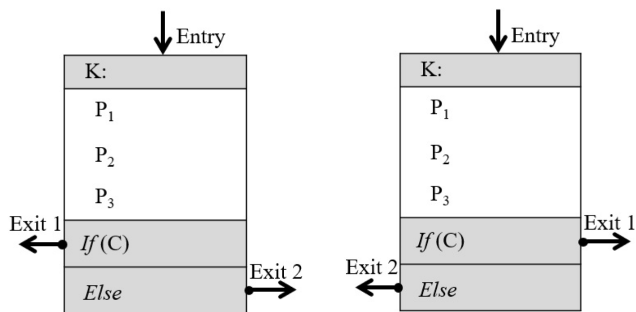
As similar to maze block, the block symbol consist of 3 primary elements as follows:
i) Entry to the block
ii) Job(s) to be executed
iii) Exit(s) from the block

The number of entries to a maze block is one for sequential flow (not concurrent flow). The jobs to be done follow the sequence from top to bottom. The number of job (J) to perform can vary from zero to any integer number. The number of exit(s) (E) from a maze block can vary from one to any integer

number as well. For multiple exits, which exit to take depends on the exit criteria (C). In more detailed definition, the program will take the first exit where its exit criterion is TRUE.

From programming point of views, the control structures that a maze block cover are:
i) Sequential Control – E.g. P_1, P_2 and P_3. No defined sequence if the jobs are within the same job row.
ii) Branching Control – E.g. If (C), take exit E_1, Else take exit E_2.

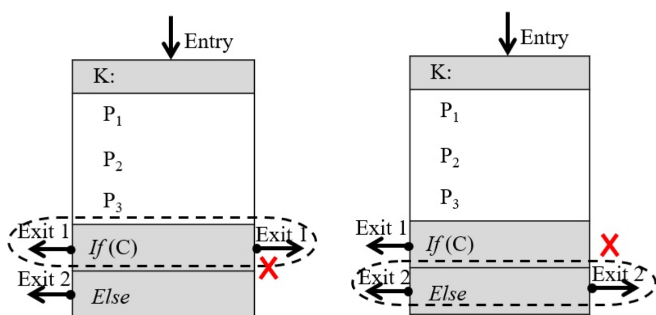
Pertaining to the graphical representation, the exit from a maze block can be on the right or left side as below in **Fig. 4**. However, in the context of sequential programming, only one exit is allowed per exit branch as shown in **Fig. 5**.



(a) Exit 1 on the left and Exit 2 on the right

(b) Exit 1 on the right and Exit 2 on the left

Fig. 4 Maze block exit can be on either side (left or right)



(a) 2 exits for Exit 1

(b) 2 exits for Exit 2

Fig. 5 Maze block exit branch on both sides is not allowed

Note that a maze block is simply represented by a big rectangle with rows of smaller rectangles stacking on one another. In contrary to flowchart, BM block diagram is shape friendly as all maze blocks has standard format and in rectangle shapes. This ease the process in choosing the correct shapes to use and as well as constructing the graphical representation for the blocks.

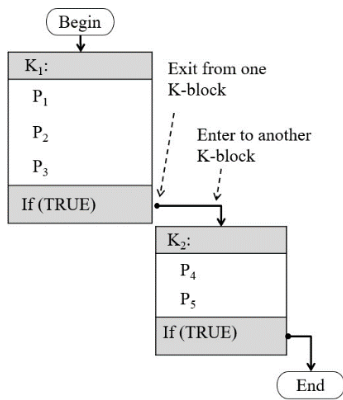
4. CONSTRUCTION of BALL-MAZE (BM) BLOCK DIAGRAM

Upon introducing maze block as standard building block, this section explains the construction of the proposed diagram, BM block diagram. In general, BM block diagram for a flow can be represented by a number of maze blocks connected through flow lines. Similar to flowchart, the flow begins with the "start" terminal and end with "end" terminal. In case of endless repetition, the "end" terminal is not needed. Unlike flowchart, there is only one type of execution node, which is maze block.

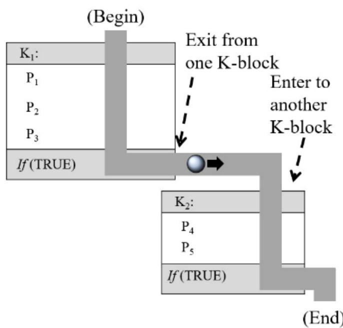
In other words, BM block diagram consists of the below elements to represent the control flow:
i) Maze blocks that encapsulate a series of jobs to be performed and exit branches.
ii) Flow lines to represent the flow from one block to another.
iii) Terminals such as "Start" and "End" to mark the start and end of the flow.

Through proper design of logic flow from one maze block to the other, various types of control structure can be implemented. These include:
i) Sequential control where a series of process carried out from one maze block to another.
ii) Branching control to logically control the next maze block to be activated (where the ball rolls to) through assessment of binary decision (Exit criteria).
iii) Looping control to allow the conditional repetition of a block or sequence of blocks. The loop control is implemented through the combination of Exit branch and the flow lines linking to the previous block.

Fig. 6(a) depicts an example of sequential control from one maze block to another. The program starts from K_1 to perform jobs P_1 to P_3. After K_1, it subsequently activates K_2, in sequence through the flow line connecting from K_1 to K_2. Thus, it can be seen that the example given is purely a sequential control from K_1 to K_2. Its equivalent representation in maze blocks is illustrated in **Fig. 6(b)** for ease of visualization on the control flow.



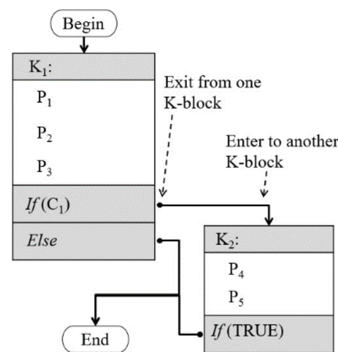
(a) Sequential control example with simplified model



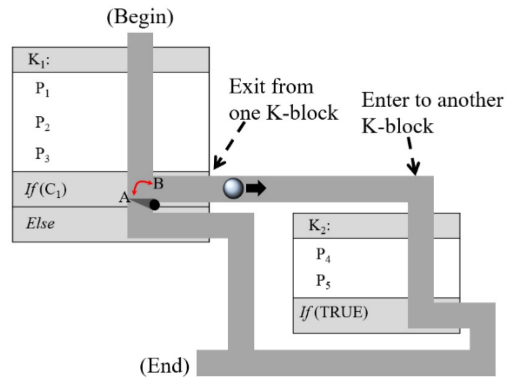
(b) Sequential control example with rolling ball model

Fig. 6 Sequential control example

Fig. 7(a) depicts an example that comprises a simple branching control to 2 different maze blocks. After activating block K_1, the program can either take the first exit to activate K_2 or through second to end the program. The decision is governed by criterion C_1. Its equivalent control flow in rolling ball model is shown in **Fig. 7(b)**. As far as there are multiple exits from a maze block, branching control is involved.



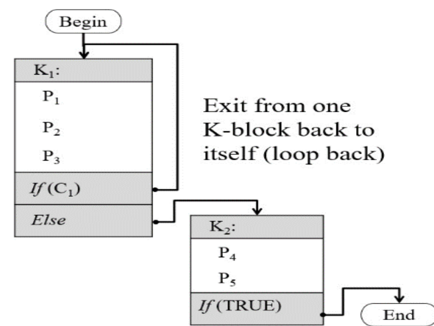
(a) Branching control example with simplified model



(b) Branching control example with rolling ball model

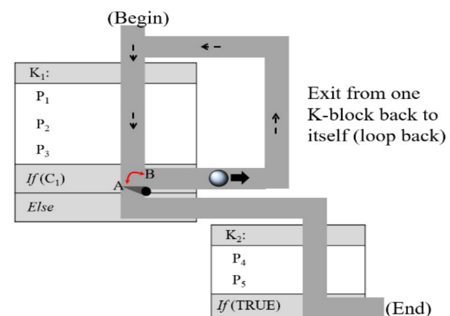
Fig. 7 Branching control example

Fig. 8(a) illustrates a simple example of looping control where the same block K_1 will be repeatedly activated as far as criterion C1 is TRUE. Its equivalent representation in rolling ball model is illustrated in **Fig. 8(b)**.



(a) Looping control example with simplified model

In the case where a BM block diagram is long winded that flow lines start crisscrossing, which may cause confusion, or to continue on separate page, connector symbols is used to connect two or more different part of flowlines in the diagram. As shown in **Fig. 9**, they are represented by a circle and a letter or digit is placed within the circle to indicate the link, which is similar to flowchart.



(a) Looping control example with rolling ball model

Fig. 8 Looping control example

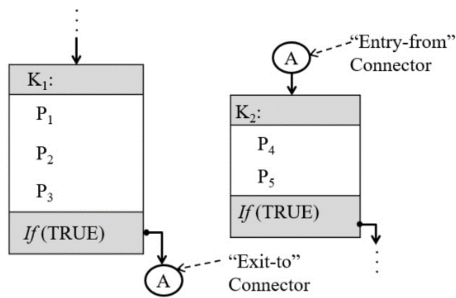


Fig. 9 Connectors that connect one block to another block

There may also be cases where a maze block itself is too long to fit into a desired space. To address the issue, another type of connectors as shown in **Fig. 10**, called "Continue-to" and "Continue-from", are introduced to join different parts of a maze block together. In the figure, both parts are combined by the connectors to form a complete K_1 block.

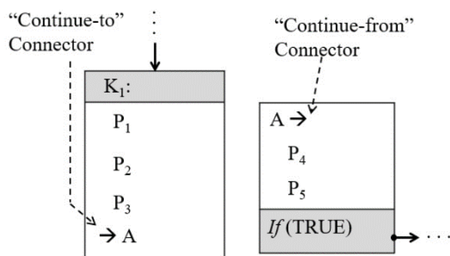


Fig. 10 Connectors that put together parts within the same block

5. Code Development for BM Block Diagram

Another objective of using BM block diagram is to ease the code development in a way systematic way and within a standard code structure. I can simplify the process of developing code, especially for novice programmers who have difficulty to develop code for complex logic flow. Nevertheless, users can choose to use their own code development method or the proposed method, called K-coding [16], in this section.

In K-Coding approach, source code is constructed at 2 levels: i) for each maze block and ii) for overall program. At block level, the code content is constructed separately for each maze block within simple branch statements. There are basically 2 set of branches for each maze block: 1) check-in branch for the incoming branch and, 2) check-out branch for outgoing branch as shown in **Fig. 9**.

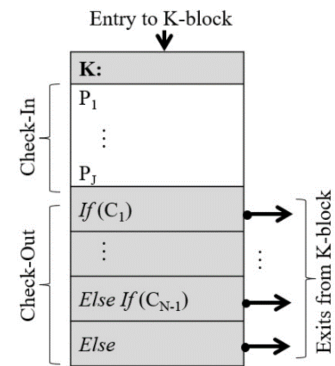


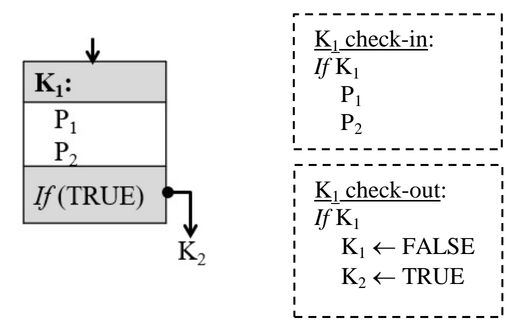
Fig. 9. Check-in and check-out branches in maze block

Check-in branch basically check the condition to enter the maze block. If the condition is TRUE, the maze block become active (or TRUE) and the process, $P_1,\ldots,P_J$ in the maze block are executed in order. On the other hand, check-out branch checks whether the program will exit from the current maze block. If the condition is TRUE to exit, it describes the next active block. In general, there are three possible outcomes in check-out branch. The program will either: 1) exit to different maze block, 2) exit to itself (loop back) or, 3) exit to end of program (End is TRUE).

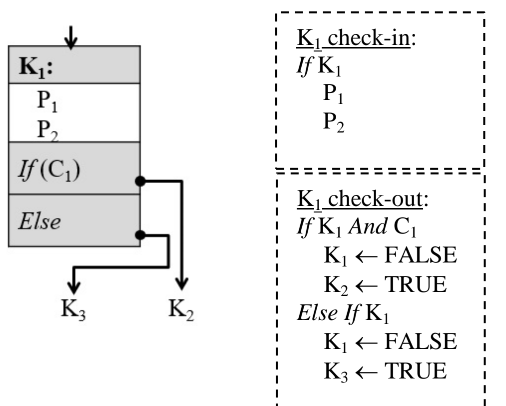
In the code development, check-in and check-out branches can be easily implemented using various types of branch statements, such as IF-THEN, IF-THEN-ELSE, CASE-SELECT, etc. For consistency, we choose to use IF-THEN or IF-THEN-ELSE statements to demonstrate the sample code for maze blocks. **Fig. 10** illustrates the pseudo code for different types of exit scenarios. For simplicity, Boolean variables are used to mark the active (TRUE) and inactive (FALSE) of each maze block. Maze block can change the state during the program runtime. In practice, users can use any variable or array type to mark the states which they think is appropriate. For the case in **Fig. 10(c)** where there is no EXIT for C_1 = FALSE as there is no change in the active state of maze block, K_1 still remains active.

There are basically 2 options in implementing maze block: either 1) split check-in and check-out branches into 2 separate sets of branch statements or, 2) combine check-in and check-out branches in single set of branch statement. The code examples shown in **Fig. 10** take the first option. **Fig. 11** illustrates the code templates, taking **Fig 10b** as an example. Both options are shown in the templates. In comparisons, option 1 implementation results in

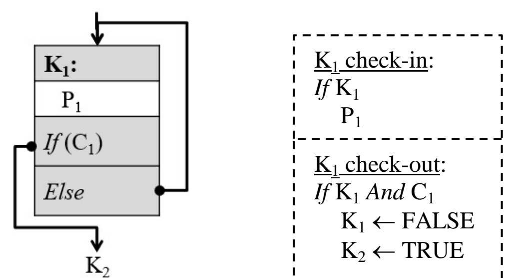
simpler code structure for novice programmer since nested branch structure can be avoided. On the other hand, option 2 implementation improves code efficiency as there is no redundancy in checking the condition for K_1. Finally, it is users' preference on which options to select.



(a) Maze block with single exit



(b) Maze block with multiple exits



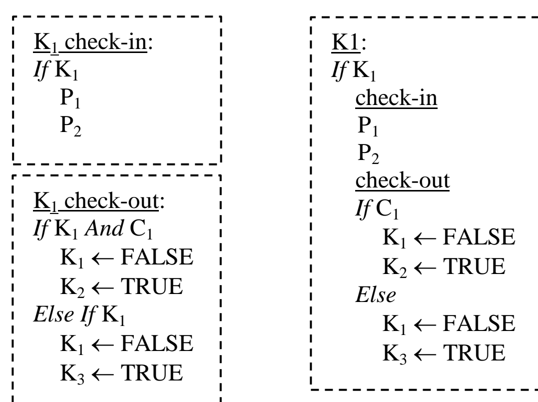
(c) Maze block with loop-back

Fig. 10. Code development for maze block

It is essential to highlight that while the standard model within the maze block, as shown in **Fig. 9** can be unstructured flow, during the K-coding for the maze block, the composed source code, as shown in **Fig. 11**, become structured.

After constructing code content for each maze block at level 1, users may proceed to assemble the code blocks together to construct the overall program in a standard logic structure as shown in **Fig. 12**. The notations <K_i Check-In> and <K_i check-out> represent the code blocks of check-in and check-out branches respectively, for block K_i.

The overall code structure of the program is standard and simple to implement regardless of the complexity of original logic flow. In general, it consists of a series of branch statements within a main loop and the program will repeat the main loop until coming to the end of the program (End is TRUE). Although sometimes K-coding may result in longer code size and/or decrease in code efficiency in terms of speed, the resulted code structure is significantly simplified and direct to implement. Thus, basic logic flow knowledge in programming skills is sufficient to solve programming task for complex logic flow.

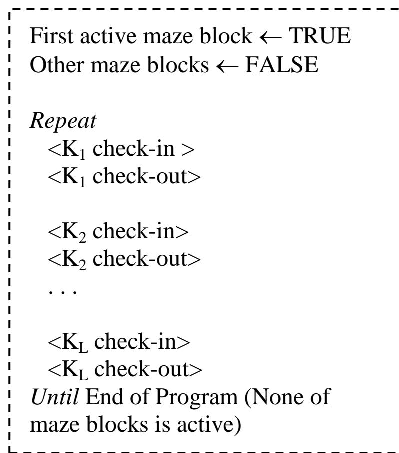


(a) Code template 1 - splitting check-in and check-out

(b) Code template 2 - combining check-in and check-out

Fig. 11. Different code implementations of maze block

The proposed process in developing source code as mentioned above is called K-composition. It can be easily noted that the resulted code is a structured program. The statement stays valid regardless of whether the original logic flow is structured or unstructured. Besides, the method is less error-prone as the source code for the content of the maze block and logic flow among the blocks can be traced separately against BM block diagram.


First active maze block ← TRUE
Other maze blocks ← FALSE

Repeat
<K_1 check-in >
<K_1 check-out>

<K_2 check-in>
<K_2 check-out>
. . .

<K_L check-in>
<K_L check-out>
Until End of Program (None of maze blocks is active)

Fig. 12. Constructing overall code with maze blocks

6. Application Example

In this section, the step-by-step implementation of the proposed method is demonstrated in practical application of on a traffic light control. **Fig. 13** depicts the layout of T-junction traffic lights. There are three sets of traffic lights, one set for each road, namely East (E) road, West (W) road, and North (N) road. As usual, each set of traffic contains 3 different light colours, i.e. red (R), amber (A) and green (G). For W road, Left-Turn (LT) light is included to direct vehicles coming from W road to do left turn into N road. In the system, each light is labelled by the road name followed by the light colour. For example, "N Red" represents the red light on N road. Instead of fixed traffic patterns, the traffic will response differently depending on the traffic condition at the respective road. In the system 2 vehicle detecting sensors are employed for this purpose. Sensor Ncar returns TRUE if there is vehicle on N road, similar to sensor Wcar for car on W road.

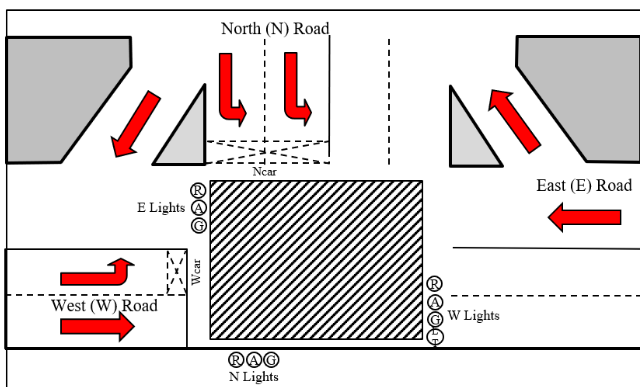


Fig. 13 Layout of T-junction traffic light

Besides following the standard traffic light pattern from Green, Amber and then Red, the traffic system is programmed in such the ways that:

- N lights allow traffics from N road to turn left while both W and E road traffics are stop.
- Traffics from W and E are allowed to travel straight while traffics at N road are stopped.
- Traffic from W is allowed to do left turn when both E and N traffics are stop. During this period, traffic from W are also permitted to go straight.
- All roads should be on red light for 1 second before the any road start green light.
- Duration for all amber lights is 2 Sec.
- Duration when both W and E roads are green is 20 Sec.
- If there is vehicle (Ncar is TRUE) on N road, the green light will be on for 10 Sec. Else (no vehicle), 1 Sec.
- If there is vehicle turning left (Wcar is TRUE) from W road, W-LT will be on for 5 Sec, followed by 3 blinks (1 Sec on and 1 Sec off) before turning off fully. Otherwise, W-LT light is skipped.

The program for the mentioned traffic light control system can be graphically represented with BM block diagram, as shown in **Fig. 14**, using the ball-maze block model as follow. Let imagine the ball start at the maze block K_1 where NS Green is turned on. Since the duration for NS Green can be either 1 or 8 Sec depending on sensor VEH1, there are 2 exits from K_1 with VEH1 as the branching condition. If VEH1 is TRUE, the ball will roll to K_2 where the delay is 8 Sec and if otherwise, it roll to K3 for 1 Sec delay. After NS green, the ball will come to block K_4 to do the NS amber, NS red, followed by EW and WE green. As the subsequent process depends on VEH2 sensor, there are 2 possible exits for K_4. One is when VEH2 sense vehicles (VEH2 is TRUE) and the other is when it sense no vehicle (VEH2 is FALSE). If there is vehicle, the ball will take the first exit to block K_5 to enable the vehicles taking right turn. If otherwise, the ball will take the second exit to block K_6 to skip the right turn. Finally, the ball will be routed back to block K1 to repeat the next cycle.

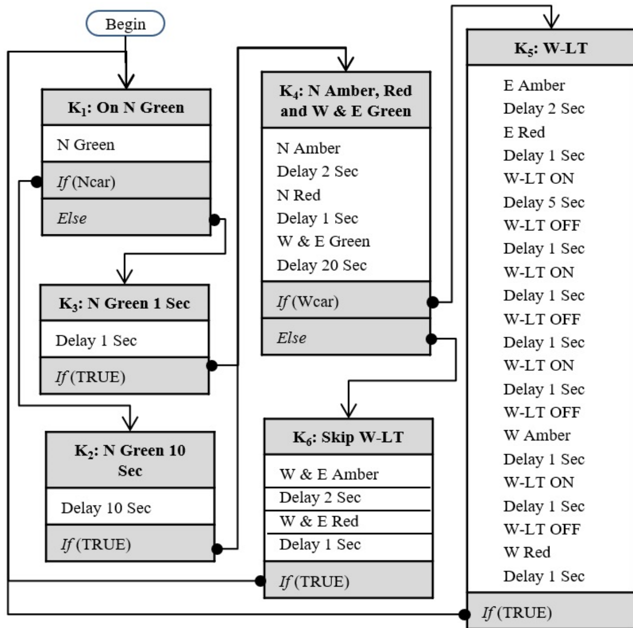


Fig. 14 BM block diagram for the traffic-light control

Upon constructing BM block diagram for the program, users can proceed to develop their source code for the diagram using the K-coding approach proposed in this paper (user can choose to use their own approach for the coding as mentioned in previous section). First, each maze block in the diagram is converted into source code using the code template of check-in and check-out branches, which are shown in **Fig. 11**, as their guideline. An example of the resulted code, using template 1, for each maze block is given in **Fig. 15**, which is simple to understand and implement.

After constructing code content for each maze block, the code can be systematically assembled together in the standard code structure as shown in **Fig. 12** where the number of maze block, *L*, in this application is 6. BM block diagram is very versatile and it can be adapted to user preference. For example, if less maze blocks is desired, some maze blocks can be combined, provided it is still within the framework of the BM block diagram. For example, blocks K_1, K_2 and K_3 can be combined into K_1' as shown in **Fig. 16** for the BM block diagram and **Fig. 17** for the code.

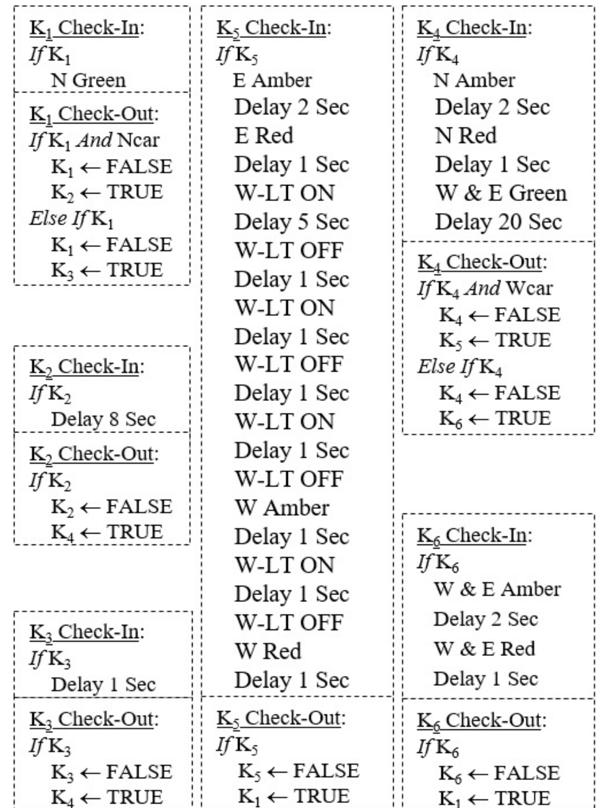


Fig. 15 Code content for each maze block

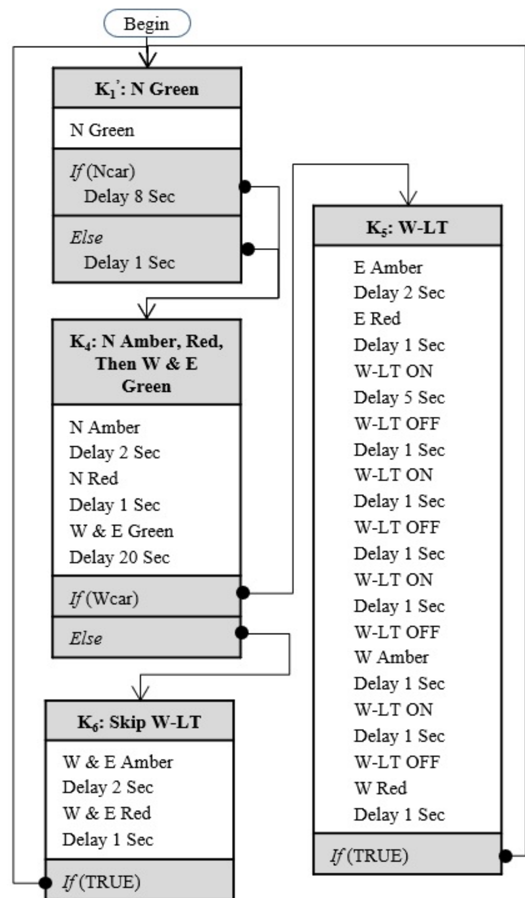


Fig. 16 BM block diagram - K_1, K_2 and K_3 combined into K_1'

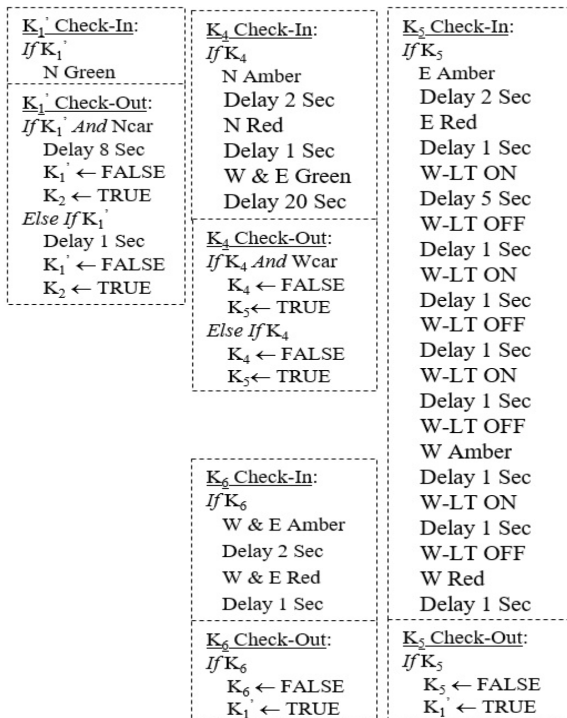


Fig. 17 Code content - K_1, K_2 and K_3 combined into K_1'

7. Conclusion

In this paper, BM block diagram is introduced as an alternative format to graphically represent of a program. It combines the advantages of both flowchart and pseudocode in a single diagram. As benefited from flowchart, it is a versatile graphical representation of program so that user can visualize the logic flow of the program effectively. At the same time, its format is closure to program code development, thus facilitating users to chunk out the flow and convert it to source code in more direct and simple way. It helps to reduce the complexity of the code structure for complex and even unstructured logic flow of a given task. The application of K-coding approach is also demonstrated, as an option, in the application example to guide users to convert BM block diagram to source code in direct and simple way. The generated code structure from BM block diagram is standard and structure-friendly.

References:

1. A.F. Blackwell, K.N. Whitley, J. Good and M. Petre, "Cognitive Factors in Programming with Diagrams," Artificial Intelligence Review, Vol. 15, Issue 1, pp. 95-114, 2001.
2. Unified Modeling Language (UML), source from www.uml-diagrams.org, (accessed on 8th Jun 2024).
3. W.R. Sutherland, "On-Line Graphical Specification of Computer Procedures," Massachusetts Institute of Technology, Dept. of Electrical Engineering, Ph.D Thesis, 1966.
4. I.E. Sutherland, "Sketchpad: A man-machine graphical communication system," Proceedings of the Spring Joint computer Conference, pp. 329–346, 1963.
5. LabVIEW, source from http://https://www.ni.com/en/shop/ labview.html, (accessed on 10th Jun 2024).
6. Scratch, source from https://scratch.mit.edu, (accessed on 10th Jun 2024).
7. A.V. Aho, R. Sethi and J.D. Ullman, "Compilers, principles, techniques," Addison-Wesley, Vol. 7, No. 8, Issue 9, 1986.
8. F.E. Allen, "Control flow analysis," ACM Sigplan Notices, Vol. 5, pp. 1–19, 1970.
9. R. Farrow, K. Kennedy and L. Zucconi, "Graph grammars and global program data flow analysis," 17th Annual Symposium on Foundations of Computer Science, IEEE, pp. 42–56, 1976.
10. C.W. Fraser and D.R. Hanson, "A retargetable C compiler: design and implementation," Addison-Wesley Longman Publishing Co., Inc, 1995.
11. T. Long, Y. Xie, X . Chen, W. Zhang, Q. Cao and Y. Yu, "Multi-View Graph Representation for Programming Language Processing: An Investigation into Algorithm Detection," Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 36, No. 5, pp. 5792-5799, 2022.
12. S. Muchnick, "Advanced compiler design and implementation," Morgan Kaufmann, 1997.
13. W. Sutherland, "On-Line Graphical Specification of Computer Procedures," Massachusetts Institute of Technology, Dept. of Electrical Engineering, Ph.D Thesis, 1966.
14. A.B. Chaudhuri, "Flowchart and Algorithm Basics - The Art of Programming," Mercury Learning and Information, 2020.
15. New World Encyclopaedia, 'Flowchart', source from http://www.newworldencyclopedia.org/entry / Flowchart, (accessed in 2013).
16. E.F. Khor, "Coding for Multitasking Without Operating System: A Method Using Simple Code Structure," CreateSpace Independent Publishing Platform, 2017.