S. Chien, S. Choo, M. A. Schnabel, W. Nakapan, M. J. Kim, S. Roudavski (eds.), *Living Systems and Micro-Utopias: Towards Continuous Designing, Proceedings of the 21st International Conference of the Association for Computer-Aided Architectural Design Research in Asia CAADRIA 2016,* 157–166.

MÖBIUS

A parametric modeller for the web

PATRICK JANSSEN, RUIZE LI and AKSHATA MOHANTY
National University of Singapore, Singapore
patrick@janssen.name, {liruizenus, akshatamohanty}@gmail.com

Abstract. For complex parametric modelling tasks, systems that use textual programming languages (TPLs) currently have clear advantages over visual programming languages (VPLs) systems. Their support for a rich variety of programming mechanisms means that the complexity of the program can remain commensurate with the complexity of the modelling task. A prototype parametric modelling system called Möbius is presented that aims to overcome the limitations of existing VPL systems. The proposed system integrates associative and imperative programming styles and supports iterative looping and higher order functions. In order to demonstrate the versatility of the Möbius, a modelling task is presented that requires the model to be modified.

Keywords. Parametric procedural modelling; generative design; visual programming; human-computer interaction.

1. Introduction

Parametric modelling is a form of programming. Two types of programming languages are Textual Programming Languages (TPL systems) and Visual Programming Languages (VPL systems) (Myers, 1990). Historically, the debate surrounding TPL systems versus VPL systems has not been conclusive, with researchers arguing in favour of both approaches (Menzies 2002). TPL systems are continuing to evolve rapidly and are becoming more user-friendly due to simplifications in syntax and improvements in the IDEs (Leitão et al, 2012). The evolution of VPL systems on the other hand seems to be slower.

In the design domain, a number of parametric modelling systems use VPLs for generating models. However, these systems are known to have

poor scalability (Stouffs and Chang, 2010; Park and Holt, 2010; Janssen and Chen, 2011; Chok, 2011; Leitão et al, 2012; Janssen, 2014) due to their weak support for fundamental programming mechanisms. Two important mechanisms are iterative loops and higher-order functions.

Iteration refers to mechanisms for repeating a sub-procedure, and includes for loops, while loops and recursive function calls. VPL systems for parametric modelling typically use a graph-based programming approach consisting of nodes and wires, and as a result do not support iteration very well. With dataflow and procedural modelling systems, it is possible to create models that incorporate iterative procedures (Janssen and Stouffs, 2015). McNeel Grasshopper is an example of a dataflow system, while SideFx Houdini is an example of a procedural system. However, creating sophisticated types of iterations in these types of systems (such as nested loops) remains prohibitively difficult (Janssen and Stouffs, 2015).

Higher-order functions refers to any functions that take another function as an argument or returns a function as a result. In parametric modelling, such functions can be very useful as they allow components in the design to be represented and manipulated using functions (Leitão, 2014). Most VPL systems do not support higher-order functions. Two recent exceptions are Autodesk Dynamo and Autodesk 3DS MCG. However, the dataflow interface makes it difficult to leverage the full power of higher-order functions. Another powerful parametric modelling system that specifically supports higher-order functions is Rosetta (Lopes and Leitão, 2011), but this is actually a textual programming system.

Another important aspect that needs to be considered is the learning curve. The context for this research is the Department of Architecture at the National University of Singapore. During the last seven years, electives have been taught teaching both Grasshopper and Houdini. Typically, students need to both learn the tools and apply them in an architectural design project within a single 12 week semester. For both VPL systems, this proved to be very challenging. In the electives teaching Grasshopper, students could make simple models quite quickly, but once more complex parametric tasks were attempted, the learning curve quickly steepened. In the electives teaching Houdini, students found the initial learning curve to be quite steep, but then discovered that some quite complex tasks were easily achievable. Nevertheless, with further increases in parametric complexity, the learning curve again steepened. Thus, although the learning curve for TPL systems is typically higher than VPL systems, the experiences with Grasshopper and Houdini show that learning these VPL systems to a level required for complex parametric modelling is also very time consuming.

A related issue with current VPL systems is that they do not allow for a graceful progression towards learning a TPL (Aish, 2013). Given that TPLs may always have certain advantages over VPLs, it would be desirable if the use of a VPL system would allow users to gradually start to learn TPL concepts. This would allow expert users of the VPL system to transition over to a TPL without having to restart their learning process from scratch. However, the concepts used in most graph-based VPL systems have limited relevance when learning textual programming.

For complex parametric modelling tasks, TPL systems therefore currently have clear advantages over VPL systems. Their support for a rich variety of programming mechanisms means that the complexity of the program can remain commensurate with the complexity of the modelling task. However, it may be possible to overcome many of the shortcomings of existing parametric modelling VPL systems. This research has developed a prototype VPL system called Möbius for creating complex procedural models. One of the key objectives is to create a system that students can become proficient in within one 12 week semester. This proficiency should go beyond simple 'toy' models, and should allow students to build parametric models that incorporate complex nested iterative loops and higher-order functions.

Section 2 will describe the key features of Möbius, and Section 3 will present an experiment that demonstrates the versatility of Möbius. Finally, section 4 discusses future work.

2. System overview

Möbius is a web-application that runs in the browser on the client side. As a result, Möbius is platform independent and there is no need to install software. Figure 1 shows the Möbius user interface.

Möbius will be introduced using the Killian Roof modelling task as an example, based on a tutorial for Generative Components developed by Axel Kilian in 2005. The task consists of building a simple parametric roof with a diagrid structure (Woodbury at al, 2007). The ridge of the roof remains at a constant height while the base of the structure varies in height in response to an undulating ground surface. An instance if the Killian Roof is shown in Figure 1.

Möbius integrates imperative and associative programming styles. The imperative style of programming is supported through blocks-based programming, where the user constructs procedures by creating sequences of code blocks (Resnick et al, 2014). The associative style of programming is supported through dataflow programming, where the user constructs net-

works consisting of nodes and wires. Each node has an imperative procedure that is defined using the blocks-based approach.

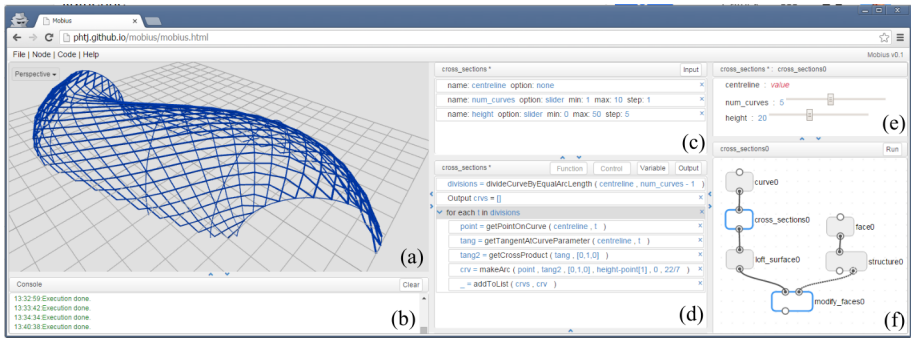


Figure 1. The Möbius interface, showing a model of the Killian roof.

2.1. IMPERATIVE PROGRAMMING

Nodes have types and instances. A node type is defined by specifying a procedure with a set of input and output ports. Each node type can have multiple node instances, all of which will have the same procedure with the same input and output ports. The input and output data for each instance will differ.

Node procedures are defined with the imperative programming style, using panes (c) and (d) in Figure 1. Pane (c) is used to define the input ports for the node type. Pane (d) is used to define the procedure and output ports for the node type using the blocks-based programming approach. In the example in Figure 1, the procedure and inputs for the *cross_section* node type are shown.

In the procedure pane, Möbius provides buttons to create five types of code blocks. Variable blocks define new variables. Function blocks call a specific function, selected from a drop-down list. Looping blocks create loops such as *for-each* loops and *while* loops. Conditional blocks create *if-then* conditions. Finally, output blocks create output variables, which result in node output ports being added to the node.

Once a code block has been created, the user can customize the block by clicking on the links and typing in values. For example, in Figure 1(d), the first code block is a function block that calls the function *divideCurveByEqualArcLength*. The user has created a variable called *divisions* to store the result, and has edited the function arguments.

When defining variable values or function arguments, the user can enter single-line Javascript expressions. These expressions avoid the need to have code-blocks for simple operations such as arithmetic operations. Expressions

can also be used to navigate topological data structures that link geometric entities such as vertices, edges, and faces. This allows lower level entities to be extracted from higher level entities using simple expressions that navigate through the data structure. For example, the expression *'mesh.faces[3].vertices[0].z'* will return the z coordinate of the first point of the of the fourth face in the mesh. This ability to extract specific pieces of data from deep within geometric objects significantly simplifies the procedures.

2.2. ASSOCIATIVE PROGRAMMING

Node networks are defined by placing and wiring together a set of node instances. At the network level, these node instances are treated as black boxes.

The node network is defined with the associative programming style, using panes (e) and (f) in Figure 1. Pane (e) is the node parameters pane, and is used to set certain input values for the node instance selected in pane (f). Pane (f) is the network pane, and is used to define the dataflow network. The inputs to a node instance can either be defined by connecting a wire to the input port (in which case the parameter will be hidden), or by entering a specific parameter value through the user interface widget. In the example in Figure 1, the *cross_section0* node instance is selected, and the parameters being shown are therefore for this node.

When the dataflow network is run, the network is converted into a single Javascript program that is then executed in one go. This represents a synchronous type of execution, where the nodes are first topologically sorted and then executed in a fixed sequence. This differs from most other dataflow system, where nodes are executed asynchronously in response to changes to their inputs.

The generation of a textual program from the dataflow network has the advantage that it allows a wide range of more advanced programming mechanisms to be used. Two mechanism will be described in more detail: iterative loops and higher-order functions.

2.3. ITERATIVE LOOPS AND HIGHER ORDER FUNCTIONS

Iterative loops can be created at both the procedure level and the dataflow level. At the procedure level, users can easily define nested loops using the blocks-based programming style. However, it may also be necessary to create loops at the dataflow level. For example, when creating performance-based models, dataflow networks may need to be repeatedly executed in an iterative optimization process. Creating iterative loops at the dataflow level

is possible due to the fact that all nodes are also callable as functions. This means that a procedure inside one node can include function blocks that call other nodes.

Higher order functions are supported by an additional output port which returns a reference to the function that wraps the procedure for that node. When this output is wired into another node, it results in a dashed wire, which indicates to the user that it contains a reference to a function rather than actual data. These wires can be used to provide a function as an input to another downstream node. That node can then have a procedure that executes that function.

The dataflow network for the Killian Roof example, as shown in Figure 1(f), includes a higher-order function to create the diagonal struts. This function takes a single mesh face as an input and generates two diagonal struts. The *modify_faces0* node uses this function to replace each mesh face with diagonal struts.

3. Experiment

In order to demonstrate the versatility of the Möbius, an example developed by Leitão et al (2012) is used. The modelling task consists of two phases: (1) implementation of a parametric model to create a structure made of cylindrical spirals; and (2) the modification of the original model to create sinusoidal variations of the cylindrical spirals. The aim of the two-phase experiment was to investigate how easily users were able to adapt the phase 1 model to the new requirements in phase 2.

In the experiments conducted by Leitão et al (2012), a user-based study was conducted to compare a TPL system with a VPL system. The TPL system was Rosetta (Lopes and Leitão, 2011) (using VisualScheme as the textual language) and the VPL system was Grasshopper. Six designers with varying levels of expertise were asked to complete the modelling tasks, and the resulting models and modelling times were compared. The results showed that the models created with Rosetta were easier to understand and significantly more adaptable than those created with Grasshopper.

The aim of revisiting this example is to demonstrate that while it is true that Rosetta models are more adaptable than Grasshopper models, this cannot necessarily be generalizable to all TPL and VPL systems. As discussed in the introduction, Grasshopper suffers from restrictive mechanisms that make it difficult or impossible to tackle complex parametric modelling tasks without resorting to scripting. Möbius is much less restrictive and as a result Möbius models are more easily adapted.

3.1. PHASE 1: CYLINDRICAL SPIRAL TOWERS

In the first phase, the task is to create a parametric model capable of generating the structures shown in Figure 2a.

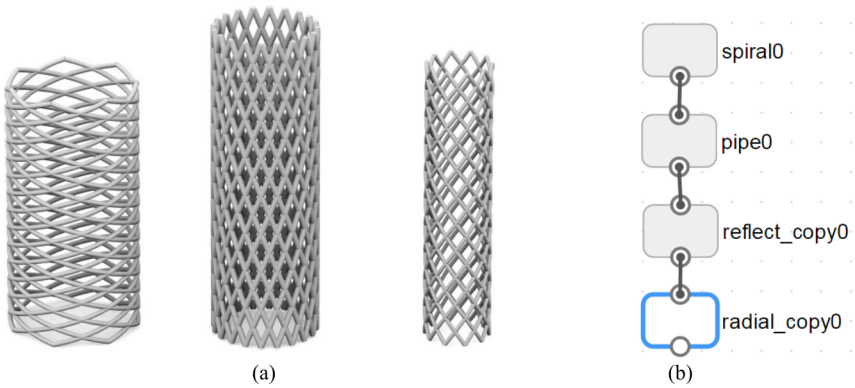


Figure 2. (a) Phase 1 cylindrical spiral towers (taken with permission from Leitão et al, 2012). (b) Möbius dataflow network for phase 1.

In Möbius, this can be achieved in many different ways. In this example, we implement what might be considered as the most straightforward approach within Möbius.

The first decision the user needs to make is how to apply associative and imperative programming styles. This comes down to the question of how many nodes to create. In general, it is possible to implement everything in one complex node, or to implement many simple nodes. The most desirable approach is to create a small number of modular reusable nodes.

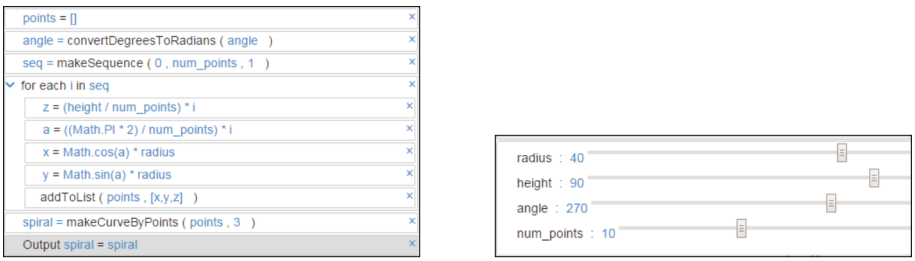


Figure 3. (a) Procedure for the spiral node. (b) Parameters for the spiral node.

In this case, four nodes are created, as shown in Figure 2b. All four nodes are highly generic and reusable. The *spiral0* node creates spirals, the *pipe0* node creates pipe surfaces from curves, the *reflect_copy0* node creates a copy of objects by reflection, and the *radial_copy0* node creates radial cop-

ies of objects. The procedure for the spiral is shown in Figure 3a. Figure 3b shows the parameter interface for the same node.

3.2. PHASE 2: SINUSOIDAL CYLINDRICAL SPIRAL TOWERS

In the second phase of the experiment, the parametric model needs to be modified in order to be able to create the structures shown in Figure 4a. Only the spiral node needs to be modified so that the radius variable in the procedure (see Figure 3a) changes with height. The easiest way to achieve this is to use a higher-order function.

The modified dataflow network is shown in Figure 4b. The *radius_func0* node takes a height factor as an input and outputs the radius. The procedure for this node is shown in Figure 5a. The calculation combines a linear function and a Sine function. The parameters for the linear function are the radius at the top and bottom of the structure. The parameters for the Sine function are the frequency and amplitude of the sinusoidal wave.

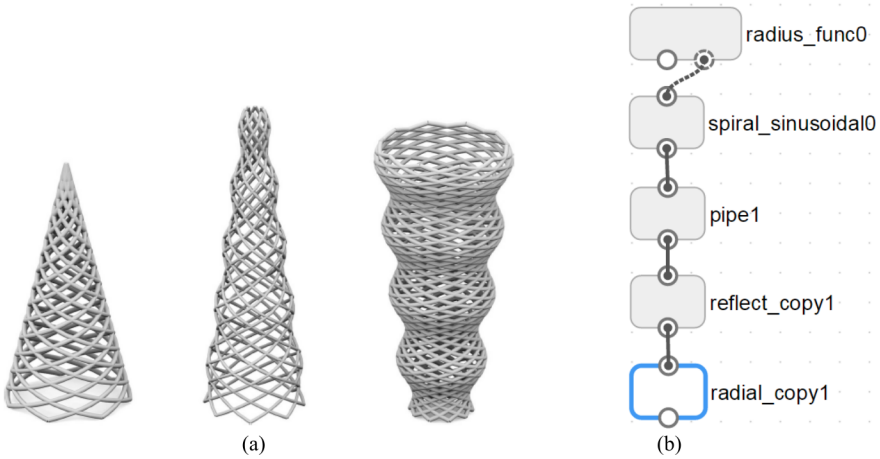
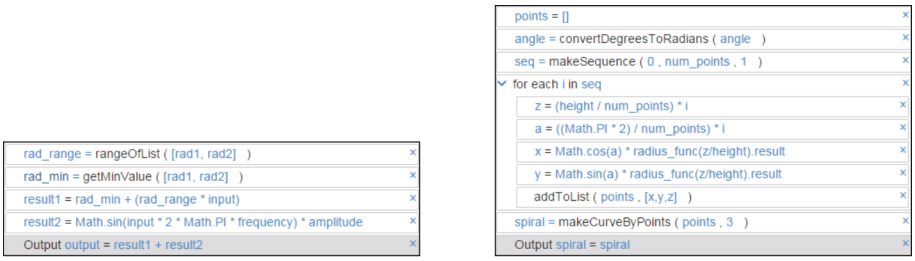


Figure 4. Phase 2 sinusoidal cylindrical spiral towers (taken with permission from Leitão et al, 2012). (b) Möbius dataflow network for phase 2.

The modified *spiral_sinusoidal0* node is almost the same as the original spiral node. The procedure is shown in Figure 5b. The only difference is that the radius is replaced by the radius function.



(a) (b)

Figure 5. (a) Procedure for the 'radius_func0' node. (b) Procedure for the modified 'spiral_sinusoidal0' node.

3.3. EVALUATION

The example demonstrates that the complexity of the dataflow networks and procedures created using Möbius remain commensurate with the complexity of the modelling task. As a result, the model is relatively easy to modify and adapt. For users that have learnt certain basic concepts of higher order functions, the transition from phase 1 to phase 2 can be implemented in a way that is natural and intuitive.

One of the key objectives of developing Möbius is to have a system that students can become proficient in within a 12 week semester. At this stage of the research, no user studies have been conducted yet and this objective can therefore not yet be evaluated. The two examples above were used as a preliminary test of the Möbius VPL. In the future, Möbius will be used as a tool for teaching parametric design. User studies will then be conducted for a variety of modelling tasks, allowing further data and feedback to be gathered.

An additional benefit of using Möbius is the fact that the creation of procedures using blocks teaches users the basic concepts of textual programming. As users become more experienced, they can then gradually transition over to textual programming.

4. Future work

Möbius is under active development, and various additional components and user interface enhancements are being worked on. Additional viewers and function modules are being developed for BIM and GIS. Support for debugging parametric models is being enhanced with more visual feedback. In addition to these ongoing development efforts, there are also two more fundamental areas that will be tackled in the future.

First, the execution of the code generated from the dataflow networks will be optimised for performance using memorisation techniques. Traditional dataflow systems have the advantage that they only need to re-execute

nodes whose inputs have changed. However, in Möbius this is not possible due to the fact that code is generated for the whole dataflow network and then executed as one standalone script. Memorisation is an optimization technique that stores the results of function calls and returns the cached result when the same inputs occur again.

Second, the current Javascript geometry libraries will be replaced by more complete geometry engines supporting a wider range of geometric functions. Möbius currently uses the Verbs geometry library developed by Peter Boyer (http://verbnurbs.com). Future geometry engines being explore include OpenCascade (http://www.opencascade.com) and CGAL (http://www.cgal.org).

Acknowledgements

This research was supported by the Singapore Ministry of Education under an AcRF Tier 1 grant (R-295-000-104-112).

References

Aish, R.: 2013, DesignScript: Scalable Tools for Design Computation, *Proceedings of the 31st eCAADe Conference*, Delft, The Netherlands, 87–95

Chok, K.: 2011 Progressive Spheres of Innovation: Efficiency, communication and collaboration, *Proceedings of the 31st ACADIA Conference*, Banff, Alberta, Canada, 234–241.

Janssen, P.: 2014, Visual Dataflow Modelling: Some thoughts on complexity, *Proceedings of the 32nd eCAADe Conference*, Newcastle, UK, 547–556.

Janssen, P. and Chen, K.W.: 2011, Visual Dataflow Modelling: A Comparison of Three Systems, *Proceedings of CAAD Futures 2011*, Liege, Belgium, 801–816.

Janssen, P. and Stouffs, R.: 2015, Types of Parametric Modelling, *Proceedings of the 20th CAADRIA Conference*, Daegu, Republic of Korea, 157–166.

Leitão, A.: 2014, Improving Generative Design by Combining Abstract Geometry and Higher-Order Programming, *Proceedings of the 19th CAADRIA Conference*, Kyoto, Japan, 575–584.

Leitão, A., Santos, L., and Lopes, J.: 2012, Programming Languages For Generative Design: A Comparative Study. *International Journal of Architectural Computing*, **10**(1), 139–62.

Lopes, J. and Leitão, A.: 2011, Portable Generative Design for CAD Applications, *Proceedings of the 31st ACADIA Conference*, Banff, Alberta, Canada, 196–203.

Myers, B. A.: 1990, Taxonomies of Visual Programming and Program Visualization, *Journal of Visual Languages and Computing*, **1**(1), 97–123.

Park, K. and Holt, N.: 2010, Parametric Design Process of a Complex Building In Practice Using Programmed Code As Master Model, *International Journal of Architectural Computing*, **8**(3), 359–376.

Resnick, M., Silverman, B., Kafai, Y., Maloney, J. and Monroy-Hernández, A.: 2009, Scratch: programming for all, *Communications of the ACM*, **52**, 60

Stouffs, R. and Chang,W.-T.: 2010, Representational programming for design analysis, *Proceedings of the International Conference on Computing in Civil and Building Engineering*, Nottingham, UK, 351–359.

Woodbury, R.: 2010, *Elements of Parametric Design*, Routledge.