JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 38, 1089-1108 (2022)
DOI: 10.6688/JISE.202211_38(6).0001

# Using User-Defined Domain-Specific Visual Languages to Modularize Programs for Conducting Experiments[*]

YUNGYU ZHUANG[1,+], JUI-HSIANG KAO[2], KUAN-SHANG LIU[1] AND CHIA-YU LIN[1]
[1]*Department of Computer Science and Information Engineering*
*National Central University*
*Taoyuan, 32001 Taiwan*
*E-mail: yungyu@ncu.edu.tw[+]; {106525011; 106522124}@cc.ncu.edu.tw*
[2]*Department of System Engineering and Naval Architecture*
*National Taiwan Ocean University*
*Keelung, 202301 Taiwan*
*E-mail: jhkao@mail.ntou.edu.tw*

Experimental programs for conducting related scientific computing or engineering simulations often share common steps but differ in their workflows. Although switching between different workflows within a single program is possible, those switches prevent from understanding the individual experimental workflows. To domain experts, it is usually tricky to modularize experimental programs for maintenance and comprehension. Suppose common steps in these workflows can be wrapped up as components in a tiny visual language. The experiments can be expressed as programs written in that language and even constructed by drag-and-drop. It not only hides implementation details in each step but also improves program comprehension. However, existing domain-specific visual languages (DSVLs) are not targeted for full customization so far as we know. We propose customizing a user-defined DSVL to represent different experimental workflows and follow Dijkstra's sequencing discipline in structured programming to develop a proof-of-concept framework. For discussion, a tiny DSVL for running wind turbine system simulation was then built upon as an example, and a comparison with existing visual frameworks was made based on diagram style, component set, and program construction. Our approach can help domain experts to express the experimental concern and quickly construct programs for running related experiments. Supporting complex syntax and parallel computing are included in our future work.

***Keywords:*** domain-specific language, visual programming language, code modularity, workflow management system, flowchart

## 1. INTRODUCTION

Programming techniques are essential in science and engineering domains, but scientists and engineers might not be experts in programming. Without advanced programming techniques or good programming support, they may repeatedly write similar programs for conducting scientific computing and engineering simulations. Such programs can be difficult to understand, modify, and reuse. How to support domain experts in developing these programs is a known issue [1] that has been studied for many research activities, including domain-specific languages (DSLs) and visual programming languages (VPLs). Many DSLs, such as COBOL, SQL, Verilog, and MATLAB, have shown effectiveness in help-

Received September 29, 2021; revised October 31, 2021; accepted November 13, 2021.
Communicated by Shin-Jie Lee.
[*] This work was supported in part by the Ministry of Science and Technology (Taiwan) as "Developing guidelines and utilities for rewriting scientific computing applications with modern programming languages" [MOST 107-2221-E-008-024-MY3].

ing users develop and maintain programs for a particular domain. Their language constructs are tailored to fit the needs of a specific domain, and therefore, the semantics tend to be simpler than those of general-purpose languages (GPLs). The total number and combinations of built-in language constructs are small, resulting in a straightforward syntax, simple semantics, and a shallow learning curve. For the code written in SQL, for example, the use of SELECT-FROM-WHERE has only one meaning. Programmers will not unintentionally misuse the clause and cannot intentionally abuse it. For programmers who read and maintain the code, the risk of misunderstanding the code can be reduced. However, customizing and extending the language constructs in these DSLs are often tricky. For people who are not programming experts, adding or removing language constructs is not easy, nor is it realistic in a proprietary standalone DSL. In order to bridge the gap between GPLs and DSLs, domain-specific embedded languages (DSELs) were developed [2]. It enables the use of various libraries and toolchains for the host language, such as parsers, debuggers, and editors. Domain-specific libraries developed for Python, such as the SciPy Stack and TensorFlow, can be classified as the DSEL approach since the semantics behind their programs are quite different from the ones in plain Python. In those programs, function calls can be regarded as the language constructs in the DSEL to instantiate objects and perform operations. Matrices are represented with array objects instead of native nested lists, and assignments/evaluations are done by function calls rather than variable access. However, there is still room to help programmers design and understand the relations between the function calls in these domain-specific libraries. Unfortunately, checking whether a combination of function calls and language constructs is valid requires advanced programming techniques. Although many sophisticated DSL frameworks were developed for either shallow embedding or deep embedding, DSL development itself remains difficult for people who are not programming experts.

On the other hand, VPLs have been intensively discussed and known for their ability to help novice programmers [3, 4]. They provide a visual environment to generate programs in a multidimensional (basically two-dimensional) fashion rather than one-dimensional text streams. VPL systems usually make the mechanics of programming more manageable, lower the barriers to programming [5], and remove the necessity to remember the syntax of programming languages [6]. VPL has been used to build environments for either standalone DSLs or GPLs for a long time, and recently, it has attracted interest again because of the appearance of Scratch [7] and Blockly [8]. Many research activities are devoted to developing domain-specific visual languages (DSVLs) for various applications [9-13] and discussing the design and modeling of DSVLs [14, 15]. However, existing DSVLs focus on implementing programs with a given component set based on existing language constructs rather than using a customized component set for hiding implementation details in individual components. On the other hand, DSVL frameworks [16, 17] are usually targeted at programming experts. As a result, nonexperts cannot easily design their fully customized DSVLs for writing and running experimental programs.

To simplify the development of experimental programs in specific domains, we propose using a lightweight, user-defined DSVL to modularize code. Our approach is targeted at domain-specific experiments described as workflows. Programmers in science and engineering domains can manage their experiment programs by wrapping their code pieces as visual components and conduct experiments by arranging them in a visual framework. To the best of our knowledge, no research activity has been devoted to supporting people who are not programming experts in designing a customized set of language constructs for conducting experiments. Existing DSVLs usually support a given set of language constructs to program, and the use of DSL/DSVL frameworks usually needs programming skills and requires knowledge about metaprogramming. The contribution of this paper is threefold. First, we discuss conducting experiments as a concern whose code needs to be modularized and separated from other concerns. Second, we propose using user-defined DSVLs to separate the concern of conducting scientific and engineering experiments from others. Third, we concretely show how a tiny DSL built atop our proof-of-concept implementation can modularize experimental programs.

## 2. MOTIVATION

Many scientific computing and engineering simulation programs are programmable to conduct a series of related experiments. These experiments often share standard modules in the workflows, and conducting an experiment is to select some of them for execution [18]. In this case, instead of developing and maintaining a set of programs, writing a configurable program is desirable. For example, researchers in mechanical engineering might develop an idea on system controlling logic and want to run simulations to verify their hypotheses. They may first implement a draft workflow for system control and find an optimized workflow for a given scenario after repeatedly reordering steps or changing conditions. Every step in a workflow is a particular operation, and a workflow represents an algorithm for performing proper operations according to system conditions. In order to change the order of steps, researchers might use parameter files, preprocessor directives, and if-else/goto statements to modify the concatenation of steps for switching between different workflows. Furthermore, selections and repetitions are also heavily used for controlling the execution flow inside a workflow. Researchers may experimentally set certain combinations of selections and repetitions of these steps to construct a workflow, observe the running results, and modify the combinations to rerun the experiment.

These concatenations (sequences), selections (branches), and repetitions (loops) are kinds of switches, and they are exactly the ones discussed in structured programming [19]. However, in this case, they are used to configure program execution rather than handle data; they are an approach to generate a set of programs for experiments. These programs that result from the switches can be considered a software product line [20]. It might remind readers of feature-oriented programming [21], but here we focus on switching between execution flows inside a single program. The code of these switches, *i.e.*, the configuration for controlling execution flow, is a concern that differs from those in individual steps; they are implemented for conducting experiments and need to be separated from other implementations [22]. As shown in Fig. 1, we name it "the experimental concern" to distinguish it from other concerns. When modifying the code for running experiments, the code for other concerns should not be touched. On the other hand, the code for the experimental concern should not be affected by the modification in individual steps. If these codes for the experimental concern can be separated from individual step implementations, code modularity could be improved. Therefore, the reusability and maintainability would be better. In this way, programmers who run experiments can be different from those who implement steps. Although many advanced techniques such as metaprogramming are available to modularize the experimental concern, they are too difficult to learn and use for people who are not programming experts. A more straightforward approach is to wrap up implementation details for individual steps with functions, but there are still two challenges. First, the code for the experimental concern is still there if programmers use only a single program to run various experiments. Writing different programs for individual workflows is helpful, but a good development strategy is needed to manage these programs. Second, the constraints on the combination of function calls must be carefully considered and noted. Suppose a function call can only be followed by specific functions calls [23]. This constraint should be well designed in library functions, otherwise it will be represented within the experimental concern code. If we divide the single program into a set of programs, these constraints will be eliminated.

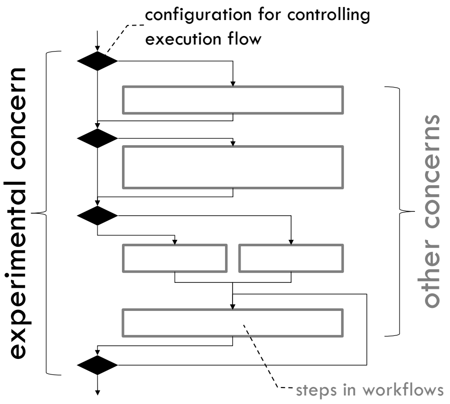


Fig. 1. The experimental concern.

This observation led us to combine the concepts of user-defined DSL and VPL for experimental programs. DSLs can help separate the experimental concern from other concerns while defining the constraints on the combination of language constructs. Every code block in Fig. 1 for implementing a step can be wrapped up as a language construct for hiding implementation details and checking the constraints. Experimental programs can then be described with these language constructs and contain only the experimental concern. On the other hand, VPLs can help the understanding of workflows. People who are not programming experts can write and read experimental programs visually. Although research activities have been performed on DSVLs, to the best of our knowledge, no one has been devoted to delivering a lightweight framework for conducting experiments with user-defined DSVLs.

## 3. A VISUAL FRAMEWORK FOR USER-DEFINED DSLS

We propose representing the steps in workflows with a user-defined DSL and using a visual framework to quickly assemble programs for running experiments, as shown in Fig. 2. The functions for individual steps can be considered the language constructs of a DSL, and every program for running a particular experiment is a program written in this

DSL. Because the DSL hides the implementation details for individual steps, programs written in this DSL can purely represent workflows, and visualizing these programs can help to understand these workflows. Scientists and engineers who know the details of the steps are DSL developers, and scientists/engineers who design and run experiments are DSL users.

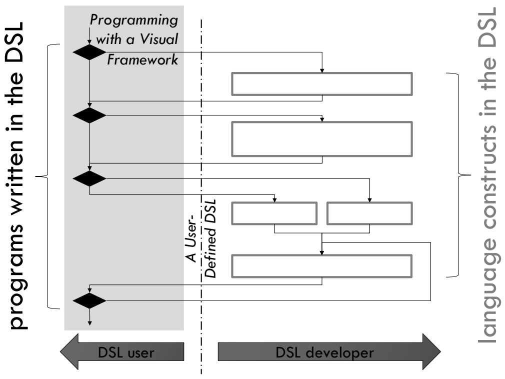

programs written in the DSL
Programming with a Visual Framework
A User-Defined DSL
language constructs in the DSL
DSL user
DSL developer

Fig. 2. Representing with a user-defined DSL.

## 3.1 The Elements in the Experimental Concern

As mentioned in Section 2, the experimental concern is implemented for quickly switching between execution flows for different experiments. Programmers may want to exchange two steps in a workflow, modify the condition for selecting between two steps, or change the condition for repeating a step. For example, wind turbine system developers may run a series of simulations with different model controlling logics and observe the gained energy output under random wind speeds. They may first implement several models for working under different wind speed conditions and define a controlling logic to switch between these models in order to gain the maximum energy output. According to the simulation results, they can refine configurations in the controlling logic. Since there are various design goals such as high efficiency and low noise, developers may design different controlling logics and run similar simulations repeatedly. We can concretely consider what configurations are involved in the experimental concern with this wind turbine system example. Because the experimental concern code is written for setting how to execute the steps for workflows, it possibly consists of three things:

***Switches for assembling different workflows.*** They are used for statically generating a set of programs. During the execution of the program, these switches are not changed. For example, wind turbine system developers might fuse different workflows into a single program for convenience, though only one workflow will be selected for program execution. They are usually implemented with parameter files and preprocessor directives. We suggest dividing the program into a set of programs rather than switching between the code inside a program. Although combining several workflows in a program is a general strategy to avoid copy-and-paste of code, it also decreases the readability of programs. If every step can be represented with a DSL language construct, we can simply write programs for every workflow without worrying about copying and pasting the details in steps.

***Configurations within a workflow.*** These configurations are used to modify the setting for a workflow and are usually implemented with if-else and do-while approaches. In the wind turbine system example, they are the logic that controls the system to switch between models for reacting to wind speeds. Although they also rely on runtime conditions in the program, they are different from the conditions used inside each step. They are some sort of hyperparameter for configuring the workflow rather than individual steps. This part should be maintained in the programs written in a DSL.

***Constraints on the step combination.*** In a series of experiments, some constraints might be placed on how the steps can be set since some step combinations might not be reasonable. For example, a wind turbine system running on a specific model might only be allowed to switch to certain models. These constraints are similar to API usage rules in library functions [23]. If we use a DSL to hide the implementation details in steps and write different programs for individual workflows, the constraints on arranging the steps must be preserved. These constraints are often hidden in the switches for assembling different workflows and the configurations within a workflow. After steps are represented with language constructs in a DSL, the constraints should be encoded in the usage of these language constructs.

## 3.2 The Requirements of the Visual Framework Design for User-Defined DSLs

To support user-defined DSLs with a visual framework for improving the code modularity of experimental programs, we analyze user scenarios and list the following requirements of the design of this visual framework:

1) *The visual framework must support at least the three types of decomposition in structured programming: concatenation (sequence), selection (branch), and repetition (loop).* Since the three types of decomposition are the basics of programming languages and the experimental concern code heavily relies on them, they must be supported with visual programming. Users can design their own DSLs based on them rather than creating them every time.
2) *Every visual component should be mapped to a single language construct in the user-defined DSL and vice versa.* This one-to-one mapping is to avoid ambiguity in program visualization. After programs are implemented with the user-defined DSL, they can always be loaded to obtain the same program visualization results, regardless of whether they are written in the visual framework or by any other editors. In addition, these visual components must be designed for being used with the three types of decomposition.
3) *A program is a code piece written in the user-defined DSL for representing an individual workflow.* In other words, programs running on this visual framework should contain only the code in the user-defined DSL. It is encouraged to define a program for a single workflow statically to avoid mixing multiple workflows.
4) *A visual editor must be provided to visualize, store, and load programs written in the user-defined DSL.* Supporting only visual programming or program visualization is insufficient. A visual editor for both reading and writing programs must be provided to simplify the creation and modification of experiment programs written in the user-defined DSL.
5) *The constraints on the combination of language constructs should be checked when*

*connecting them in the visual editor.* Although the usage of language constructs in a program will be checked during code generation, immediately performing the check on the visual editor can help users see where the problem is. It means that the usage check on the user-defined DSL must be visualized, for example, showing errors when trying to connect two visual components illegally.

## 4. A PROOF-OF-CONCEPT IMPLEMENTATION: MASONPY

Based on the requirements stated above, we implemented a draft version of this visual framework, MasonPy[1], to show the feasibility and discuss its usage and benefits. Currently, MasonPy supports only restricted topology flowcharts rather than allowing arbitrary connections between any visual component. MasonPy only provides the three types of decomposition by default: sequence, branch, and loop; it minimizes the set of built-in components in user DSVLs. We followed Dijkstra's sequencing discipline in structured programming [19] and flowchart techniques for structured programming [24]. This design decision allows the development and usage of user-defined DSVLs to stay simple for our potential users. Compared with other flowchart standards, such as ANSI/ISO symbols, MasonPy only has flowline, terminal, process, and decision components. Although flowcharts have been studied for a long time and programming experts might recommend UML-like diagrams rather than flowcharts for software development, many research results still show that flowcharts help understand and discuss uncomplicated cases. Furthermore, the engineering workflow discussed in Section 2 can be naturally described with a flowchart. The three types of decomposition are even introduced in computational thinking education [25] because they are the basics of programming and are supported in most languages.

MasonPy users can create their DSLs on top of it and let their users write programs with the DSL. Typically, developing DSL components requires domain and programming knowledge. However, MasonPy provides necessary built-in components and a component template to decrease the difficulty of creating user-defined DSLs. Furthermore, the user-defined DSL users can generate programs by dragging and dropping the components on our visual editor instead of writing textual code directly. Thus, the users of MasonPy need little programming knowledge, and the users of these DSLs can even be unaware of the implementation details of DSLs. Defining a DSL helps separate the experimental concern and modularize the code for different steps in workflows.

### 4.1 The Syntax and Interpretation

MasonPy is built on top of Python. Currently, the syntax of DSLs on MasonPy is based on the syntax of Python. Every step in a workflow is represented as a list in Python:

```
[Identifier, Type, InputLinks, OutputLinks, Parameters]
```

where `Identifier` is the unique name of the component in the program, and `Type` is used to specify which kind of DSL component it is. The elements in `InputLinks` and `OutputLinks` are named slots to connect with other components, and `Parameters` are the parameters for

[1] This version is available on our project page: http://psl.csie.ncu.edu.tw/masonpy.

this component. For example, a Decision component named isSpeedLargerThanFive is written as follows:

```
['isSpeedLargerThanFive', 'Decision',
  ['beforeCheck'], ['thanDo', 'elseDo'], ['speed', '>', 5]]
```

Here, the last element, Parameters, is also a list, which means that this component will check whether the speed value is greater than five. If this is true, it will continue to execute the component connected by the output link thanDo. Otherwise, it goes to the component pointed by the other output link elseDo. The input link beforeCheck specifies that this check should be performed after the components whose output links contain beforeCheck. Note that MasonPy allows maintaining an environment to store the states of programs, such as speed.

MasonPy will evaluate these lists for representing an experiment to construct the workflow and run the experiment. The string specified in Type is used to instantiate the corresponding Python object, and the string specified in Identifier is its name. Every list then becomes a call to a function object with the given parameters. The implementation of MasonPy is similar to the command pattern [26] and relies on the built-in eval function in Python. Thanks to Python, little effort is needed to pass these lists and convert between values and variable names in MasonPy. Beautifying the syntax, such as removing quotation marks and square brackets, is on our to-do list.

Users can first implement components for the steps in their workflows with Python. In some sense, Python's syntax has similarities to Fortran and C, which many scientists and engineers prefer. Moreover, Python is known for its wide choice of libraries, and it is helpful to customize user components. After defining standard components in workflows as MasonPy modules, users can simply generate programs by modifying these lists in Python or even arranging these modules in the visual editor, *i.e.*, creating a workflow by drag-and-drop. To implement programs for running experiments, we usually divide code into several functions in a library and then list function calls in order. With MasonPy, users can wrap code in components and visually check the order and conditions in the workflow. Furthermore, similar to other DSL frameworks, MasonPy can separate the roles of DSL developers and DSL users: the creators of components are DSL developers, and the users of components are DSL users. However, because MasonPy is targeted at simple usage, people who are not programming experts can be either DSL developers or DSL users.

## 4.2 The Built-in Components and Recognized Errors

MasonPy provides a minimal component set containing Start, End, Decision, Loop, and Process, so users only have to define the components used in their specific domain. The list representation of the five built-in components is shown in Table 1, and the errors visually recognized by MasonPy are summarized in Table 2. Start and End represent the start point and the endpoint of a workflow and have only one output link or input link, respectively. The remainders, namely Process, Decision, and Loop, represent sequence, branch, and loop, respectively. The Process is the prototype of user processes–users can generate their processes by copying and customizing this component. Decision accepts conditions, one input link, and two output links, where the conditions are used to decide which output link should be selected. An example of using Decision is shown in Fig. 3 (a). Loop allows one input link and two output links along with counter or stop conditions to select a specified output link repeatedly; it is a special kind of Decision. Note that because the loops in workflows are actually represented by decisions, the Loop in MasonPy is given for convenience; it is close to do-while in other programming languages. The difference from Decision is that Loop owns a counter. Fig. 3 (b) shows an example of using Loop.

**Table 1. The five built-in components.**

| Type | InputLinks | OutputLinks | Parameters |
|---|---|---|---|
| Start | [] | [output] | n/a |
| End | [input1, ...] | [] | n/a |
| Process | [input1, ...] | [output] | [ [var1-to-set, value], ...] |
| Decision | [input1, ...] | [out-for-true, out-for-false] | [var-to-compare, op, value] |
| Loop | [input1, ...] | [out-for-end, out-for-continue] | [var-to-compare, op, value] |

**Table 2. Recognized errors.**

*End Point Error*

| Type | number of end points |
|---|---|
| Start | $\neq 1$ |
| End | $\leq 0$ |

*Connection Error*

| Type | number of input | number of output |
|---|---|---|
| Start | $\neq 0$ | $\neq 1$ |
| End | $\leq 0$ | $\neq 0$ |
| Process | $\leq 0$ | $\neq 1$ |
| Decision | $\leq 0$ | $\neq 2$ |
| Loop | $\leq 0$ | $\neq 2$ |

*Parameter Error*

| Type | error inside |
|---|---|
| Decision | comparison |
| Loop | comparison, counter |

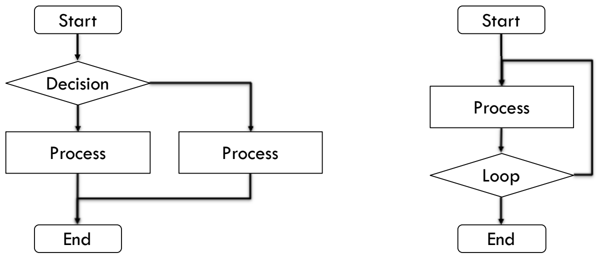


Fig. 3. Using Decision (a) and Loop (b).

### 4.3 Wrapping up Steps as Components

Users can modularize the code for common steps in their workflows with a `Process` and then generate their program by composing these `Process` components. This approach can help programmers divide their programs into several modules based on the steps and let their users quickly assemble their programs with these modules. Every kind of `Process` is a language element in the DSL, and the workflows are programs written in this DSL. We realized that creating a DSL from scratch is trivial to people who are not programming experts. In our current implementation, users need to create subclasses of `Process` to customize their `Process` – this task might ask users to understand the mechanism in object-oriented programming. Therefore, MasonPy comes with a component template to create a `Process` by filling in the given template to simplify the usage. With the provided template, MasonPy users can be unaware of MasonPy implementation details. Users are also encouraged to fill in docstrings for their processes as component description. Once users copy and modify the template, corresponding components will be created in the visual editor.

### 4.4 A Visual Editor

Steps in a workflow can be wrapped as a component in the visual editor, and then the visual editor can be used to arrange and control the components. By connecting the components and setting the conditions in components, the DSL users can avoid directly handling parameter passing and arbitrarily setting parameters. The parameters are written inside components and hidden from DSL users. MasonPy reads the parameters in the components and lists them in the visual editor; only the listed parameters can be set, and their types will be checked. An illegal link connection will also result in errors. MasonPy also benefits from the libraries for visualization in Python, such as Matplotlib [27], to show the simulation results with plotting. Fig. 4 is the screenshot of a simulation program running in the visual editor. The programs are stored as the code in the user-defined DSL and can be loaded later for modification. In our current implementation, the programs are stored as Python lists. Although it is not far from readable, making the syntax more elegant is our future work. We also plan to add more visual supports, including highlighting the present executed component in the workflow, grouping components to reuse as a bundle, and showing a pane for global variables.

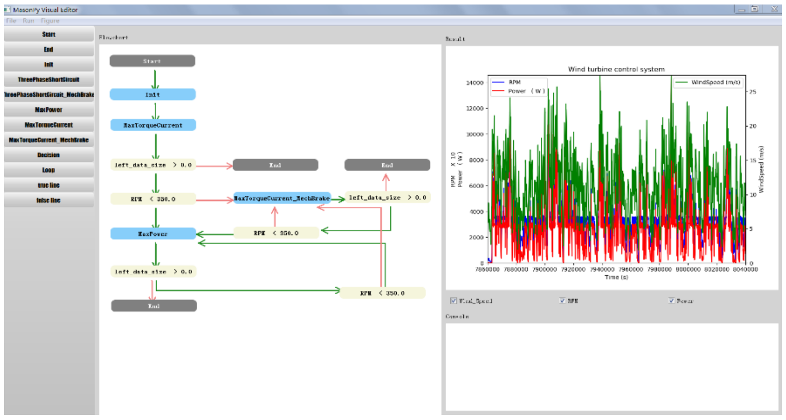


Fig. 4. Screenshot of a simulation program running on the visual editor.

## 5. DISCUSSION: A TINY DSL FOR WIND TURBINE SYSTEMS

Many research activities have been conducted to develop strategies for controlling a wind turbine system's rotating speed to attain high stability and high performance [28-32]. The manufacturing process of wind turbine systems usually starts with a design stage, followed by a farm test for verification. The farm test items include power performance, life cycle, duration, and electronic control, and the process of the farm test needs at least two years. If the test results are not satisfying, engineers must return to the design stage. We developed Hybrilog, a tiny DSL for describing hybrid controlling logic for wind turbine systems [33]. It is built on top of MasonPy and has only six components, including an initial step and five control models based on maximum power point tracking, three-phase short circuit, and maximum torque loading current. Engineers can write Hybrilog programs to simulate farm tests at the design stage to shorten product development time. Below we discuss how Hybrilog modularizes experimental programs and hide implementation details.

### 5.1 A Preliminary Survey on the Usability

To understand the usability of Hybrilog and obtain initial feedback on user experience, we invited five subjects in the engineering domain to develop programs for simulating hybrid controlling logic. Note that this is not an assessment but a very preliminary survey. They are either engineers who graduated from the engineering domain or master's students in the engineering department who have basic knowledge of wind turbine systems. After we explained the design problem of control logic for wind turbine systems, we asked them to use Fortran, Hybrilog, and MasonPy visual editor to write simulation programs based on given control logics. The time they spent implementing three given logics is shown in Table 3. Note that to fairly compare the effort of implementing Hybrilog and Fortran, we did not ask the subjects to write down all the Fortran code. Instead, the subjects only needed to write subroutine calls for different models, for example, "`CALL maxpower(...)`". In other words, we compared the effort of exchanging steps and configuring workflows. The results show that Hybrilog did help users quickly write and modify the control logic, and using the visual editor can further save time. Regarding user experience, all the subjects thought the visual editor provided a better debugging experience, especially in modifying programs. Compared with Fortran, the implementation details of control models can be hidden better.

**Table 3. The average time to develop with Fortran, Hybrilog, and the visual editor.**

| The average time | Fortran | Hybrilog | Visual Editor |
|---|---|---|---|
| Implement logic 1 (simple) | 11.4 | 5.8 | 3.0 |
| Implement logic 2 (complex) | 61.6 | 38.0 | 6.8 |
| Modify logic 2 into logic 3 | 26.6 | 16.0 | 1.2 |

in minutes

### 5.2 Comparing with the Original Fortran Code

To discuss how MasonPy can help code modularization of experimental programs, we compared the code written in Hybrilog with the original Fortran code. We analyzed the Fortran implementation for logic 2, as shown in Fig. 5. The five models used in the logic are wrapped as subroutines placed at the file end. The main function starts with an initialization followed by five code blocks for the checks. In every code block, the conditions such as wind speed and RPM are used to determine which model to switch to and call the corresponding subroutines. This code is not complicated, and honestly speaking, it is well structured with Fortran's mechanism in some sense. However, it has many conditional jumps and goto statements for switching between models. When engineers want to rearrange the components in the workflow or configure the check conditions, the modification might be error-prone. Because these simulations are exactly workflows, representing them with the three types of compositions is better than using code blocks.

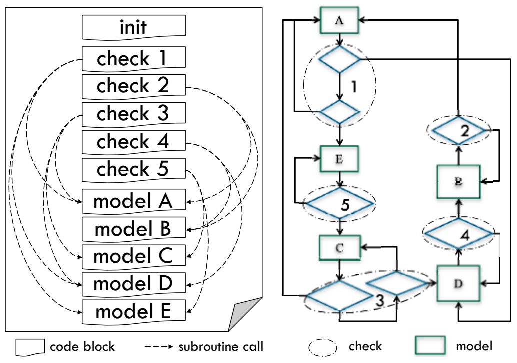


Fig. 5. The original implementation for logic 2.

Regarding the lines of code of implementations, we compared the original Fortran implementation we have, the ones implemented by the subjects in our preliminary survey, and the ones written in Hybrilog, as shown in Table 4. Note that initially, we did not have the implementations for logic 1 and logic 3; they were directly implemented in Hybrilog after we developed Hybrilog/MasonPy. The column of subjects' code shows the numbers of lines of code implemented by the subjects mentioned above, which vary in a wide range because of the differences in using branch statements, goto statements, and variables. Unsurprisingly, the code written in Hybrilog can be much shorter than the Fortran code because every line of Hybrilog code is longer and contains branch information. Note that here, the numbers of lines of code do not include the initialization part and subroutines but only contain the check and workflow. However, it is interesting that using Hybrilog and the visual editor can avoid the difference caused by programmers. All the subjects have the same Hybrilog implementation for a given logic, and it is identical to what we implemented. The results show that using MasonPy to draw these workflows can avoid confusion about the equivalence between different implementations.

**Table 4. Lines of code of the implementations.**

| Lines of Code | Original | Subjects' | Hybrilog |
|---|---|---|---|
| Implementation of logic 1 | n/a | 23–48 | 14 |
| Implementation of logic 2 | 97 | 59–98 | 26 |
| Implementation of logic 3 | n/a | 69–105 | 28 |

## 6. THE POSITION OF MASONPY AND RELATED WORK

Since the code written in a user-defined DSL atop MasonPy is essentially a diagram. Here, we discuss the similarities and differences between MasonPy and existing visual frameworks for drawing/showing program diagrams. The diagrams supported in existing works are kinds of directed graphs, especially flowcharts in a broad sense, emphasizing the structure of program code, the execution flow of control, the flow of data propagation, or the software design models. Note that several remarkable works are already discussed in the taxonomies of visual frameworks [4, 34-39], and we are not going to give an in-depth review of existing visual frameworks here. Instead, we focus on how MasonPy is different from related work to clarify the position of MasonPy. Below we consider from the viewpoint of diagram style, component set, and program construction.

### 6.1 Diagram Style

According to the connection in diagrams, we can broadly classify visual frameworks into four categories: nested blocks, control flow diagrams, data flow diagrams, and modeling diagrams. Nested blocks are the representation used in block programming [40], allowing users to snap graphical pieces together as working on a jigsaw puzzle. Every block corresponds to a language construct in conventional textual programming languages. For example, get/set, if-else, and repeat-until, and blocks like repeat-until can further contain other blocks. The style was developed by Logo Blocks at the MIT Media Lab and inherited by Scratch [7], Snap! (formerly BYOB) [41], MIT App Inventor [42], Blockly [8], *etc.* Strictly speaking, they are not a diagram because the components are fit together rather than connected by lines. Nested blocks are very close to code because they naturally represent the scope of code blocks in programs. In some sense, it is a kind of indent representation to help users understand the program code structure and can be transformed to code without complicated handling. Recently, it has been widely adopted by programming environments for education and has attracted many users. Although some are essentially targeted at encouraging invention and innovation [7, 43], this graphical representation greatly helps novice programmers and children try programming without much programming knowledge. From the viewpoint of program generation, this design also simplifies the implementation of underlayer frameworks. On the other hand, the grain size of components tends to be smaller because they are mapped to general language constructs, making it difficult to understand programs at a higher level. As in textual languages, we need to compose code blocks using language constructs in functions and modules to consider the workflow steps. Even though many frameworks support customizing blocks for wrapping a set of blocks, such customization usually needs a certain programming skill level.

Control flow diagrams, which use visual components to describe the process of handling, might be the most intuitive representation of program code because programs are composed of a set of control operations. It exactly corresponds to the control flow graph used in static code analysis [44]. Although the structure of the control flow diagrams is essentially the same as nested blocks, components are connected by arrow lines rather than snapped. Every component denotes the operation to perform, *i.e.*, how to handle the data in this step. Control flow diagrams are usually referred to as flowcharts in a narrow sense [24, 45] and are supported by many frameworks, such as GRAIL [46], DRAKON [47], and RAPTOR [48]. Generating programs from these diagrams is relatively easy because program code structure is directly reflected in the diagrams.

On the other hand, in data flow diagrams, we are concerned about variables instead of tasks, although they may also be considered a kind of flowchart. It means that programs are modeled as a set of operator nodes interconnected by data-carrying arcs [49]. Techniques are needed to translate the code written in non-dataflow languages to data flow diagrams, for example, the side effects resulting from assignments to a global variable [50]. Each component in the data flow diagrams represents a function, and each line connecting components indicates the flow of data [51]. Thinking and describing operations in the form of data is considered a better approach because spliced data flow diagrams can represent all needed information and indicate what crosses from one component to the next. Although the meaning of computation is usually defined as acts and every expression in a program is an operation, the subjects to address are eventually data. Proprietary commercial software, such as LabVIEW and Keysight VEE and the JavaScript visual framework Rete.js, can be classified into this category.

Modeling diagrams consider programs on a higher level than control flow diagrams and data flow diagrams. They are used to describe the design of software but not the computation. In other words, what we think in modeling diagrams is a whole picture of static system architecture rather than the program states at a specific timing. Several famous modeling and description languages, such as unified modeling language (UML) and specification and description language (SDL), were proposed for software development. They support a set of diagrams to cover different aspects of programs, and some of them might use control flow or data flow to describe the design, for example, the activity diagram in UML. These modeling diagrams are often encouraged in software development because they provide a higher-level abstraction on system behavior and provide a better understanding of software architecture. In contrast, generating code from these modeling diagrams is not as easy as the other three kinds of diagrams. Many research activities work on program generation from modeling diagrams, for example, generating code according to the state diagram [52, 53] or the sequence diagrams [54] in UML.

MasonPy is in the category of control flow diagrams and a flowchart in a narrow sense. Although there has been much discussion of the weakness of flowcharts [45], it has also been shown that structured flowcharts outperform pseudocode [55] and help novice programmers write programs [6]. The comparison is summarized in Table 5.

**Table 5. The classification of related work and MasonPy.**

| style | **nested blocks** | **control flow diagram** | **data flow diagram** | **modeling diagram** |
|---|---|---|---|---|
| component | construct-like | function-like | function-like | object-like |
| construction | bottom-up | bottom-up | bottom-up | top-down |
| examples | Logo Blocks, Scratch, Snap!, App Inventor, Blockly | GRAIL, DRAKON, RAPTOR, MasonPy* | LabVIEW, VEE, Rete.js | UML, SDL |

* Developing DSLs in MasonPy is top-down construction, while writing programs with the DSLs is bottom-up construction.

## 6.2 Component Set

We can also compare MasonPy with other frameworks in terms of the component set. Flowcharts come from the engineering domain and have a long history. The flowchart was first mentioned as a flow process chart by Frank and Lillian Gilbreth at the ASME (American Society of Mechanical Engineers) Annual Meeting in 1921. It later became a standard of the ASME in 1947 [56]. In computer programming, Goldstine and Von Neumann presented how to draw a flowchart for a given problem and write code based on them [57].

There are two common complaints about using flowcharts in software development [45, 58], but they are not the issues in separating the experimental concern with DSVLs. First, considering the mismatch between flowcharts and actual code. Programmers often face a situation in which it is difficult to maintain an accurate flowchart for their programs. When flowcharts are used as a blueprint, programmers need to update them to reflect the changes in a rapidly evolving software system. However, DSVL frameworks, such as MasonPy, provide an environment to automatically generate code by design, which means no overhead in maintaining the consistency between the code and design. Second, flowcharts tend to be too complicated to read. Programmers usually have different concerns for code to mix together and code to modify because of constraints. If the flowcharts are too specific and detailed, reading the code will be cumbersome and difficult. To address this issue, we suggest using a fully user-defined DSVL that has a minimal number of components to describe the workflow. In this case, the flowcharts for running experiments can represent only the experimental concern and remain simple.

The standards of flowcharts and their symbols, *i.e.*, components, were set by ANSI in approximately 1970 [59] and adopted by ISO in 1973 [60]. The current version was revised in 1985 and confirmed again in 2019 [61]. It supports many kinds of flowcharts, such as data flowcharts, program flowcharts, system flowcharts, program network charts, and system resources charts. If we ignore the symbols for connecting components and pages, at least eight basic symbols are supported in most flowcharts: terminal, process, decision, database, document, input/output, preparation, and predefined process. DRAKON can be considered an extended variant of ISO flowcharts. It defines 26 visual signs, including title, end, formal parameters, question, choice, case, and begin/end of the FOR loop [62]. These visual alphabets are named "DRAKON letters" and can be used to compose "DRAKON words." On the other hand, UML activity diagrams have a reduced set of components since they focus on the representation of activities. Standard components include Start/End, Activity, Decision, Note, Option Loop, Join, and Fork. MasonPy supports only a reduced set of flowcharts. It is similar to the activity diagrams without the support of parallel activities, *i.e.*, no Fork/Join node.

## 6.3 Program construction

MasonPy is different from other frameworks regarding how to construct programs. We can consider program construction in two forms: top-down modularization and bottom-up modularization [63]. Top-down modularization considers and specifies a global program code structure for a given task, *i.e.*, designing components at the top level first and implementing component details later. In contrast, bottom-up modularization means specifying common elements of a fine structure to construct a program, *i.e.*, implementing standard low-level components first and building programs with them. Most flowcharts in the broad sense, including nested blocks, control flow diagrams, and data flow diagrams, are bottom-up modularization. Users are asked to construct their programs based on a given set of components at a low level. On the other hand, modeling languages, such as UML and SDL, are top-down modularizations. Users can design with nodes and objects at a high level and then concretely implement them. MasonPy uses the concept of user-defined DSLs to divide the thinking of code modularization since DSLs can separate the roles of developers and users. Programmers who develop a tiny DSL on top of MasonPy design coarse-grained components to model workflows for experimental scenarios. It is similar to drawing elements in UML diagrams with top-down modularization. On the other hand, programmers who use the tiny DSL use the components provided by the tiny DSL to construct their experimental programs with bottom-up modularization. It is the same as how they use fine-grained components in other flowchart frameworks.

## 7. CONCLUSIONS AND FUTURE WORK

To conduct related experiments for scientific computing and engineering simulation, programmers need to manage program codes with a good strategy, which requires programming skills. However, domain experts might not be masters of programming. We proposed a simple and lightweight framework, MasonPy, to help domain experts design user-defined DSVLs for separating the experimental concern from other concerns. With a fully customized DSVL, implementation details in individual components can be hidden from the programmers who conduct the experiments. We demonstrated a tiny DSL named Hybrilog for wind turbine systems built on MasonPy to show the feasibility and discuss its benefits. People who are not programming experts can design and run experiments with Hybrilog. To clarify the position of MasonPy, we discussed related work and compared them from the viewpoint of diagram style, component set, and program construction. The discussion and comparison show that users can use MasonPy to quickly modularize experimental programs with a customized DSVL and quickly generate programs for different workflows. In the future, we plan to make the syntax independent from Python syntax and add the support of parallel computing representation.

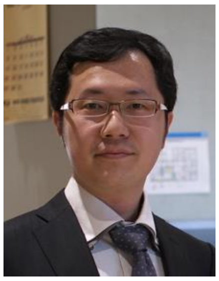
## REFERENCES

1. G. Fischer, K. Nakakoji, and Y. Ye, "Metadesign: Guidelines for supporting domain experts in software development," *IEEE Software*, Vol. 26, 2009, pp. 37-44.
2. P. Hudak, "Building domain-specific embedded languages," *ACM Computing Surveys*, Vol. 28, 1996, pp. 196-es.
3. M. M. Burnett, "Visual object-oriented programming," *ACM SIGPLAN OOPS Messenger*, Vol. 5, 1993, pp. 127-129.
4. B. A. Myers, "Taxonomies of visual programming and program visualization," *Journal of Visual Languages & Computing*, Vol. 1, 1990, pp. 97-123.
5. C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys*, Vol. 37, 2005, pp. 83-137.

6. K. Charntaweekhun and S. Wangsiripitak, "Visual programming using flowchart," in *Proceedings of International Symposium on Communications and Information Technologies*, 2006, pp. 1062-1065.
7. M. Resnick *et al.*, "Scratch: programming for all," *Communications of the ACM*, Vol. 52, 2009, pp. 60-67.
8. N. Fraser, "Ten things we've learned from blockly," in *Proceedings of IEEE Blocks and Beyond Workshop*, 2015, pp. 49-50.
9. H. Khalajzadeh, M. Abdelrazek, J. Grundy, J. Hosking, and Q. He, "BiDaML: A suite of visual languages for supporting end-user data analytics," in *Proceedings of IEEE International Congress on Big Data*, 2019, pp. 93-97.
10. J. E. Rivera, F. Durán, and A. Vallecillo, "On the behavioral semantics of real-time domain specific visual languages," in *Proceedings of International Workshop on Rewriting Logic and its Application*, 2010, pp. 174-190.
11. E. Guerra, J. de Lara, A. Malizia, and P. Díaz, "Supporting user-oriented analysis for multi-view domain-specific visual languages," *Information and Software Technology*, Vol. 51, 2009, pp. 769-784.
12. J. C. Grundy, J. G. Hosking, R. W. Amor, W. B. Mugridge, and Y. Li, "Domain specific visual languages for specifying and generating data mapping systems," *Journal of Visual Languages & Computing*, Vol. 15, 2004, pp. 243-263.
13. M. Almorsy, J. Grundy, R. Sadus, W. van Straten, D. G. Barnes, and O. Kaluza, "A suite of domain-specific visual languages for scientific software application modelling, ," in *Proceedings of IEEE Symposium on Visual Languages and Human Centric Computing*, 2013, pp. 91-94.
14. J. Sprinkle and G. Karsai, "A domain-specific visual language for domain model evolution," *Journal of Visual Languages & Computing*, Vol. 15, 2004, pp. 291-307.
15. G. Guizzardi, L. F. Pires, and M. J. Van Sinderen, "On the role of domain ontologies in the design of domain-specific visual modeling languages," in *Proceedings of the 2nd Workshop on Domain-Specific Visual Languages*, 2002, pp. 25-38.
16. J. C. Grundy, J. Hosking, K. N. Li, N. M. Ali, J. Huh, and R. L. Li, "Generating Domain-specific visual language tools from abstract visual specifications," *IEEE Transactions on Software Engineering*, Vol. 39, 2013, pp. 487-515.
17. N. Zhu, J. Grundy, J. Hosking, N. Liu, S. Cao, and A. Mehra, "Pounamu: A meta-tool for exploratory domain-specific visual language tool development," *Journal of Systems and Software*, Vol. 80, 2007, pp. 1390-1407.
18. E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-science: An overview of workflow system features and capabilities," *Future Generation Computer Systems*, Vol. 25, 2009, pp. 528-540.
19. O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press Ltd., London, 1972.
20. S. Apel, D. Batory, C. Kästner, and G. Saake, "Software product lines," in *Feature-Oriented Software Product Lines*, Springer, 2013, pp. 3-15.
21. C. Prehofer, "Feature-oriented programming: A fresh look at objects," in *Proceedings of European Conference on Object-Oriented Programming*, 1997, pp. 419-443.
22. E. W. Dijkstra, "Dijkstra archive: On the role of scientific thought (EWD447)," http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html, 2020.

23. T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 1-3.
24. I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *ACM SIGPLAN Notices*, Vol. 8, 1973, pp. 12-26.
25. K. M. Rich, C. Strickland, T. A. Binkowski, C. Moran, and D. Franklin, "K-8 learning trajectories derived from research literature: sequence, repetition, conditionals," *ACM Inroads*, Vol. 9, 2018, pp. 46-55.
26. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *Proceedings of European Conference on Object-Oriented Programming*, 1993, pp. 406-431.
27. J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, Vol. 9, 2007, pp. 90-95.
28. G. Xu, F. Liu, J. Hu, and T. Bi, "Coordination of wind turbines and synchronous generators for system frequency control," *Renewable Energy*, Vol. 129, 2018, pp. 225-236.
29. V. Petrović and C. L. Bottasso, "Wind turbine envelope protection control over the full wind speed range," *Renewable Energy*, 2017, Vol. 111, pp. 836-848.
30. K. T. Magar, M. J. Balas, and S. A. Frost, "Direct adaptive torque control for maximizing the power captured by wind turbine in partial loading condition," *Wind Energy*, Vol. 19, 2016, pp. 911-922.
31. J. Bystryk and P. E. Sullivan, "Small wind turbine power control in intermittent wind gusts," *Journal of Wind Engineering and Industrial Aerodynamics*, Vol. 99, 2011, pp. 624-637.
32. E. Koutroulis and K. Kalaitzakis, "Design of a maximum power tracking system for wind-energy-conversion applications," *IEEE Transactions on Industrial Electronics*, Vol. 53, 2006, pp. 486-494.
33. J. H. Kao, Y. Zhuang, and P. Y. Tseng, "Proposing hybrid controlling logic (HCL) for the wind turbine system with verification by the DSL framework," *Electric Power Systems Research*, Vol. 187, 2020, p. 106280.
34. M. Sulír, M. Bačíková, S. Chodarev, and J. Porubän, "Visual augmentation of source code editors: A systematic mapping study," *Journal of Visual Languages & Computing*, Vol. 49, 2018, pp. 46-59.
35. M. Boshernitsan and M. S. Downes, "Visual programming languages: A survey," Computer Science Division (EECS), University of California, 2004.
36. G.-C. Roman and K. C. Cox, "A taxonomy of program visualization systems," *Computer*, Vol. 26, 1993, pp. 11-24.
37. B. A. Price, R. M. Baecker, and I. S. Small, "A principled taxonomy of software visualization," *Journal of Visual Languages & Computing*, Vol. 4, 1993, pp. 211-266.
38. D. D. Hils, "Visual languages and computing survey: Data flow visual program-ming languages," *Journal of Visual Languages & Computing*, Vol. 3, 1992, pp. 69-101.
39. B. A. Myers, "Visual programming, programming by example, and program visualization: a taxonomy," *ACM Sigchi Bulletin*, Vol. 17, 1986, pp. 59-66.
40. M. Tempel, "Blocks programming," *CSTA Voice*, Vol. 9, 2013, pp. 3-4.
41. B. Harvey and J. Mönig, "Bringing "no ceiling" to scratch: Can one language serve kids and computer scientists," in *Proceedings of Constructionism*, 2010, pp. 1-10.

42. B. Magnuson, "Building blocks for mobile games: a multiplayer framework for App inventor for Android," Massachusetts Institute of Technology, 2010.
43. S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books, Inc., 1980.
44. F. E. Allen, "Control flow analysis," *SIGPLAN Notices*, Vol. 5, 1970, pp. 1-19.
45. N. Ensmenger, "The multiple meanings of a flowchart," *Information & Culture*, Vol. 51, 2016, pp. 321-351.
46. T. O. Ellis, J. F. Heafner, and W. L. Sibley, "The GRAIL Project: An experiment in man-machine communications," Rand Corp Santa Monica CA, 1969.
47. V. Parondzhanov, "Visual syntax of the DRAGON language," *Programming and Computer Software*, Vol. 21, 1995, pp. 142-153.
48. M. C. Carlisle, T. A. Wilson, J. W. Humphries, and S. M. Hadfield, "RAPTOR: introducing programming to non-majors with flowcharts," *Journal of Computing Sciences in Colleges*, Vol. 19, 2004, pp. 52-60.
49. P. G. Whiting and R. S. V. Pascoe, "A history of data-flow languages," *IEEE Annals of the History of Computing*, Vol. 16, 1994, pp. 38-59.
50. J. B. Dennis, "Data flow supercomputers," *IEEE Computer*, Vol. 13, 1980, pp. 48-56.
51. A. L. Davis and R. M. Keller, "Data flow program graphs," *Computer*, Vol. 15, 1982, pp. 26-41.
52. E. Sunitha and P. Samuel, "Automatic code generation from UML state chart diagrams," *IEEE Access*, Vol. 7, 2019, pp. 8591-8608.
53. I. A. Niaz and J. Tanaka, "Code generation from UML statecharts, ," in *Proceedings of the 7th IASTED International Conference on Software Engineering and Application*, 2003, pp. 315-321.
54. D. Kundu, D. Samanta, and R. Mall, "Automatic code generation from unified modelling language sequence diagrams," *IET Software*, Vol. 7, 2013, pp. 12-28.
55. D. A. Scanlan, "Structured flowcharts outperform pseudocode: an experimental comparison," *IEEE Software*, Vol. 6, 1989, pp. 28-36.
56. B. B. Graham, "Detail process charting: speaking the language of process," *John Wiley & Sons*, 2004.
57. H. H. Goldstine and J. von Neumann, "Planning and coding of problems for an electronic computing instrument," Institute for Advanced Study, Princeton, NJ, 1947.
58. J. M. Yohe, "An overview of programming practices," *ACM Computing Surveys*, Vol. 6, 1974, pp. 221-245.
59. N. Chapin, "Flowcharting with the ANSI standard: A tutorial," *ACM Computing Surveys*, Vol. 2, 1970, pp. 119-146.
60. ISO, "ISO 1028:1973 – Information processing – Flowchart symbols," https://www.iso.org/standard/5500.html.
61. ISO, "ISO 5807:1985 – Information processing – Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts," https://www.iso.org/standard/11955.html.
62. V. P. Ivannikov, "Visual syntax of the DRAKON language," *Programming and Computer Software*, Vol. 21, 1995, pp. 142-153.
63. W. Wulf, C. Geschke, D. Wile, and J. Apperson, "Reflections on a systems programming language," *ACM SIGPLAN Notices*, Vol. 6, 1971, pp. 42-49.



**YungYu Zhuang (莊永裕)** received the B.S. and M.S. degrees in Mechanical Engineering and Computer Science from National Taiwan University in 2002 and 2004, respectively, and the Ph.D. degree in Information Science and Technology from the University of Tokyo, Japan, in 2014. From 2014 to 2016, he was a Project Assistant Professor with the University of Tokyo. He is currently an Assistant Professor with the Department of Computer Science and Information Engineering, National Central University, Taiwan. He was a Research Assistant with the Central Weather Bureau, Taiwan, from 2004 to 2006, and worked as a Software Engineer in the industry from 2006 to 2011. His research interests include programming language design, software engineering, high-performance computing, machine learning, and programming education.

**Jui-Hsiang Kao (高瑞祥)** was born in 1975, and received the Ph.D. degree in Department of Systems Engineering and Naval Architecture from National Taiwan Ocean University in June 2004. He worked as the design manager in Hung Shen Propeller Company from 2004 to 2010. From 2010 to 2014, he worked in the Delta Electronics Company as the deputy manager. In 2014, he was invited to be an Assistant Professor in the Department of Systems Engineering and Naval Architecture, National Taiwan Ocean University, Keelung, Taiwan. His research focuses on propeller acoustic and vibration, marine propulsion system, and wind turbine design.

**Kuan-Shang Liu (劉冠尚)** graduated from National Central University, where he majored in Computer Science and Information Engineering. In the college period, he learned lots of knowledge about electrical engineering, programming, and software engineering. This experience has given him a deeper understanding of programming and also sparked my interest in the field.

**Chia-Yu Lin (林佳育)** graduated with a Master degree in Computer Science and Information Engineering from National Central University in Taiwan. He currently works for Pro Brand Technology in Taiwan. He’s engrossed in software engineering and the improvement of user experience.