A Malay Language-based Visual Programming Language for Personal Digital Assistant

Md. Nasir Sulaiman, Sazly Anuar, Zaiton Muda,
Hamidah Ibrahim and Aida Mustapha

University Putra Malaysia, Serdang, Selangor Darul Ehsan, Malaysia

Summary
Most of the visual programming languages that have been developed so far do not focus on Personal Digital Assistant (PDA), due to the limited screen size and resources. This paper proposes the Malay language-based Visual Programming Language (MaVi) on PDA. The syntax and the semantics of MaVi are grammatically designed based on the visual programming. The hierarchical layout technique is used to optimize the graph produced, so that it can suit on limited size of the PDA screen. The grammar of the developed language is a subset to a Java language, but it is in a graphical representation. The compilation of the program will produce a byte code, which can be executed in any PDAs that has a Java Virtual Machine (JVM). The testing on different test programming is carried out to prove that it can be executed on embedded devices and produces expected output.

Key words:
Embedded devices, Malay language-based Visual Programming Language (MaVi), Personal Digital Assistant (PDA), Java Virtual Machine (JVM).

1. Introduction

The research in the field of Visual Programming Language (VPL) has becoming more prevalent since the past few years with the increase of computer speed and the graphic capabilities [1, 2, 3, 4]. Basically, the concept of VPL is similar to any traditional textual programming languages. However, instead of following through a line of text, VPL allows the programmers to structure their codes in a pictorial display. A VPL programmer creates a program by connecting a picture or icon using a directed arrow that will produce a directed graph, which will show the flow of the program in VPL environment.

The main focus of this paper is to design and implement a compiler of MaVi that compiles visual programming styles of programs and translates it into byte code, which can be executed in any embedded devices that has a JVM. One of the challenges of this project is to deal with limited screen space. However, this will not be a limitation problem when the technique is applied on less restricted platforms.

The remainder of this paper is organised as follows. Section 2 is a short survey of related works on VPLs and some VPLs on PDA. Section 3 discusses the proposed architecture for MaVi. The design of MaVi is explained in section 4. Section 5 and section 6 are devoted to the implementation as well as testing and validation, respectively. Final section concludes the current findings and discusses some potential future.

2. Related Works

A textual programming language can be learnt only through extensive training or education through books. One must have prior programming knowledge to advance his skills pertaining to textual languages. While such knowledge would be undoubtedly useful, a VPL does not assume prior programming knowledge on the part of the user. Moreover, according to Shu [8], pictorial descriptions of programs have a number of textual description advantages. For instance, pictures convey more meaning in a concise unit of expression. Pictures also help in understanding and memorizing while making programming more interesting.

Meyer and Masterson [7] classified three main kinds of VPL. The imperative language uses Graphical User Interfaces (GUI) to smooth out the design, coding, and testing processes. The examples of imperative language include Visual Basic and Visual C++. The second kind of language is the true visual language. This type of visual language only targets at a particular problem area such as modelling, simulation and process control. The examples of true visual language include Stella, Arena, and Lab View. The third kind of language is the true general-purpose VPL. Programmers might create a program in a non-textual way. The main example of true general-purpose VPL is Prograph.

Very few academic researches have been done regarding the customization and scripting on PDA [1, 5]. For instance, the PDAGraph is a true visual language for use on handheld devices. It is a domain-specific language, specifically aimed at end-users for monitoring and controlling the wireless devices on PDA. They develop a VPL because they claim that it promotes code reuse, integration readiness, easy maintainability, better organization of the code, and easier trouble shooting.

PDAGraph researchers address a difficulty in developing Virtual Programming Environment (VPE) for PDA, which is to deal with extremely limited screen size and space. They develop a user-friendly, high-level component-based message flow VPL and environment based on research and intuition.

Another VPL on PDA is a Hyperflow [9]. Hyperflow, however, is designed for the use of children on a pen-based (but not necessarily handheld devices) multimedia computer system. It is a dataflow-based visual language but is intended for a range of software development. Hyperflow was implemented within the PenPoint Operating System only.

Not much research and development has been done in mobile computing area especially in light of programming language design [5]. Since human being has more capability to interpret an algorithm using visual language than textual language, a research on developing a VPL on PDA should be highlighted.

3. Architecture of the MaVi

The development of the compiler begins with proposing the architecture, which consists of two parts, the structural architecture and the functional architecture. Both parts of architecture are discussed as follows.

3.1 Structural Architecture

Figure 1 shows the structural architecture of the language. Basically it is implemented as a front-end to a compiler. A program is created on the language environment through the PDA's by drawing a connected structure, in the form of graph. This graph is also known as a source program or an input language with an extension of .mvp, which is the file of data representation.

The front-end of the language consists of three phases, which are the syntax analysis, semantic analysis, and code generation. The compilation of program will initially produce a Java source code with an extension of .java. Finally, the Java source code is compiled using a Java code compiler to produce a Java byte code, which is a target program in this work.

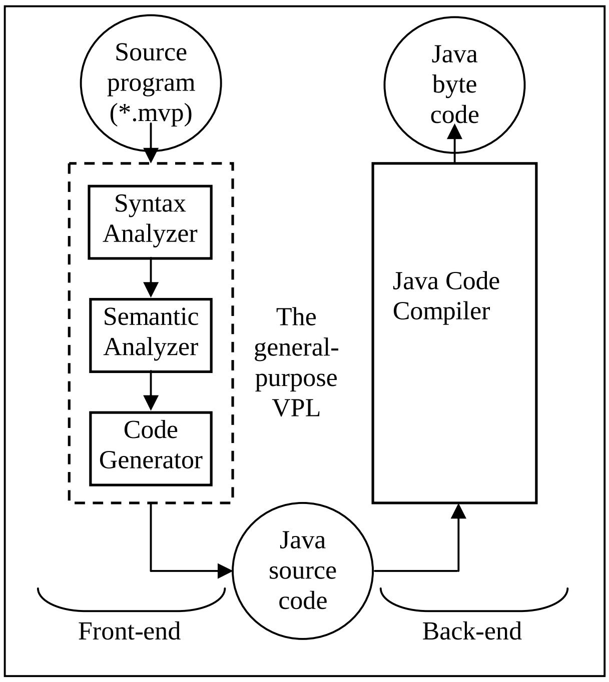


Fig 1 The structural architecture of the language

3.2 Functional Architecture

Functional architecture is an arrangement of functions and their sub-functions, and interfaces that define the execution sequencing and conditions for control or data flow. The functional architecture is depicted in Figure 2. Facing down arrows represent the flow or the steps of functionality in the language. The vertical arrow shows the flow of control, while vertical dash arrow shows the flow of data.

The entries start with "GUI Menu" where a specific instruction is selected from an existing list that has been obtained from the "System data" files. To create a program, "VPL" will be activated. It provides sets of components and edges that can be edited depending on the type of action. These actions are controlled by the rules in "System data" file.

The "Data representation" keeps the data of the program from "VPL" and sent it to "Code Generator", which will generate the Java Source code file. A "Compile to executable" is the final steps of functionality, which results in an executable Byte code file.

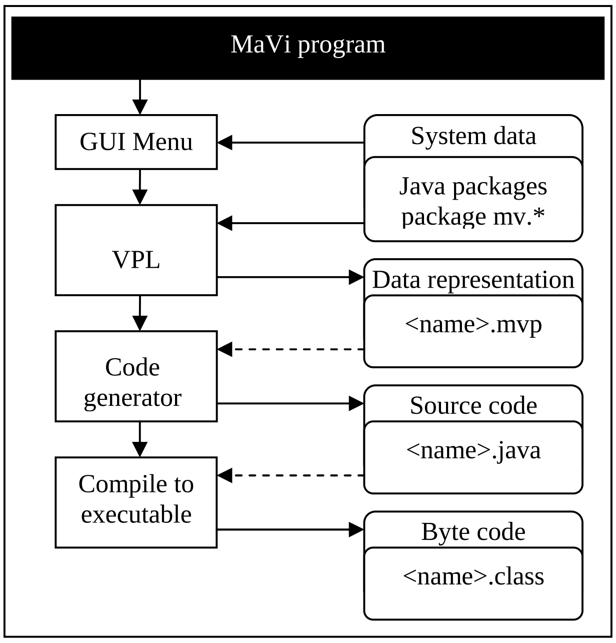


Fig 2 The functional architecture of the language environment

4. The Design of MaVi

In the first stage of designing the MaVi Environment, the list of Malay language terms that will be used by the user in the process of programming is designed. Table 1 shows the list of Malay language terms translation. Basically, the terms are representation of the components, which is also a visual representation of the grammar in MaVi Environment.

Table 1 List of Malay language terms translation in MaVi environment

Malay Terms	English Terms	Descriptions
Pembolehubah	variable	To declare a variable
Objek	object	To define an object
Ungkapan	expression	To give an expression
Jika	if	As a control selection statement
Pilihan	switch	As a control selection statement
Ulang-Bagi	for	As a control loop statement
Ulang-Selagi	while	As a control loop statement

There are two major parts of a language that have to be taken into consideration:

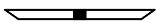
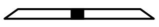
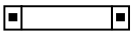
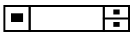
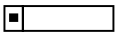
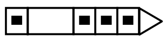
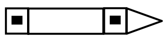
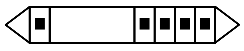
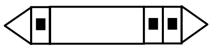
- Syntax
- Semantics

A grammar is used to accurately define the syntax of a language. Extended Backus-Naur Form (EBNF) is a common way of describing grammars. The grammars of the language follow the notation of Greedy quantifiers [10]. Basically, the grammar of the language is a subset to a grammar of Java language version 1.1.

The semantic analyzer performs semantic analysis during syntax analysis. The term semantic is frequently used to differentiate the meaning of an instruction from the language syntax. The semantic analysis in this research is an analysis of an iconic sentence to determine the underlying meaning. Actually, it is similar to any conventional programming languages only that it is represented in graphical ways.

The semantic checking is only performed on variable declaration, which involves only the variable component. For example, to declare an integer variable, user needs to declare a variable component as an integer and gives a variable name to that component. An input dialogue box will help users to declare a variable correctly. Users should follow the rules in declaring variable to avoid semantic errors. The component that is used on the language is shown in Table 2.

Table 2 The component in language based on grammars

Operation	Component	Action
Begin method node		Beginning of method
End method node		End of method
Variable		Declaring a variable name
Object		Declaring an object
Expression		Give an expression
if		If control statement
switch		Switch control statement
for		For loop control statement
while		While loop control statement

Code generator is the process by which a compiler converts a syntactically-correct program into a series of instructions that could be executed by a machine. It is used to produce programs in some automatic manner, reducing the need for human programmers to write code manually. The code generator examines the data representation file at a time using the ObjectInputStream class from java.io.* and produces the Java source code when it is compiled.

For this paper, an ObjectOutputStream class from java.io.* is used to produce the data representation in the Java object format. An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream. The objects can only be reconstituted using an ObjectInputStream. Figure 3 shows an example of data representation format in text for each component.

```
class: kelas1                                   //class file name
path: My Documents\mv\                          //class file path
file: My Documents\mv\kelas1.java               //class file path
                                           //and name
attribute: int attribute1=12;                   //class attribute if any
method: main                                    //method name
public static void main (String[] args)         //class declaration and
                                           //parameters
START                                   //<begin_method_node>
VarNode_<memory address>-SET                    //variable data
                                           //representation
END-LitNode: <value>
END-VarNode:<dataType> <varName>
ObjNode_<memory address>-SET                    //object data
ObjNode_<memory address>-CALL              //representation
END-ObjNode: <varName>
END-LitNode: <exp >                             //expression data
                                           //representation
IfNode_<memory address>-CONDITION               //if data
IfNode_<memory address>-TRUE               //representation
IfNode_<memory address>-FALSE
END-IfNode
SwitchNode_<memory address>-0                   //switch data
SwitchNode_<memory address>-1              //representation
SwitchNode_<memory address>-2
…
…
END-SwitchNode
ForNode_<memory address>-INIT                   //for data
ForNode_<memory address>-LOOP              //representation
ForNode_<memory address>-ITERATION
ForNode_<memory address>-DO
END-ForNode
WhileNode_<memory address>-INIT                 //while data
WhileNode_<memory address>-DO              //representation
END-WhileNode
END                                     //<end_method_node>
```

Fig 3 Data representation format

5. Implementations of the Compiler

The Java programming language is used in implementing the language. It is used due to its portability, its rich set of libraries, and its pervasive appearance in most academic research institutes around the world [6]. Moreover, it also benefits in cross platform support and ease of use, as evidenced by the growing transition from C++ to Java. It is also a general purpose programming language with a number of features that make the language well suited for use on embedded devices.

The language organizes the design and implementation into a set of Java Packages that are separated by function. A package is a way of grouping and naming a collection of related classes so that they can serve as a library of classes. It groups related classes, interfaces and sub packages and can be used in any program without having to place all those classes in the same directory.

5.1 User Interface

The user interface (UI) is everything designed into an information device with which human beings interact. Basically, it is a program that controls the display for the users. Figure 4 shows the main page with a project named projek1. As noted, *lajur* is the attribute for the *kira_lajur* class while *main* is a method in the *kira_lajur* class. Figure 5 is the programming environment where user uses the visual expressions in the process of programming.

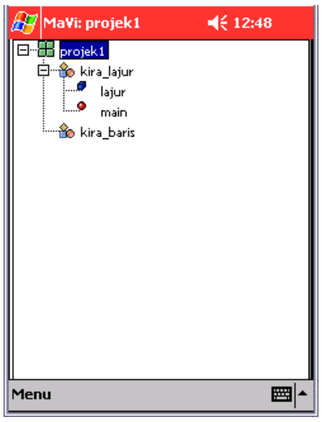


Fig 4 The main page

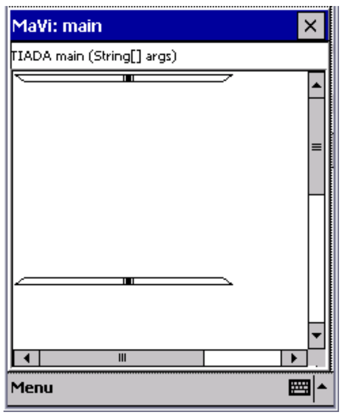


Fig 5 The MaVi Environment UI

5.2 Component Implementation

MaVi Environment represents the syntax called a component as a symbol. Figure 6(a) shows the variable syntax in MaVi Environment. The label of *Pembolehubah* component can be edited to change the data type and name of the variable. An input dialogue box will appear when user needs to change the content of selected variable. Figure 6(b) shows the input dialogue box that gives the instruction for users to define the new variable.

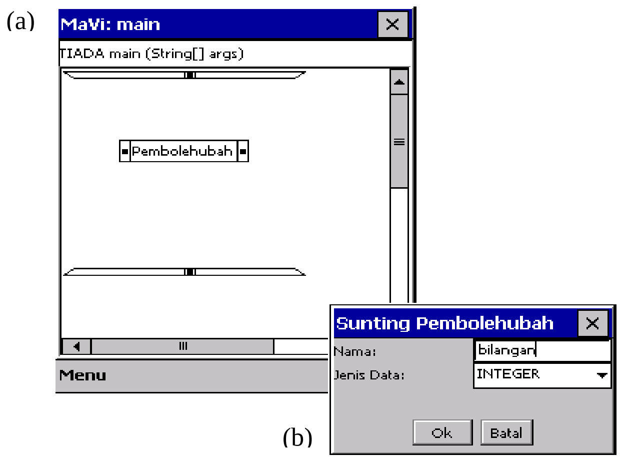


Fig 6 (a) The *Pembolehubah* component and (b) shows an input dialogue box to edit the *Pembolehubah* component.

The syntax of *Objek* component is visualized in Figure 7(a). The content of an *Objek* component can also be edited to change the name of the object. Figure 7(b) shows an input dialogue box, giving an instruction for user to edit the component.

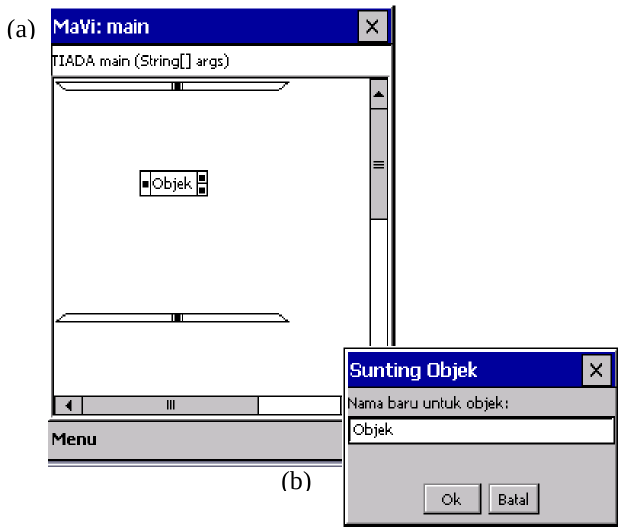


Fig 7 (a) The *Objek* component and (b) shows an input dialogue box to edit the *Objek* component.

Ungkapan syntax is implemented based on the design and it is similar with an expression in Java. The content of the *Ungkapan* component can also be edited. Figure 8(a) shows the example of *Ungkapan* component, while Figure 8(b) shows an input dialogue box that will appear for editing the *Ungkapan* component.

Next is *Pilihan* component, which is the representation of switch control statement in Java. It can have one or more cases, depending on the user defined. Users need to edit the number of cases in *Pilihan* component by inserting the new information in the input dialogue box. Figure 9(a) visualizes the *Pilihan* component in MaVi Environment, while Figure 9(b) shows the input dialogue box that will appear when user needs to edit the component.

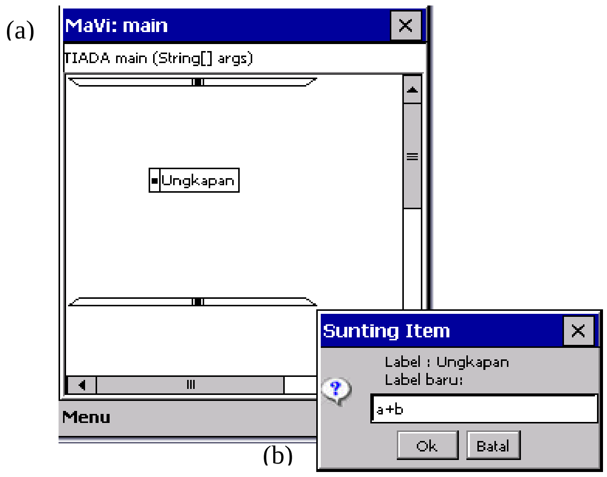


Fig 8 (a) The *Ungkapan* component and (b) shows an input dialogue box to edit the *Ungkapan* component.

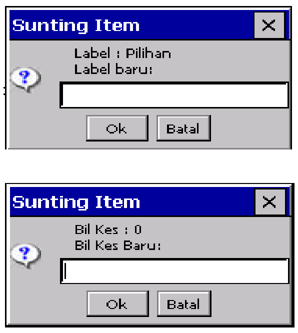


Fig 9(a) The *Pilihan* component

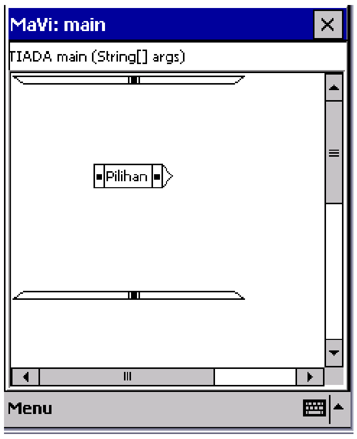


Fig 9 (b) An input dialogue box to edit the *Pilihan* component

The label of the *Jika* component cannot be edited since the component follows the syntax of the if control statement. Similar goes to *Ulang-Bagi* and *Ulang-Selagi* components. These statements cannot be edited because they are reserve words and the syntax is already fixed. The visualization of the *Jika*, *Ulang-Bagi* and *Ulang-Selagi* components in MaVi Environment are shown in Figure 10.

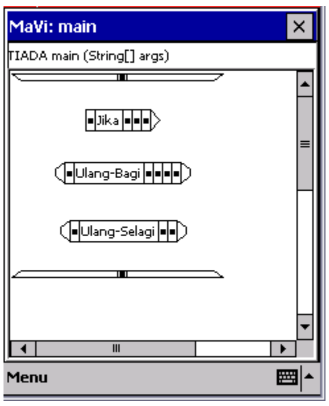


Fig 10 The *Jika*, *Ulang-Bagi*, and *Ulang-Selagi* components

5.3 Edges Implementation

Each component must be connected using an edge, which is like an arrow that will show the flow of the program. There are two types of edges that are implemented in MaVi Environment, which are the *Penuding* or the pointer edge and *Aliran* or the flow edge as explained in previous chapter. The use of pointer and flow edges is visualized in the example of a program in Figure 11.

The only way to differentiate the pointer and flow edges is from the source and the target of the edge connection. The source and target element of flow edge is the middle point of the component, which means it connects the first component at every level from the start to the end of a program. While the source and target element of pointer edge are the attachment areas, which means they connect the component at every level, to complete the semantics of the statement.

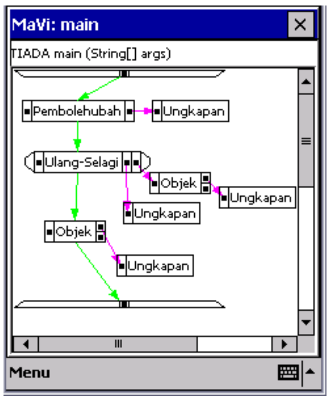


Fig 11 A program that shows the use of pointer and flow edges

5.4 Error Handler

Users have to follow the rules in creating a program in MaVi Environment. The rules are based from the grammars as explained in section 4. The error handler handles the errors that are found while creating a program. For example, when the components were wrongly connected by the edge, it will not be connected. Another example, if user does not give the exact detailed of the component value, an error message will appear and order the user to correct it.

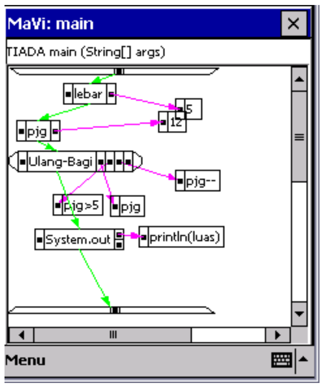
5.5 Graph Optimization in MaVi Environment

Users can optimize the graph layout in MaVi Environment at any time just by selecting the *Atur Graf* in *Atur* menu.

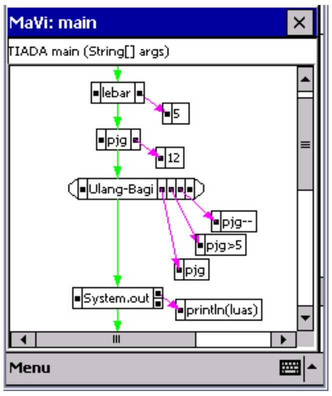
The graph optimization is based and followed from the early work of Sugiyama et. al [11]. This is one of the important issues since a minimal change to the graph drawing affects the entire graph layout. In addition, the vertical and horizontal toolbars are also used to ensure that users able to see the entire object since the small screen size limits the number of objects that can be displayed on the screen. Figure 12(a) shows an example of program in MaVi Environment created by user, while Figure 12(b) shows the similar program but after the user uses the layout optimization.

5.6 Build and Compile

The project that users create will be compiled using the MaVi Environment compiler to build the Java source code. The compilation can be done by selecting the *Bina* menu on the *Projek* menu bar. Once the users select the *Bina* menu, a message box as Figure 13 will appear. User needs to click the *Bina* button which will compile the project. A success compilation will produce Java source code, which will be located at the same directory with the project file.



(a)



(b)

Fig 12 (a) A program created by user and (b) shows the similar program but with the layout optimization

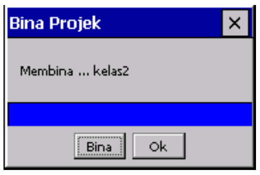


Fig 13 A message box for compilation

The next step is the compilation of the Java source code using the Java compiler, which is embedded into the MaVi Environment. For this step, user needs to select the *Kompil* menu on the *Projek* menu bar and the successful compilation will produce Java byte code. The process of build and compile is shown in Figure 14.

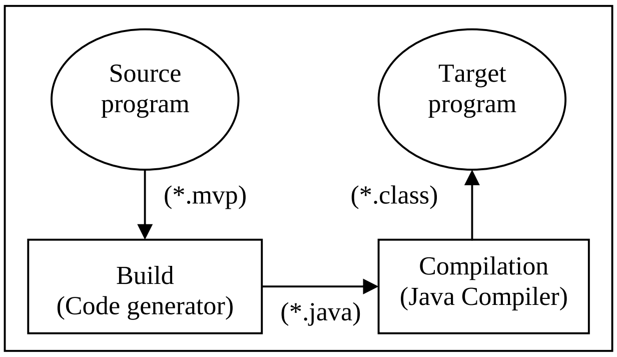


Fig 14 The process of building and compiling a program

6. Testing and Validating

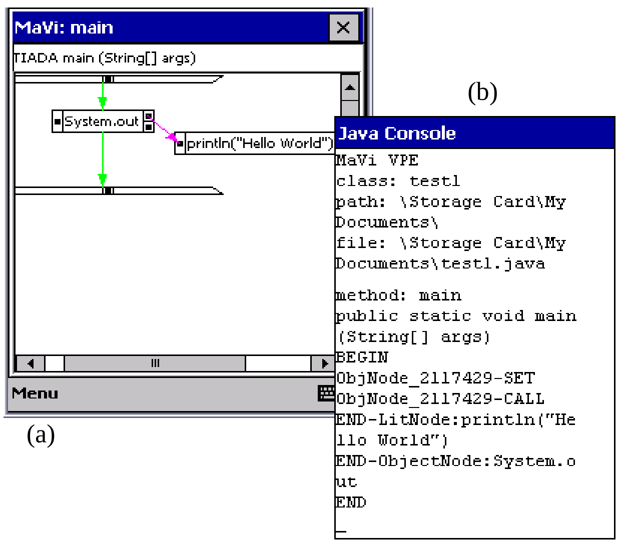
Testing is a necessary part of program creation to find any errors existed in the code. The aim of testing the language is basically to prove that it can execute and produce expected output. For this reason, a black box testing method is used.

For most programs, it is practically impossible to prove that the program is correct on all inputs. Therefore, in order to test, a collection of test program is created using the components on the language. These test programs act as inputs. The process of building and compiling the test programs are the internal details and are not taken into consideration as in black box. If the output produced as expected from various inputs, this will indicate that the program works correctly, thus validates the objective of this paper. Figure 15 shows the program that is created for this test program.

As depicted in figure 15, only two components were used, *Objek* and *Ungkapan*. A label System.out on *Objek* component is an object in Java that requests a compiler to display an output on console. The call_port is connected with an *Ungkapan* get_port, which means that the value of

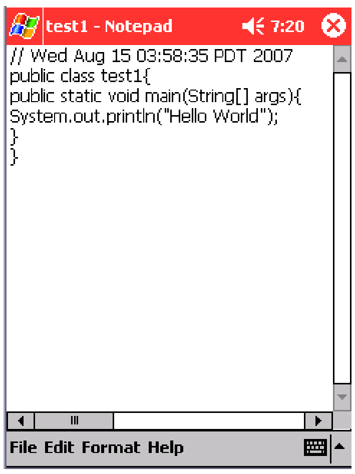
Ungkapan is called by System.out object. Figure 15(b) shows the data representation of the test program.

The building of the test program produced the output as shown in Figure 16. The compilation of the output has successfully produced a Java byte code. Therefore, the correctness of *Objek* and *Ungkapan* components in MaVi Environment is validated thus proves that it can be executed and produce expected output.



```
MaVi VPE
class: test1
path: \Storage Card\My
Documents\
file: \Storage Card\My
Documents\test1.java

method: main
public static void main
(String[] args)
BEGIN
ObjNode_2117429-SET
ObjNode_2117429-CALL
END-LitNode:println("He
llo World")
END-ObjectNode:System.o
ut
END
_
```

Fig 15 (a) Example of test program and (b) shows its data representation



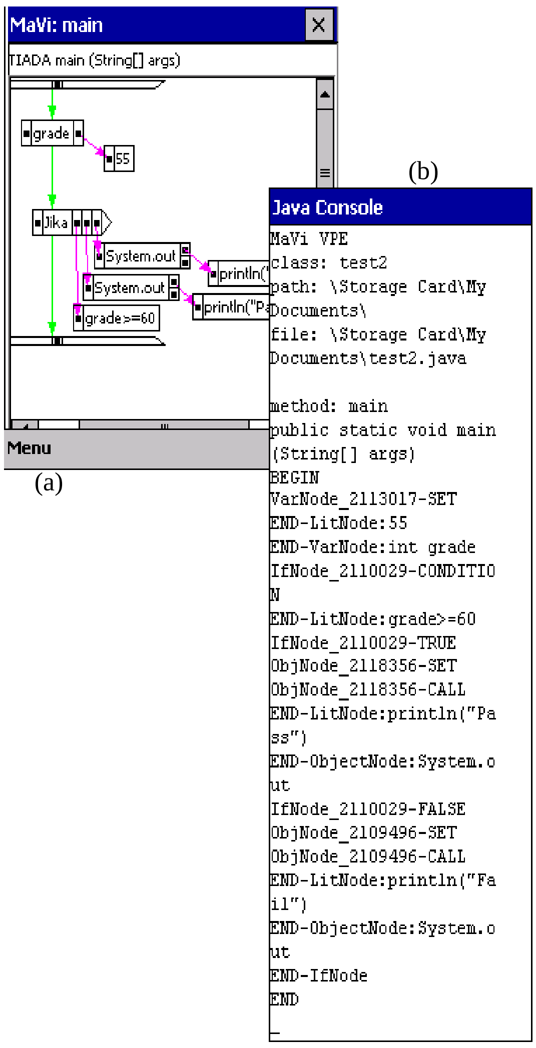
```
// Wed Aug 15 03:58:35 PDT 2007
public class test1{
public static void main(String[] args){
System.out.println("Hello World");
}
}
```

Fig 16 Java code produced from test program

The second test program is to prove that MaVi Environment can produce output using the *Jika* component. The aim is to validate the correctness of *Jika* and *Pembolehubah* components.

Figure 17(a) shows the program that is created for this test program. As can be seen, a program is constructed to give a selection result between true or false statement, based on the given condition. The given value for the variable is int gred=55, and the condition of the statement is gred>=60. Figure 17(b) shows its data representation.

The building of this program produced the output as shown in Figure 18. The compilation of the output has successfully produced a Java byte code. Therefore, the correctness of *Jika* and *Pembolehubah* components in MaVi Environment is proved thus shows that it can be executed and able to produce expected output.

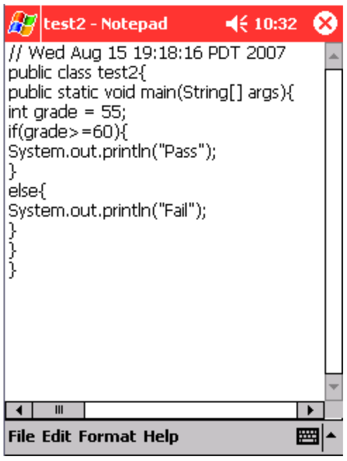


```
MaVi VPE
class: test2
path: \Storage Card\My
Documents\
file: \Storage Card\My
Documents\test2.java

method: main
public static void main
(String[] args)
BEGIN
VarNode_2113017-SET
END-LitNode:55
END-VarNode:int grade
IfNode_2110029-CONDITIO
N
END-LitNode:grade>=60
IfNode_2110029-TRUE
ObjNode_2118356-SET
ObjNode_2118356-CALL
END-LitNode:println("Pa
ss")
END-ObjectNode:System.o
ut
IfNode_2110029-FALSE
ObjNode_2109496-SET
ObjNode_2109496-CALL
END-LitNode:println("Fa
il")
END-ObjectNode:System.o
ut
END-IfNode
END
_
```

Fig 17 (a) Example program using *if* and (b) shows its data representation

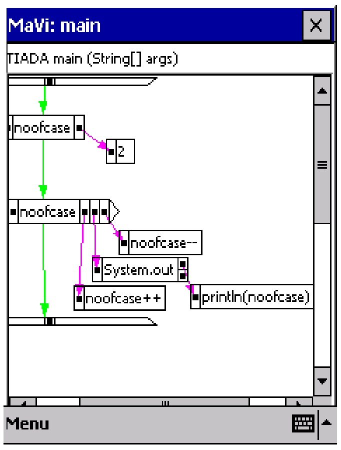
The third test program is to prove that MaVi Environment can produce output using the *Pilihan* component. The aim is to validate the correctness of *Pilihan* component. Figure

19(a) shows the program that was created for this test program while Figure 19(b) shows the data representation of the program. The building of the program produced the output as shown in Figure 20. The compilation of the output has successfully produced a Java byte code. Therefore, the correctness of *Pilihan* component in MaVi Environment is confirmed and proves that it can be executed and produce expected output.

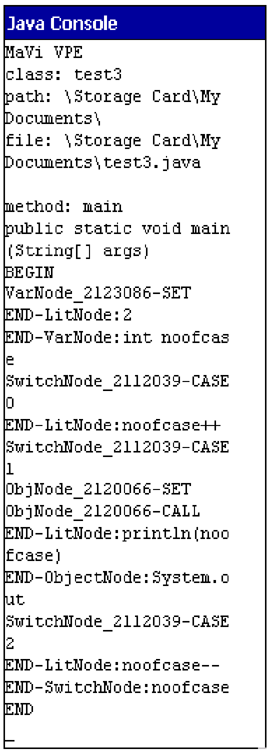


```
// Wed Aug 15 19:18:16 PDT 2007
public class test2{
public static void main(String[] args){
int grade = 55;
if(grade>=60){
System.out.println("Pass");
}
else{
System.out.println("Fail");
}
}
}
```

Fig 18 Java code produced from program of Figure 17



(a)



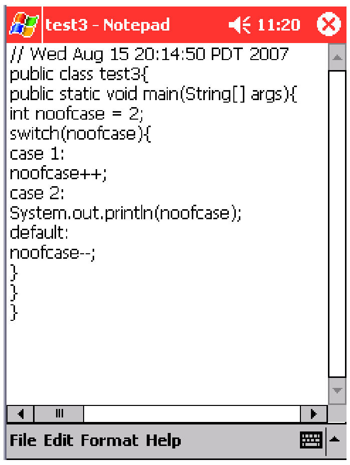
```
MaVi VPE
class: test3
path: \Storage Card\My
Documents\
file: \Storage Card\My
Documents\test3.java

method: main
public static void main
(String[] args)
BEGIN
VarNode_2123086-SET
END-LitNode:2
END-VarNode:int noofcas
e
SwitchNode_2112039-CASE
0
END-LitNode:noofcase++
SwitchNode_2112039-CASE
1
ObjNode_2120066-SET
ObjNode_2120066-CALL
END-LitNode:println(noo
fcase)
END-ObjectNode:System.o
ut
SwitchNode_2112039-CASE
2
END-LitNode:noofcase--
END-SwitchNode:noofcase
END
_
```

(b)

Fig 19 (a) Example program using *switch* and (b) shows its data representation

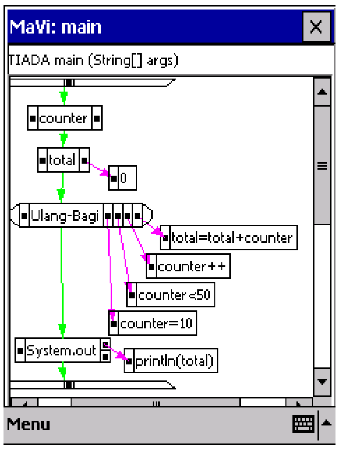
The final test program is to prove that MaVi Environment can produce output using the *Ulang-Bagi* component. The aim is to validate the correctness of *Ulang-Bagi* component. Figure 21(a) shows the program that was created for this test program while Figure 21(b) shows its data representation.

The building of the program produced the output as shown in Figure 22. The compilation of the output has successfully produced a Java byte code. Therefore, the correctness of *Ulang-Bagi* component in MaVi Environment is valid and it can be executed and produces expected output.

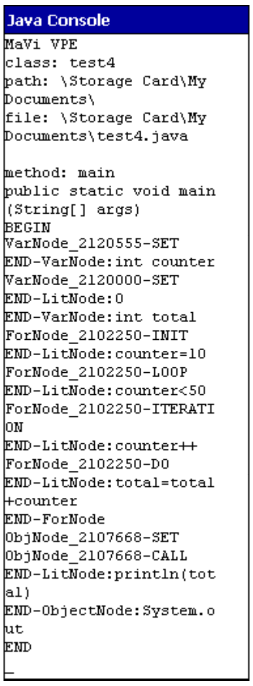


```
// Wed Aug 15 20:14:50 PDT 2007
public class test3{
public static void main(String[] args){
int noofcase = 2;
switch(noofcase){
case 1:
noofcase++;
case 2:
System.out.println(noofcase);
default:
noofcase--;
}
}
}
```

Fig 20 Java code produced from the program of Figure 19



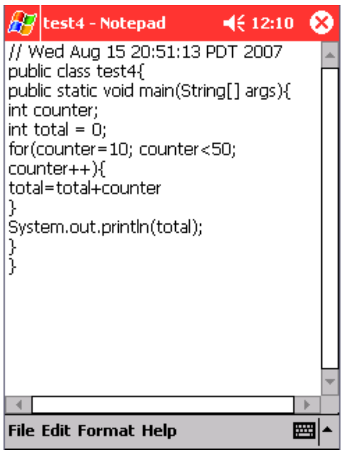
(a)



```
MaVi VPE
class: test4
path: \Storage Card\My
Documents\
file: \Storage Card\My
Documents\test4.java

method: main
public static void main
(String[] args)
BEGIN
VarNode_2120555-SET
END-VarNode:int counter
VarNode_2120000-SET
END-LitNode:0
END-VarNode:int total
ForNode_2102250-INIT
END-LitNode:counter=10
ForNode_2102250-LOOP
END-LitNode:counter<50
ForNode_2102250-ITERATI
ON
END-LitNode:counter++
ForNode_2102250-DO
END-LitNode:total=total
+counter
END-ForNode
ObjNode_2107668-SET
ObjNode_2107668-CALL
END-LitNode:println(tot
al)
END-ObjectNode:System.o
ut
END
_
```

(b)

Fig 21 (a) Example program using *for* and (b) shows its data representation



```
// Wed Aug 15 20:51:13 PDT 2007
public class test4{
public static void main(String[] args){
int counter;
int total = 0;
for(counter=10; counter<50;
counter++){
total=total+counter
}
System.out.println(total);
}
}
```

Fig 22 Java code produced from program of Figure 21

7. Conclusion

MaVi requires more attention to become successfully acceptable by all users. Many promising directions exist for extending the work performed in this paper. The following list includes some of the more interesting ones:

- More features can be added into it. Other components such as array, function, do-while statement can be added so it can be a complete environment of visual representation for programming language.
- Further evaluation is needed to make the language more widely accepted.

Finally, future works should be made for optimizing the code that will speed up the execution time. This is due to the reason that most PDA has very limited memory size and speed. There are many techniques of code optimization, which can be embedded such as global and local optimization.

References

[1] Kollet, Y. and Smedley, T.J. Message-Flow Programming in PdaGraph. In Proceedings of the 2004 IEEE Symposium on Visual Language and Human Centric Computing (VLHCC '04). 2004, pages 229-232.

[2] Burnett, M. Software Engineering for Visual Programming Languages. Handbook of Software Engineering and Knowledge Engineering. World Scientific Publishing Company. 2001, Vol. 2.

[3] Lourens, T. TiViPE – Tino's Visual Programming Environment. In Proceeding of the 28th Annual International Computer Software and Application Conference (COMPSAC '04). 2004, pages 10-15.

[4] Masterson, T. F. and Meyer, R. M. SIVIL: A True Visual Programming Language for Students. Journal of Computing Sc. in Colleges. 2001, 16(4), 74-86.

[5] Dwyer, K.D. and Smedley, T.J. A Mobile, Visual Scripting Environment for Monitoring and Controlling Wireless Devices. IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob '05). 2005, Vol. 4, pages 151-158.

[6] Gauvin, S. and Smedley, T.J. Vivid: A Framework for Creating Visual Programming Languages. In proceedings on 12th International Conference on Intelligent and Adaptive Systems and Software Engineering, San Francisco. 2003.

[7] Meyer, R.M. and Masterson, T. Towards a Better Visual Programming Language: Critiquing Prograph's Control Structures. Journal of Computing Sc. in Colleges. 2000, 15(5):181-193.

[8] Shu, N.C. Visual Programming: Perspectives and approaches. IBM Systems J., 1989, 28(4):525-547.

[9] Kimura, T.D. Hyperflow: A Uniform Visual Language for Different Levels of Programming. In Proceedings of the 1993 ACM Conference on Computer Science Indianapolis, USA, 1993.

[10] Bennett, J.P. Introduction to Compiling Techniques: A First Course Using ANSI C, LEX and YACC, International Edition, McGraw-Hill Education (Asia). 2003, pages 291-314.

[11] Sugiyama, K., Tagawa, S., and Toda, M. Methods for Visual Understanding of Hierarchical System Structures. IEEE Transactions on Systems, Man and Cybernetics, 1981, 11(2):109-125.

Md. Nasir Sulaiman is an Associate Professor in Department of Computer Science, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia. He obtained Ph. D. in Neural Network Simulation from Loughborough University, U.K. in 1994. His research interests include intelligent computing, intelligent agents, and data mining.