EMITTER International Journal of Engineering Technology
Vol. 6, No. 1, June 2018
ISSN: 2443-1168

An Embedding Technique for Language-Independent Lecturer-Oriented Program Visualization Tool

Lisan Sulistiani, Oscar Karnalim

Faculty of Information Technology
Maranatha Christian University
lisans1601@gmail.com, oscar.karnalim@it.maranatha.edu

Abstract

Program Visualization (PV) tool aims to help novice programmers to learn how a particular program works through interactive and descriptive visualization. However, most of the tools are language-dependent: they use either a language-dependent debugger or a language-dependent code to generate visualization. Such dependency may become a problem when a program written in new programming language is incorporated. Therefore, this paper proposes an embedding technique to handle given issue. To incorporate new programming language, it only needs five language-dependent features to be set: compile command, run command, library-import instruction set, file-writer function-declaration instructions, and file-writer function-invocation instruction. In general, our proposed technique works in threefold: embedding some statements to target program, generating visualization states by running the program with console commands, and visualizing the given program based on generated visualization states. According to our evaluation, proposed technique is able to incorporate program written in any programming languages as long as those languages provide required language-dependent features. Further, it is practical to be used since it still has the benefits of conventional PV despite its language-independent behavior.

Keywords: embedding technique, language independence, program visualization, educational tool, computer science education

1. INTRODUCTION

As the demand of programmer is increased, programming becomes a promising skill to be learned [1]. However, learning programming is not a trivial task; some programming concepts are either abstract or difficult to be taught [1], [2]. Consequently, computer-based educational tools have been developed [3]. These tools are expected to provide clearer understanding for novice programmers.

Program Visualization (PV) tool is a kind of computer-based educational tool that is mainly focused on visualizing program data and process [3]. It could help novices to learn how a particular program works in

debug-like fashion through interactive and descriptive visualization. According to several works [4]–[8], PV tool helps novice programmers in positive manner. It provides clearer view of data and process flow from a particular program run.

Nevertheless, to our knowledge, most PV tools are not designed to incorporate new programming language with ease. They use either a language-dependent debugger or a language-dependent code to generate visualization. Such generation mechanism takes a considerable amount of effort while a new programming language is incorporated. This paper proposes an embedding technique to cover given issue. Instead of relying to language-dependent features, it separates those features from the independent ones and makes them modifiable by users. It is important to note that separating those features require the user to have technical knowledge about target programming language when using given PV. Hence, we encourage this technique to be used only by lecturers for teaching programming material to novice programmers; lecturers are assumed to have sufficient technical knowledge to use proposed technique.

2. RELATED WORKS

When perceived based on the involvement of visualization, educational tool for learning programming can be classified into two categories: standard educational tool and Software Visualization (SV) tool [3]. Standard educational tool enhances user understanding in a conventional way. It only automates lecturer's teaching mechanism without providing a specific emphasis on visualization. The works proposed in [9]–[11] are several examples which fall into this category. On the other, SV tool enhances user understanding with a strong emphasis on visualizing software data and process [3]. Such tool works in debug-like fashion where user can see how the program works in detail. In general, SV can be further classified into two sub-categories: Algorithm and Program Visualization tool.

Algorithm Visualization (AV) tool is focused on visualizing high-level representation of program (i.e., algorithm). It is frequently used to learn well-known algorithms such as searching and sorting [12]. By providing algorithmic knowledge through AV, user is expected to be capable for writing a program related to it. Several examples of AVs are VisuAlgo [13], AP-ASD1 [14], AP-SA [15], and AP-BB [16]. In addition to the AV tools, several supportive tools for developing and maintaining the AV tools are also proposed. Three samples of such tools are JHAVE [17] (i.e., an environment for developing AV tool), JSAV [18] (i.e., a Javascript library for developing AV tool), and AlgoViz [19] (i.e., a digital library for AV tools).

Program Visualization (PV) tool is focused on visualizing direct representation of program (i.e., source code). The salient difference between PV and AV is that PV is usually featured with numerous technical information such as variable type and memory allocation. Several examples of PV tools are:

a. PythonTutor [20], a web-based PV which originally focuses on visualizing Python program.
b. Omnicode [21], an extended version of PythonTutor that incorporates live programming environment.
c. SRec [22], a PV specifically focused on teaching recursion.
d. Jelliot 3 [23], a PV specifically focused on visualizing Java program.
e. JIVE [24] , a PV visualizing Java program in either an object or a sequence diagram.
f. VILLE [25], a PV which, at some extent, enables user to incorporate new programming language with ease.

Among aforementioned PV tools, VILLE [25] is the only tool which mitigates effort required to incorporate new programming language. Instead of providing a dedicated debugger or a dedicated code to generate visualization toward new programming language, it only asks the user to provide a syntax equivalence dictionary between Java and target language (Java is VILLE's default target programming language). Even though such technique enables language independence, we would argue that it is impractical to use; providing translation for all syntaxes is exhaustive. In addition, not all syntaxes in target programming language have their equivalent form in standard Java syntaxes (e.g., list comprehension in Python). Hence, a simpler language-independent technique is required.

3. ORIGINALITY

This paper proposes an embedding technique for language-independent PV tool. It is simpler than a technique proposed in [25] in terms of incorporating new programming language. It only needs five language-dependent features to be set: compile command, run command, library-import instruction set, file-writer function-declaration instructions, and file-writer function-invocation instruction. In general, our proposed technique works in three phases: embedding some statements to target code, generating visualization states by running the code with console commands, and visualizing given code based on generated visualization states. It is true that such technique will cause several limitations. However, these limitations are acceptable under several circumstances. We will discuss the detail of limitations and circumstances further on system design.

4. SYSTEM DESIGN

Generally speaking, SV visualization can be generated in either a direct or an indirect interaction with programming language compiler. Direct interaction means that software data and process are visualized while the software is running. It is usually implemented by utilizing a particular debugger (e.g., *pdb*, a Python debugger that is used in PythonTutor [20] and Omnicode [21]). In contrast, indirect interaction means that software data and process are visualized after the software has been completely run. It is usually implemented by utilizing predefined code to capture visualization

state on given software. An example SV which utilize this technique is AP-ASD1 [14]. It generates all visualization states before visualization by embedding predefined code to both target program and the SV itself. Our technique aims to mitigate language dependency on SV. Hence, the latter technique will be used. It is implemented in our technique in three phases: embedding, generation, and visualization phase. The relation, input, and output of those phases can be seen on Figure 1.

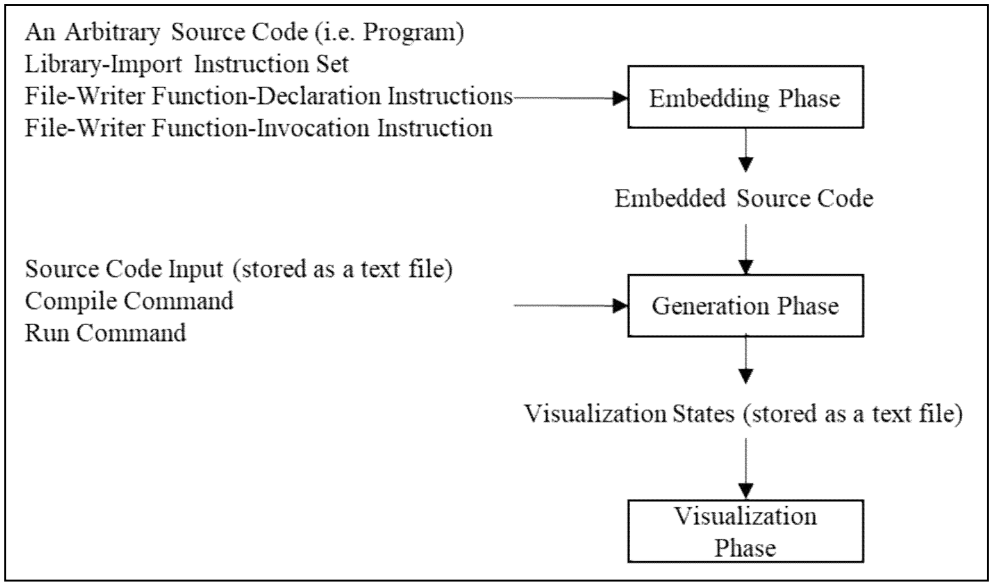

Figure 1. Phases from proposed indirect technique. It consists of three phases: embedding, generation, and visualization phase. The first two phases prepare visualization states that will be used on the last phase.

Embedding phase takes an arbitrary source code (i.e., program), library-import instruction set, file-writer function-declaration instructions, and file-writer function-invocation instruction as its input. Afterwards, it will embed the last three mentioned features into desired position on target source code, resulting embedded source code as its output. The detail of these features can be defined as follows:

a. Library-import instruction set is a bunch of instructions to import required library for writing variable states and line number to a text file.
b. File-writer function-declaration instructions represents a function declaration about writing a visualization state to a text file in predefined format. This function should accept a line number of executed instruction as its parameter and write it along with all visualized-to-be variables to a text file (for convenient access, those variables should be stored as global variables). Each variable written on resulted text file should be separated with a newline where variable value will be placed right after the variable name, separated by a colon (":"). The line number of executed instruction is stored in

similar fashion as variables except that its variable name should be referred as *linenumber*.

c. File-writer function-invocation instruction is an instruction to invoke predefined file-writer function. It can be embedded numerous times where each embedding (after an instruction) means that a visualization state toward preceding instruction will be generated. It is important to note that the line number of executed instruction will be embedded automatically by proposed technique in two phases. First, line number for each instruction will be defined by splitting inputted source code per line. Later, each file-writer function invocation will be embedded with resulted line number in regards to invocation position.

The position of each instruction-based feature will be defined manually by user according to several rationales. First, it is impractical to automatically define the position of each instruction-based feature while keeping language-independent perspective in mind. Even though the position of these features are, at some extent, considerably similar across programming languages, some slight differences still apply. For instance, in Java, library-import instruction set can only be placed after package declaration (if any); whereas, in Python, such set can be placed anywhere. Second, for recording visualization states, not all states are required to be considered. According to our informal survey toward expectant users, displaying all visualization states is difficult to be understood due to enormous information provided.

To provide clearer insight toward embedding phase, an example of embedded code in Java can be seen in Figure 2. Library-import instruction set is placed at the beginning of source code, followed by file-writer function-invocation instructions, and file-writer function-declaration instructions respectively. It is important to note that all visualized-to-be variables (i.e., *firstInteger*, *secondInteger*, and *multiplyResult*) are stored as global variables, following the rule defined for file-writer function-declaration instructions.

Generation phase takes source code input (stored as a text file), compile command, run command, and embedded code to generate a text file containing visualization states. This phase is implemented by overriding console commands to utilize language-dependent compiler (which should be installed beforehand). The detail of this phase can be seen in Figure 3. It is important to note that visualization states are not embedded as a part of program output since separating those states with real output may be impractical. Program output will be simply ignored by our proposed technique.

In most programming languages, console commands (i.e., compile and run command) need information related to target source code. In other words, these commands will be defined differently per source code even though some codes use same programming language. Since changing commands each time a new source code is incorporated is quite impractical, variable convention mechanism proposed in [11] is adapted in our technique.

Instead of writing such information directly to console commands, user can write variables related to such information and change the content of those variables in separate process; the proposed technique will automatically link each variable with its content prior processing given code. Consequently, written console commands can be generalized to be used for any source codes that use same programming language. The detail of involved variables in our technique can be seen in Table 1. To distinguish variable with standard console command's string, each variable will be prefixed with "@".

```
1  import java.util.Scanner;
2  import java.io.PrintWriter; - Library-Import Instruction Set (3 lines)
3  import java.io.File;
4  import java.io.FileOutputStream;
5  public class Main{
6      static int firstInteger = 0;
7      static int secondInteger = 0;
8      static int multiplyResult = 0;
9      public static void main(String[] args){
10         Scanner sc = new Scanner(System.in); func(10); - File-Writer Function-Invocation Instruction
11         System.out.print("first input: ");
12         firstInteger = sc.nextInt(); func(12); - File-Writer Function-Invocation Instruction
13         System.out.print("second input: ");
14         secondInteger = sc.nextInt(); func(14); - File-Writer Function-Invocation Instruction
15         multiplyResult = firstInteger * secondInteger; func(15); - File-Writer Function-Invocation Instruction
16         System.out.println("result: " + multiplyResult); func(16); - File-Writer Function-Invocation Instruction
17     }
18     public static void func(int lineNumber){ - File-Writer Function-Declaration Instructions (13 lines)
19         try{
20             PrintWriter writer = new PrintWriter(
21                 new FileOutputStream(new File("states.txt"),true));
22             writer.println("firstInteger:"+firstInteger);
23             writer.println("secondInteger:"+secondInteger);
24             writer.println("multiplyResult:"+multiplyResult);
25             writer.println("lineNumber:"+lineNumber);
26             writer.close();
27         }catch(Exception e){
28             e.printStackTrace();
29         }
30     }
31 }
```

Figure 2. An example of embedded Java code; library-import instruction set takes three lines from line 2 to 4; file-writer function-invocation instruction is embedded on line 10, 12, 14, 15, and 16; file-writer function-declaration instructions takes 13 lines from line 18 to 30.

Visualization phase takes a text file containing visualization states and visualizes it through an user interface. Recorded line number of executed instruction will be displayed by highlighting related line on source code preview; whereas, recorded variables and their values will be displayed on variable list.

To prove the applicability of our proposed technique, a prototype PV tool is developed. It is named Language-Independent Source code visualizatioN (LISN). We use *source code* instead of *program* as our terminology since it is less ambiguous; program could refer to both source code and executable file. A default view of LISN can be seen on Figure 4. For portability reason, source code and required features can be saved as a project; original source code will be stored as it is while required features will be stored as JSON files.

To use LISN, user is required to provide target source code and two feature sets. The first set is related to programming language (which should be changed only if a new programming language is incorporated) while the

latter one is related to source code (which should be changed each time a new source code is incorporated).

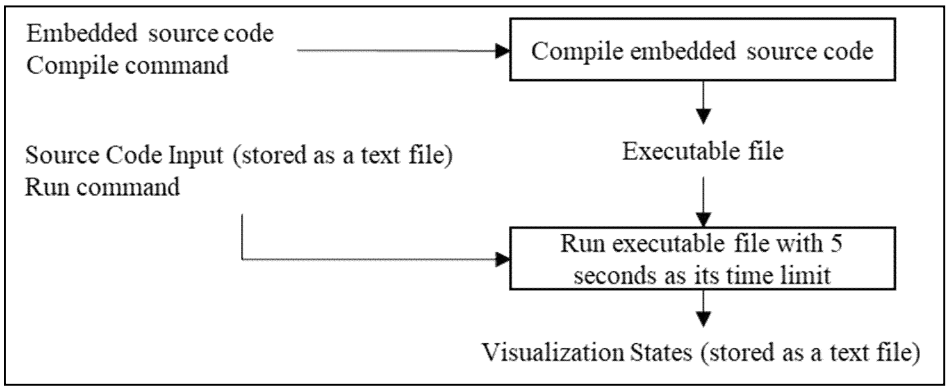

Figure 3. Sub-phases in generation phase. It consists of two sub-phases: compile and run sub-phase. At first, embedded source code will be compiled using compile command. Later, resulted executable file will be run by providing the input using file-based console pipeline mechanism. A forced termination will be conducted if given executable file takes more than 5 seconds since inputted source code may contain endless execution due to programmer error. Otherwise, visualization states will be generated.

Table 1. Variables used in console command

Variable	Reference
@dirpath	Return working directory (i.e., a location where the source code, input file, and output file are placed)
@srcname	Return source code file name
@srcnamewithoutext	Return source code file name without file extension
@exename	Return executable file name
@exenamewithoutext	Return executable file name without file extension
@inputname	Return input file name
@inputnamewithoutext	Return input file name without file extension
@outputname	Return output file name
@outputnamewithoutext	Return output file name without file extension

Programming-language feature set contains six features: compile command, run command, executable file name, input file name, output file name, and state file name. The last feature refers to a name of a file that stores all visualization states. It should be in-sync with target file name written in file-writer function-declaration instructions at embedding phase (on Figure 2, we refer such name as *states.txt*). It is true that, in this set, only compile and run command are directly related to programming language. However, other features are still considered to be included on such set; they can be only changed if a new programming language is incorporated.

Source-code feature set contains four features: source code input (as a string; LISN will convert it to a file automatically), library-import instruction set, file-writer function-declaration instructions, and file-writer function-

invocation instruction. Even though the last three features can be generalized for all source codes written in similar programming language, they are still included as source-code features instead of programming-language features; they need to be changed on source codes which name of file-writer function collides with predefined function.

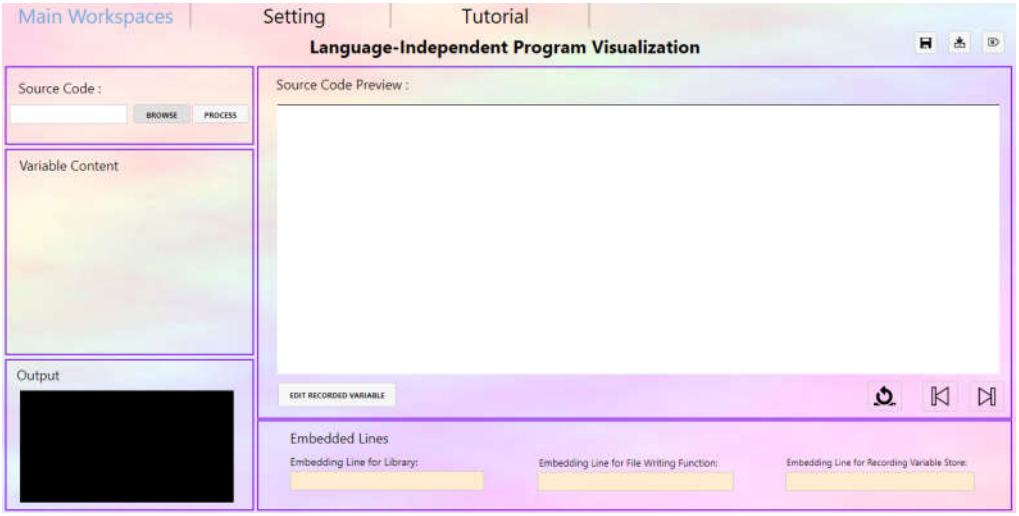

Figure 4. A default view of LISN. User can provide target source code via file chooser located at left-top panel.

For embedding phase, we assume that each line can only be embedded with one feature. Hence, embedding positions can be simply defined by clicking targeted lines on source code preview. After clicked, the color of selected lines will be changed regarding to embedded feature and selected line numbers will be displayed as comma-separated values at the bottom of source code preview (see Figure 5).

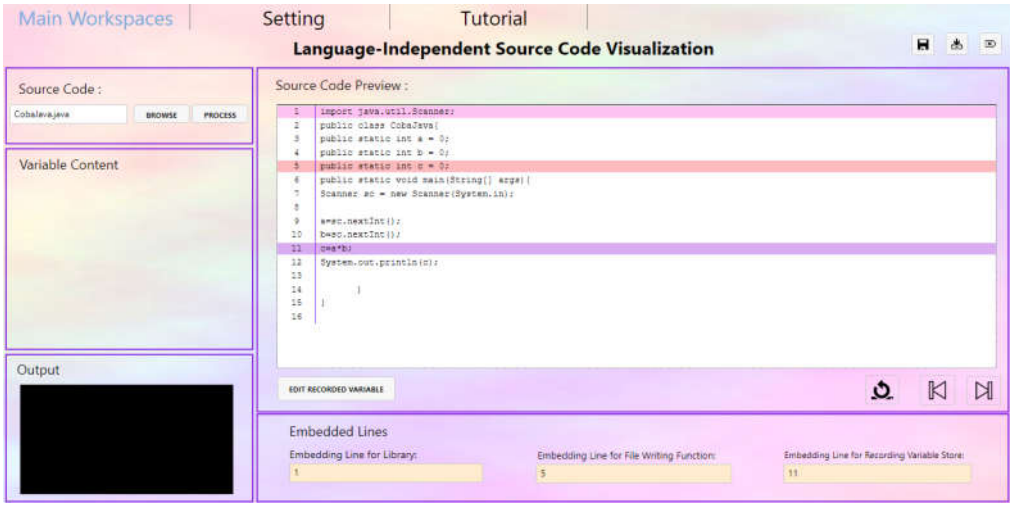

Figure 5. An example view of LISN for embedding phase Each feature is represented with a unique color on source code preview and its selected lines can be seen at the bottom of source code preview.

After target source code and feature sets have been provided, LISN will generate, compile, and run embedded code, resulting visualization states. At visualization phase, such states are then stored as in-memory list-like data structure where current index refers to current visualization state. Each visualization state will display variables' content and highlight currently executed line (see Figure 6 for example view). Next and previous state can be accessed by simply changing current index value of in-memory data structure; next state is attained by incrementing current index while previous state is attained in reverse. In LISN, this mechanism can be accessed by clicking next and previous button placed at the right-bottom of source code display.

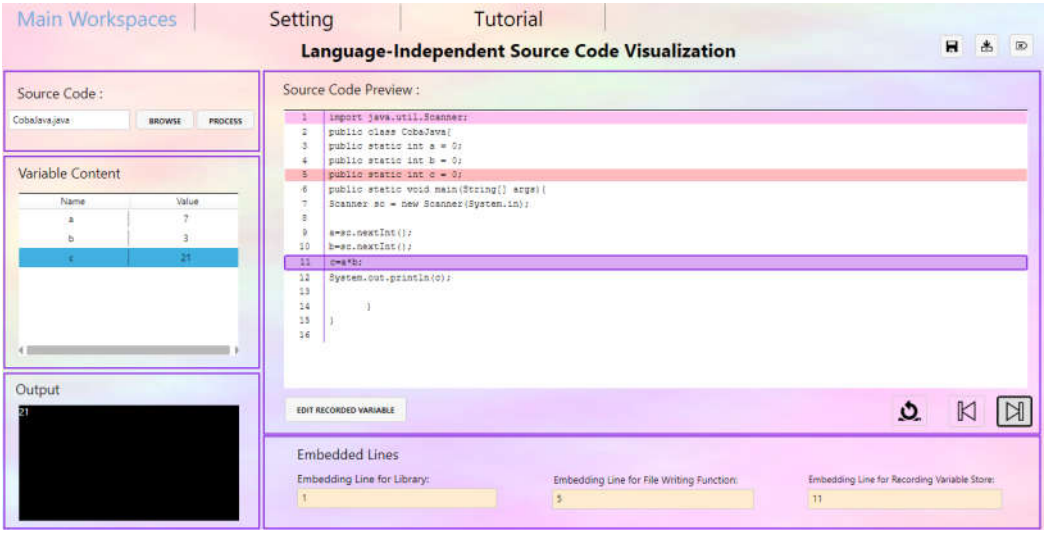

Figure 6. An example view of LISN during visualization. Variables' content is displayed in a table while currently executed line is highlighted by adding an outside border on current line. It is important to note that colors resulted from embedding phase are still displayed to remind the user which lines he has marked.

5. EXPERIMENT AND ANALYSIS

Our proposed technique was experimented from two perspectives: language independence and user satisfaction. All experiments were conducted on LISN, our prototype PV that acts as the implementation of proposed technique.

Experiment regarding language independence checks whether proposed technique is able to visualize program from any languages by only changing language-dependent features. It was conducted by designing language-dependent features for five popular programing languages and checking whether proposed technique works properly on such features.

We used Java, Python, C++, Kotlin, and Ruby as our case studies. Designed language-dependent features for those languages can be seen in Table 2, Table 3, Table 4, Table 5, and Table 6 respectively; those sets are generated by analyzing the characteristics for each programming language manually. It is important to note that *VarName* and *VarValue* in file-writer

function-declaration instructions are only some examples of storing a variable. Both terms should be replaced with recorded-to-be variable name and value before visualization. Further, instruction involving both terms could also be propagated when there are more than one recorded-to-be variable.

Table 2. Language-dependent features for Java

Feature	Value
Compile command	javac @srcname
Run command	java @srcnamewithoutext < @inputname > @outputname
Library-import instruction set	import java.io.PrintWriter; import java.io.File; import java.io.FileOutputStream;

File-writer function-declaration instructions	public static void func(int linenumber){
	try{
	PrintWriter writer = new PrintWriter(new FileOutputStream(new File(@targetfilename),true));
	writer.println("VarName:"+VarValue);
	writer.println("linenumber:"+linenumber);
	writer.close();
	}catch(Exception e){
	e.printStackTrace();
	}
	}

File-writer function-invocation instruction	func(linenumber)

Table 3. Language-dependent features for Python

Feature	Value
Compile command	
Run command	@srcname < @inputname > @outputname
Library-import instruction set	
File-writer function-declaration instructions	def func(linenumber): file=open(@targetfilename,"a") file.write("VarName:"+(str)(VarValue)) file.write("\n") file.write("linenumber:"+(str)(linenumber)) file.write("\n")
File-writer function-invocation instruction	func(linenumber)

After language-dependent features for all languages had been defined, we evaluated it based on 8 x 5 source codes focusing on 8 Introductory Programming concepts (e.g., branching, looping, and function) and 5 target programming languages (i.e., Java, Python, C++, Kotlin, and Ruby). According

to our evaluation, it is clear that proposed technique works correctly on given languages.

Table 4. Language-dependent features for C++

Feature	Value
Compile command	g++ -o @exenamewithoutext @srcname
Run command	@exename < @inputname > @outputname
Library-import instruction set	#include <fstream>

File-writer function-declaration instructions	void func(int linenumber)
	{
	ofstream myfile;
	myfile.open (@targetfilename, std::ios_base::app);
	myfile << "VarName"<<VarValue<<"\n";
	myfile << "linenumber:"<<linenumber<<"\n";
	myfile.close();
	}

File-writer function-invocation instruction	func(linenumber)

Table 5. Language-dependent features for Kotlin

Feature	Value
Compile command	kotlinc @srcname -include-runtime -d @exename
Run command	java -jar @exename < @inputname > @outputname
Library-import instruction set	import java.io.PrintWriter import java.io.File import java.io.FileOutputStream

File-writer function-declaration instructions	fun func(linenumber: Int) {
	try {
	val writer = PrintWriter(FileOutputStream(File(@targetfilename), true))
	writer.println("VarName:" + VarValue
	writer.println("linenumber:" + linenumber)
	writer.close()
	} catch (e: Exception) {
	e.printStackTrace()
	}
	}

File-writer function-invocation instruction	func(linenumber)

Experiment regarding user satisfaction checks whether proposed technique is practical to be used. It was conducted by performing questionnaire survey to 10 lecturer assistants at our university. Lecturer assistants were chosen as our respondents instead of lecturers according to three rationales. First, lecturer assistants, at some extent, share similar

teaching responsibility as the lecturers. Second, lecturer assistants know student perspective better considering they are closer to students when compared to lecturers. Third, lecturer assistants have looser schedule than the lecturers, meaning they can participate in our experiment.

Table 6. Language-dependent features for Ruby

Feature	Value
Compile command	
Run command	@srcname > @outputname
Library-import instruction set	
File-writer function-declaration instructions	def func(linenumber) File.open("target.txt", "a") do |f| f.write("VarName:") f.write(@VarValue) f.write("\n") f.write("linenumber:") f.write(linenumber) f.write("\n") end end
File-writer function-invocation instruction	func(linenumber)

Each respondent were asked to rate 3 statements in 7-points Likert scale (1 refers to completely disagree and 7 refers to completely agree). The detail of each statement including its ID can be seen on Table 7. The first two statements check whether proposed technique still holds similar benefits as conventional PV while the last statement checks the applicability of our proposed embedding technique. To mitigate biased result, the first author simulated how LISN works to each respondent prior distributing the questionnaire.

Table 7. Questionnaire survey statements which should be rated in 7-points Likert scale by lecturer assistants.

ID	Statement
Q1	Displaying variable content can help user to teach student regarding given source code
Q2	Highlighting currently-executed line can help user to teach student regarding given source code
Q3	Proposed embedding technique enables user to incorporate new programming languages with ease

According to our respondents, all statements were positively agreed (see Figure 7). Each of them was assigned with mean score higher than 5.5 (i.e., a minimum top quartile threshold in 7-points Likert scale). In other words, it can be stated that proposed technique still holds similar benefit as

conventional PV while keeping language-independent behavior in mind. Q3 was assigned with the mediocre mean score, that is higher than Q1's and lower than Q3's. Hence, it can be stated that, in our proposed technique, language independence is more prominent than displaying variable content while still less prominent than highlighting currently-executed line. It is natural that language independence is less prominent than highlighting currently-executed line on language-independent PV; such highlighting mechanism is one of the most leading PV features for learning programming.

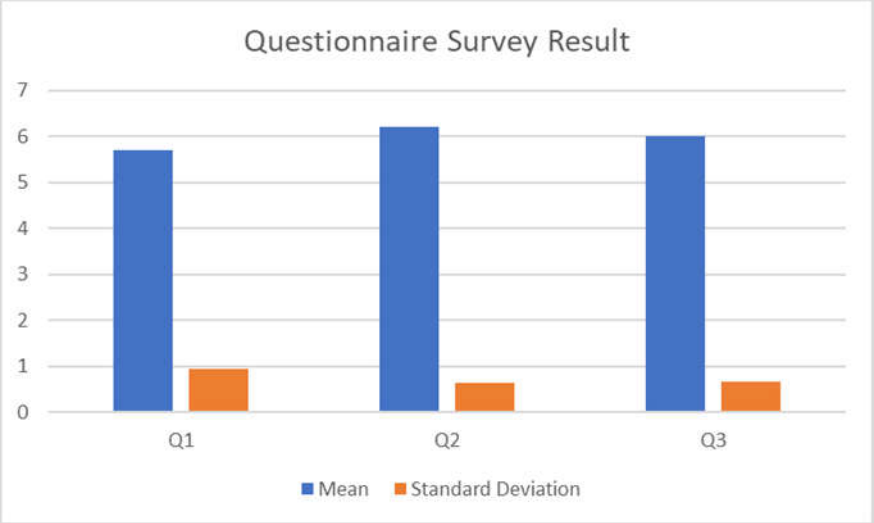

Figure 7. Questionnaire survey result. Horizontal axis represents survey statements while vertical axis represents resulted score.

In terms of variability, respondents experienced the highest variability while rating Q1; some respondents felt that variable name and value is sufficient for learning programming while the others felt that additional variable data should be featured (e.g., variable reference and type). In contrast, the respondents experienced the lowest variability while rating Q2. We would argue that such finding is natural considering highlighting currently-executed line is implemented in similar manner as in most PV tools.

6. CONCLUSION

In this paper, an embedding technique for language-independent PV tool has been developed. It can visualize a program written in a new programming language by setting only five language-dependent features: compile command, run command, library-import instruction set, file-writer function-declaration instructions, and file-writer function-invocation instruction. According to our evaluation, two general findings can be deducted. First, proposed technique, at some extent, is able to incorporate program written in any programming languages as long as those languages provide required language-dependent features. Second, proposed technique

is practical to be used since it still has the benefits of conventional PV despite its language-independent behavior.

Three future directions are provided for this paper. First, we plan to evaluate language independence of our technique on more programming languages (e.g., Javascript). Second, we plan to evaluate proposed technique in real teaching environment through quasi-experiment [26] and check whether the result is promising. Third, we plan to provide a more general framework for covering advanced programming aspects such as high performance computing [27] on visualization.

REFERENCES

[1] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer science education*, vol. 13, no. 2, pp. 137–172, 2003.

[2] M. Kölling, "The greenfoot programming environment," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 14, 2010.

[3] J. Sorva, V. Karavirta, and L. Malmi, "A Review of Generic Program Visualization Systems for Introductory Programming Education," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 1–64, Nov. 2013.

[4] E. Kaila, T. Rajala, M. J. Laakso, and T. Salakoski, "Effects of Course-Long Use of a Program Visualization Tool," in *Australasian Computing Education Conference*, Brisbane, 2010.

[5] O. Karnalim and M. Ayub, "The Effectiveness of a Program Visualization Tool on Introductory Programming: A Case Study with PythonTutor," *CommIT (Communication and Information Technology) Journal*, vol. 11, no. 2, 2017.

[6] O. Karnalim and M. Ayub, "The Use of PythonTutor on Programming Laboratory Session: Student Perspectives," *KINETIK*, vol. 2, no. 4, 2017.

[7] S. M. Cisar, R. Pinter, and D. Radosav, "Effectiveness of Program Visualization in Learning Java: a Case Study with Jeliot 3," *International Journal of Computers, Communications & Control*, vol. 6, no. 4, 2011.

[8] O. Karnalim and M. Ayub, "A Quasi-Experimental Design to Evaluate the Use of PythonTutor on Programming Laboratory Session," *International Journal of Online Engineering (iJOE)*, vol. 14, no. 02, pp. 155–164, Feb. 2018.

[9] J. Á. Velázquez-Iturbide, A. Pérez-Carrasco, J. Á. Velázquez-Iturbide, and A. Pérez-Carrasco, "Active learning of greedy algorithms by means of interactive experimentation," in *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education - ITiCSE '09*, New York, New York, USA, 2009, vol. 41, p. 119.

[10] O. Debdi, M. Paredes-Velasco, and J. Á. Velázquez-Iturbide, "GreedExCol, A CSCL tool for experimenting with greedy algorithms," *Computer Applications in Engineering Education*, vol. 23, no. 5, pp. 790–804, 2015.

[11] E. Elvina and O. Karnalim, "Complexitor: An Educational Tool for Learning Algorithm Time Complexity in Practical Manner," *ComTech:*

Computer, Mathematics and Engineering Applications, vol. 8, no. 1, p. 21, Mar. 2017.
[12] C. A. Shaffer *et al.*, "Algorithm Visualization: The State of the Field," *ACM Transactions on Computing Education*, vol. 10, no. 3, pp. 1–22, Aug. 2010.
[13] S. Halim, Z. Chun KOH, V. Bo Huai LOH, and F. Halim, "Learning Algorithms with Unified and Interactive Web-Based Visualization," *Olympiads in Informatics*, vol. 6, pp. 53–68, 2012.
[14] L. Christiawan and O. Karnalim, "AP-ASD1 : An Indonesian Desktop-based Educational Tool for Basic Data Structure Course," *Jurnal Teknik Informatika dan Sistem Informasi*, vol. 2, no. 1, Apr. 2016.
[15] F. C. Jonathan, O. Karnalim, and M. Ayub, "Extending The Effectiveness of Algorithm Visualization with Performance Comparison through Evaluation-integrated Development," in *Seminar Nasional Aplikasi Teknologi Informasi (SNATI)*, 2016.
[16] S. Zumaytis and O. Karnalim, "Introducing an Educational Tool for Learning Branch & Bound Strategy," *Journal of Information Systems Engineering and Business Intelligence*, vol. 3, no. 1, p. 8, Apr. 2017.
[17] T. L. Naps, J. R. Eagan, L. L. Norton, T. L. Naps, J. R. Eagan, and L. L. Norton, "JHAVÉ—an environment to actively engage students in Web-based algorithm visualizations," in *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education - SIGCSE '00*, New York, New York, USA, 2000, vol. 32, pp. 109–113.
[18] V. Karavirta and C. A. Shaffer, "JSAV: the JavaScript algorithm visualization library," in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education - ITiCSE '13*, New York, New York, USA, 2013, p. 159.
[19] C. A. Shaffer, M. Akbar, A. J. D. Alon, M. Stewart, and S. H. Edwards, "Getting algorithm visualizations into the classroom," in *Proceedings of the 42nd ACM technical symposium on Computer science education - SIGCSE '11*, New York, New York, USA, 2011, p. 129.
[20] P. J. Guo, "Online python tutor: embeddable web-based program visualization for cs education," in *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13*, New York, New York, USA, 2013, p. 579.
[21] H. Kang and P. J. Guo, "Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations," in *The 30th ACM Symposium on User Interface Software and Technology (UIST*, 2017.
[22] J. Á. Velázquez-Iturbide, A. Pérez-Carrasco, and J. Urquiza-Fuentes, "SRec: : an animation system of recursion for algorithm courses," in *Proceedings of the 13th annual conference on Innovation and technology in computer science education - ITiCSE '08*, New York, New York, USA, 2008, vol. 40, p. 225.

[23] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, "Visualizing programs with Jeliot 3," in *Proceedings of the working conference on Advanced visual interfaces - AVI '04*, New York, New York, USA, 2004, p. 373.
[24] P. Gestwicki and B. Jayaraman, "Interactive Visualization of Java Programs," in *Symposia on Human Centric Computing Languages and Environments*, 2002.
[25] T. Rajala, M.-J. Laakso, E. Kalla, and T. Salakoski, "VILLE: a language-independent program visualization tool," in *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88*, Darlinghurst, 2007, pp. 151–159.
[26] J. W. Creswell, *Educational research : planning, conducting, and evaluating quantitative and qualitative research*. Pearson, 2012.
[27] I. E. W. Rachmawan *et al.*, "An Embedded System for applying High Performance Computing in Educational Learning Activity," *EMITTER International Journal of Engineering Technology*, vol. 4, no. 1, pp. 46–64, Aug. 2016.