# Visual Dataflow Modelling

## *Some thoughts on complexity*

*Patrick Janssen*
*National University of Singapore*
*www.patrick.janssen.name*
*patrick@janssen.name*

*When visual programming is applied to design, it results in a parametric modelling approach that we refer to as Visual Dataflow Modelling (VDM). This paper reviews existing VDM environments and identifies key issues that result in dataflow graphs with high complexity. In order to explore how these issues could be overcome, a new VDM environment is being developed, called Vidamo. For any given parametric modelling task, Vidamo aims to support a style of VDM where the complexity of the dataflow graph remains commensurate with the complexity of the task. An initial experiment is conducted in which the existing environments and the proposed Vidamo environment are used to complete the same parametric modelling task. The results indicate that the dataflow graphs created using Vidamo are much smaller and less complex.*

**Keywords:** *parametric and generative modelling, visual dataflow modelling, cyclomatic complexity*

## INTRODUCTION

Visual programming languages enable users to create computer programs by manipulating graphical elements rather than by entering text. In the design domain, visual programming can be used as a parametric modelling approach that we refer to as Visual Dataflow Modelling (VDM) (Janssen and Chen 2011). The appeal of VDM environments lies in the fact that complex form generating procedures can be created without having to learn programming. However, a well known problem with these environments is that the complexity of the dataflow graphs can quickly become unmanageable.

Analysing dataflow graphs created using existing VDM environments, three key issues are identified: too many links, too many nodes, and confusing iteration. In general, it would seem that it should be possible to create typical parametric models using far fewer nodes linked together in more linear ways using iteration that is easier to understand. In order to address these issues, a new more scalable VDM environment is being developed, called Vidamo, supporting a hybrid style of visual programming that allows larger and more complex types of parametric models to be created. At this stage, this environment is in the early stages of conceptualization. This paper discusses some of the early design decisions.

The paper first gives an overview of four commonly used existing VDM environments. The proposed VDM environment is then described, high-

lighting how limitations of the existing environments are overcome. Finally, the conclusions summarises the results and indicates future research directions.

## OVERVIEW OF VDM ENVIRONMENTS

Four commonly used VDM environments are analysed: McNeel Grasshopper 0.9 [1], Autodesk Dynamo 0.7 [2], Bentley Generative Components v8 [3], and Sidefx Houdini 13 [4]. Other design environments that incorporate VDM include CityEngine by Esri [5], 3DVia Studio by Dassault [6], and the Sverchok plugin for Blender [7].

The four VDM environments all make use of dataflow graphs consisting of nodes and links. Nodes represent some type of operation and links represent the flow of data between these operations. Nodes have input ports and output ports and can have parameters that affect their behaviour. (In some environments, parameters are treated the same way as inputs, while in some these are treated differently.)

One important restriction that has been imposed is that multi-line scripting is disallowed, as this is seen to be challenging for many designers (see for example Senske 2014). The use of single-line expressions is however considered to be acceptable since it does not require users to have any advanced knowledge of scripting or programming.

### *Modelling task*

In order to be able to analyse the types of dataflow graphs generated by these four environments, all of them were used to complete a simple parametric modelling task. The chosen task is the Kilian Roof, 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. The task consists of a linear sequence of modelling steps, loosely defined as follows:

1. Centreline Coordinate Systems: Create a set of coordinate systems oriented along the centreline. The centreline consists of a spline assumed to be lying on an undulating ground surface. The coordinate systems are arrayed along the centreline, orientated perpendicular to the centreline and the global z-axis.

2. Hoop Sections: Use the coordinate systems to construct a set of hoop sections, consisting of spline curves orientated perpendicular to the centreline. The hoop sections are scaled so as to ensure that the roof ridge remains at a constant height.

3. Roof Surface: Use the hoop sections to create a discretised roof surface. The hoop sections are first lofted to create spline surface, and the spline surface is then discretised into a grid of four sided components.

4. Roof Modules: Use the roof components to create a set of roof modules consisting of X frames. The X frames connect diagonal corners of each component, with the frame diameter being proportional to the length. When all components are replaced by these frames, a diagrid structure results.

In the following sections, each VDM environment is analysed. First, three key aspects of each environment are described: the way nodes are used; the way links are used; and the way iteration is handled. These aspects strongly influence the types of dataflow graphs that are generated. A brief description is then given of how each VDM environment was used to build the Kilian roof.

### *Grasshopper*

Nodes may process either geometric data or non-geometric data. For geometric data, nodes are provided to create, modify and delete geometry. For non-geometric data, nodes are provided for various operations, including logical, scalar, and vector-based operations. Many of the nodes tend to perform fairly simple types of operations.

Links are created by the user connecting nodes in the graph. Both geometric data and non-geometric data is represented in the same way, as trees. A tree is a sorted collection of lists, where each list is identified by a path. A specific path can only occur once in a tree. When two trees are merged, lists with identical paths are appended to each other.

Iteration results from nodes iterating over lists in trees using a data matching algorithm. This algorithm looks at the inputs, and selects the data tree with the longest path to be the 'master' tree. The algorithm then iterates over the lists in this master tree and generates an output tree with matching paths. If the master tree has fewer lists than some of the other inputs, then the algorithm automatically generates the missing input data by duplicating the last list in the master tree. (This data matching algorithm is new since version 0.9)

For the modelling task, the coordinate systems were generated as frames along the centreline curve and a hoop section was copied onto each frame. A list of origin points was extracted from the frames and used to calculate a list of scale factors. These were then used to scale the hoop sections. The hoops were lofted to create a surface, which was then discretised by dividing the surface into a set of patches. The corner points of the patches were then extracted and used to generate two lists of corner point pairs. These lists were then used to create the lists of diagonal X lines, which were then piped, thereby creating the final piped structure.

### *Dynamo*

Nodes are similar to Grasshopper. Two additional types of nodes are provided: code block nodes and custom nodes. Code blocks allow users to write short expressions inside a node to perform simple calculations. Custom nodes allow users to define their own custom operations that can be saved as a separate file and then used within parametric models in the same way as the built-in nodes.

Links are also similar to Grasshopper, and are created by the user connecting nodes in the graph. Both geometric data and non-geometric data is represented using nested lists.

Iteration results from nodes iterating over data items in nested lists. Users are given the option to select from three 'lacing' patterns: 'shortest', 'longest', and 'cross product'. In addition, Dynamo supports recursion using custom nodes, thereby allowing for more complex types of iteration without requiring any scripting.

For the modelling task, the process is the same as in Grasshopper until the stage where the lofted surface is created. In Dynamo, this surface was then used to generate a grid of points, and these points were grouped into a list containing four sub-lists for the four corner points. These lists are then used to create the diagonal X lines, which were then piped, thereby creating the final piped structure.

### *Generative Components*

Nodes create new geometric entities. There are no nodes for modify or delete existing entities (although there is a node that transforms between coordinate systems.) Nodes are specified in two stages: the user first specifies the type of geometric entity to be created and then specifies the method that will be used to create that entity. Nodes do not output any non-geometric data, but may require non-geometric data as inputs. The dataflow graph is used mainly for visual feedback rather than for direct editing.

Links are created automatically by the system in response to expressions written by the user. When the user creates a new node, the parameters for the node can be defined using expressions that link to data within other nodes in the dataflow graph. Geometric data is represented using lists, with node inputs and outputs being handled in slightly different ways. Nodes outputs are always flat lists of geometric entities of the same type (including compound types). Node inputs may require different types of geometric and non-geometric data, structured either as lists or as nested lists.

Iteration results from using nodes with input arguments that are defined as being 'replicable'. This

indicates that a list of inputs can be used, resulting in the node iterating over these inputs. For methods with multiple parameters, users are given the option to toggle the replication pattern between one-to-one replication and cross replication.

For the modelling task, the coordinate systems were generated along the centreline curve and then scaled, with the scale factor being calculated using an expression. A hoop section was created and then copied onto the scaled coordinate systems, resulting in the hoops being automatically scaled to the right size. The hoop sections were then lofted to create a surface. A grid of points is then generated on the surface, which were then used to generate a grid of polygons. A component was defined that had a polygon as the input and a pair of diagonal pipes as the output. This component was then applied to the list of polygons, thereby replacing each polygon with a diagonal pair of pipes.

### *Houdini*

Nodes process geometric data, including creating, modifying, and deleting geometry. Nodes tend to perform complex operations and may have many parameters that affect their behaviour.

Links may be of two types: links for geometric data and links for non-geometric data. Geometric links are created by the user connecting nodes in the graph (similar to Grasshopper). Non-geometric links are created by the user writing expressions that link node parameters to data within other nodes in the dataflow graph (similar to Generative Components). Geometric data is represented using a topological data structure consisting of three levels: points, vertices, and primitives. Primitives are any type of geometric entity, including lines, polygons, splines, and so forth. The topological data structure supports user-defined attributes at all three levels.

Iteration results from nodes iterating over lists of geometric entities within the topological data structure. In order to support more complex types of control flow, 'switch' nodes and 'for-each' nodes can be used. Complex nested loops can be created by nesting 'for-each' nodes inside one another.

For the modelling task, the centreline curve was converted into a polyline with equal segments, and the vectors for the coordinate systems were then attached as attributes to the points on the polyline. The scale factors were calculated and also attached as attributes to the same polyline points. A hoop section was created and then copied onto each of the points, with the section being automatically oriented and scaled according to the point attributes. A surface was then generated by lofting the hoop sections, which was then converted onto a polygon surface. A loop was then defined (using a 'for each' node) that replaced each polygon with a diagonal pair of pipes.

### *Analysis of dataflow graphs*

For all four VDM environment, the aim was to model the Kilian roof using the minimum number of nodes without resorting to multi-line scripting. In Grasshopper and Dynamo, input nodes such as sliders were not counted, as these are considered to be optional. In Generative Components, multiple points were generated using a single node (using the `Point.ByCoordinateList()` method) rather than one node for each point. In Houdini, the nodes nested inside the 'for-each' node were also counted.

The four dataflow graphs for the Kilian roof were analysed using three metrics: the number of nodes in the graph, the number of links in the graph, and the cyclomatic complexity of the graph (McCabe 1976). Cyclomatic complexity is a metric for measuring code structure by calculating the number of independent paths through a directed acyclic graph. In order to ensure that code is understandable and easily maintainable, it is typically recommended that the cyclomatic complexity should be no greater than 10. The complexity M is then defined as:

$$M = E–N + 2 \quad (1)$$

where E is the number of edges in the graph and N is the number of nodes in the graph. Note that for nodes that are connected using parallel links, only one link is counted.

The results are shown in the Table 1. Despite the inherent linearity of the modelling task, none of the environments resulted in graphs with cyclomatic complexity of 1. For Grasshopper, Dynamo, and Generative Components, the results are similar, with an average of 24 nodes, 27 links, and a cyclomatic complexity of 5. These numbers are seen to be very high and reflect the fact that with these environments, the complexity of the dataflow graph is not commensurate with the complexity of the modelling task. Furthermore, the results suggest that tasks with a more realistic level of complexity will likely exceed the recommended cyclomatic complexity threshold of 10. A similar conclusion was reached by Davis at al. (2011), who analysed a wide range of dataflow graphs and found that 60% of the models had a cyclomatic complexity greater than 10, despite the fact that many of the models were thought to be snippets of larger more complex models.

The results for Houdini were markedly better. This improvement was attributable to a number of factors. First, fewer nodes were required due to the fact that nodes in Houdini may perform more than one modelling operation . Second, fewer links were required due to the fact that data could be embedded within the topological data structure as attributes. Third, the iterative loop for creating the X frames was easily handled with just a 'for each' node.

Table 1
Analysis of the dataflow graphs created using existing VDM environments. 'Nodes' is the number of nodes, 'Links' is the number of links, and 'CC' is the cyclomatic complexity.

| VDM Environment | Nodes | Links | CC |
|---|---|---|---|
| Grasshopper | 24 | 27 | 5 |
| Dynamo | 27 | 29 | 4 |
| GenerativeComponents | 20 | 24 | 6 |
| Houdini | 16 | 16 | 2 |

Another issue that was considered was comprehensibility of the iterative behaviour of nodes. This was not quantitatively measured, but was instead evaluated subjectively through experience of working with all four environments. The approach used by Grasshopper, Dynamo, and Generative Components, whereby the node iterates over trees or nested lists, was found to quickly get very complicated. Since nodes can have multiple inputs, with each input possibly consisting of some complex nested structure, the way that the iteration will be executed can be very difficult to predict. For Houdini, iteration was less complex and avoided the many data restructuring nodes require in other environments. However, the behaviour of individual nodes is complex and it is generally very difficult for novice users to know which are the right nodes to use.

One last issue to mention is the use of sub-graphs (also referred to as 'clusters' or 'subnets'). All four VDM environments allow users to create nodes that contain a sub-graph. To some extent, this may reduce complexity by enabling users to encapsulate a closely related set of nodes as a single sub-graph. However, it also has the critical downside that it obfuscates the dataflow, requiring users to continually navigate in and out of such sub-graphs.

## PROPOSED VDM ENVIRONMENT

The analysis of the dataflow graphs created using existing VDM environments resulted in the identification in three key issues: too many nodes, too many links, and confusing iteration. Too many nodes refers to datafow graphs that get very large due to the fact that each operation is represented by a separate node, even when some are relatively trivial. Too many links refers to dataflow graphs that get highly tangled (also sometimes referred to as 'spaghetti code') due to the fact that nodes tend to require many inputs. Confusing iteration refers to dataflow graphs that use techniques for iterating that are either highly implicit, or when defined explicitly are very complex.

For any given parametric modelling task, Vidamo aims to support a style of VDM where the complexity of the dataflow graph remains commensurate with the complexity of the task. In order to achieve this, Vidamo introduces two key features: a hybrid visual programming approach and a topological data structure. The hybrid approach allows nodes to be defined that perform more than one operation. The topological data structure allows complex sets of interconnected geometric entities to be represented as a sin-

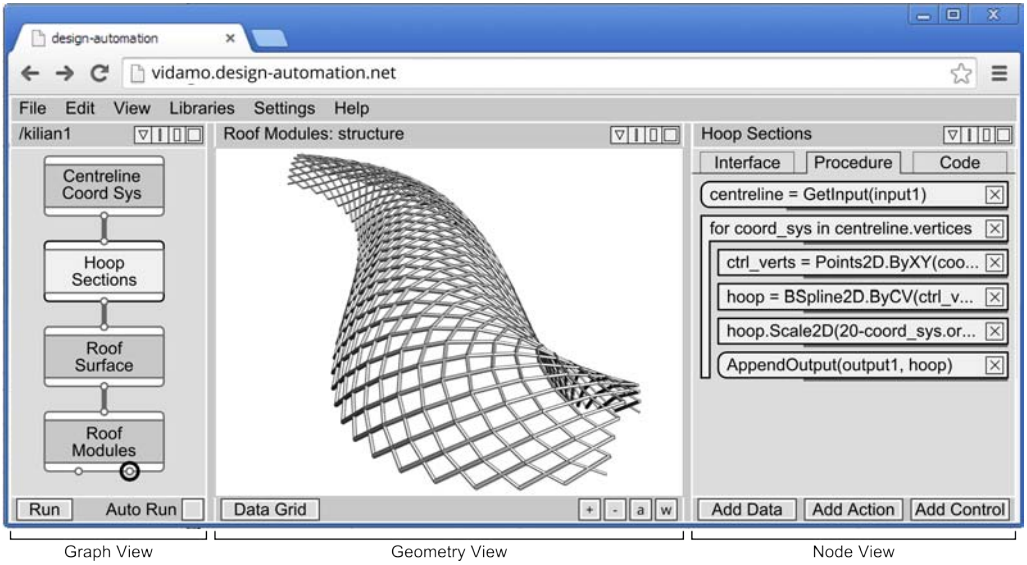



Figure 1
The Vidamo interface in a web browser.

gle geometry collection.

The hybrid visual programming approach combines two styles : the dataflow nodes-based style and the procedural tile-based style. The dataflow style is used to define an overall sequence of modelling procedures, while the procedural style is used to define each individual procedure. (Note that this approach combines dataflow and procedural styles of programming without needing to invent a new programming language, as proposed by Aish (2013)). The dataflow style can be traced back to the existing VDM environments described above, while the procedural style can be traced back to tile-based visual programming environments such as Scratch (Resnick et al. 2009). The way that these various ingredients are combined is novel and supports a more powerful style of VDM that is capable of handling greater complexity.

The approach allows users to decide on the granularity of the dataflow graph. For the Kilian roof modelling task, the user could decide to perform the whole task using just a single node containing a procedure with 16 instructions, or they could decide to break the task into 8 nodes, with each node containing a short procedure consisting of just a few instructions. In the example given in Figure 1, the task is broken into 4 nodes. The 'Centreline Coord Sys' node creates the centreline curve and uses it to create a set of correctly orientated coordinate system. The 'Hoop Sections' node creates a hoop section on each coordinate system and scales each section by the appropriate amount. The 'Roof Surface' node lofts the section curves to create a roof surface and then converts this into a polygon surface. The 'Roof Modules' node replaces each polygon on the roof surface with a diagonal pair of pipes. The 'Hoop Sections' and 'Roof Surface' nodes are described in more detail below.

Vidamo is being developed as a web-application that will run entirely in the browser. Figure 1 shows a sketch of the proposed user-interface, showing three types of views: the graph view, the geometry view, and node view. The graph view displays the dataflow graph. The geometry has two modes: a 3D mode that shows the model or a spreadsheet mode, called the 'Data Grid'. The latter allows geometry to be viewed as data, with every row in the spreadsheet represent-

ing one geometric entity. The node view has three tabs: the 'Interface' tab allows users to customize the node's user interface, including adding various interface components such as sliders and tick boxes; the 'Procedure' tab allows users to define the node's underlying procedure using a tile-base visual programming style; and the 'Code' tab allows users to inspect the generated code. In this paper, only the geometry 3D mode and the 'Procedure' tab will be discussed.

The information that is displayed in the geometry or the node view is dependent on what is selected in the graph view, with two types of selection being possible: selecting a port and selecting a node. The geometry view displays the data that is passing through the selected port. In Figure 1, one of the output ports of the Roof Module nodes is selected (as shown by the black circle), and the geometry view therefore shows the roof X modules. The node view displays the information for the selected node. In Figure 1, the 'Hoop Sections' node is selected, and the node view therefore shows the procedure for that node. These procedures will be described in more detail below.

### *Fewer nodes*

In order to reduce the number of nodes, Vidamo uses nodes capable of executing a procedure consisting of more than one operation. For all geometric operations, there is only one type of node, which acts as a generic container for a geometric procedure.

The generic geometry node can have an arbitrary number of inputs and outputs. Nodes may have no inputs, in which case the procedure will generate geometry from scratch. Nodes with no outputs are possible, but are only used for specific purposes such as writing files. Node input and output data may consist of either geometric data or non-geometric data. For geometric data, the data is represented using a topological data structure (see section "Fewer Links"). For non-geometric data, the data can be structured as either flat or nested lists.

When a user adds a generic geometry node to the graph, they need to define the procedure for that node. Procedures are created by inserting three types of instructions: data instructions, action instructions, and control instructions. Data instructions get and set data to local variables and input and output ports. Action instructions perform any types of geometric operations. Control instructions consist of control flow statements such as 'for' loops, 'while' loops, and 'if' statements.

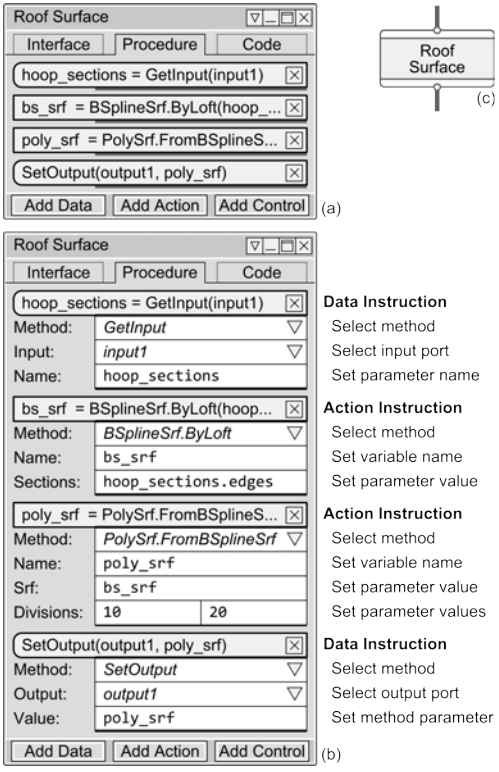



Figure 2
The node view for the 'Roof Surface' node. (a) Instructions with collapsed settings. (b) Instructions with expanded settings. (c) The node in the dataflow graph.

Figure 2 shows the 'Roof Surface' node with a procedure consisting of four instructions: a data instruction, two action instructions, and another data instructions. Each instruction is defined by a number of settings that can be collapsed or expanded. Figure 2 (a) shows the collapsed settings, while Figure 2 (b) shows the expanded settings. In order to spec-

ify the action to be performed, the user must select a method using a series of nested menus. Once the method is selected, parameter boxes are then dynamically generated so that the user can specify the values for the method parameters.

The data instructions read and write data to the input and output ports. For reading data, the `GetInput()` method is used, and for writing data, the `SetOutput()` method is used. For the first action, the `BSplineSrf.ByLoft()` method has been selected which generates a BSpline surface by lofting a set of section polylines. The surface is assigne to the local variable `bs_srf`. The section lines are specified by the expression `hoop_sections.edges`, which refers to a set of curves in the topological data structure (see section "Fewer Links"). For the second action, `PolySrf.FromBSplineSrf()` method has been selected which generates a polygon surface from a BSpline surface. The polygon surface is assigne to the local variable `poly_srf`. The BSpline surface is the one created by the previous action, `bs_srf`.

### *Fewer links*

In order to create graphs that are less tangled, Vidamo uses a topological data structure capable of representing geometric relationships and storing non-geometric data. The use of such a data structure means that many of the relationships between geometric entities do not have to be represented in the dataflow graph, but are instead an integral part of the data structure (which is itself a graph). The data structure and underlying geometric entities are defined using the Open Cascade modelling kernel [8].

A set of geometric entities represented using the topological data structure is referred to as a 'geometry collection'. Within such a collection, complex shapes are modelled as assemblies of simpler shapes. Table 2 lists the eight types of shapes defined by the OpenCascade kernel. These shapes are abstract entities that are associated with underlying geometric entities. For example, a model may contain a number of edges, each of which may be associated with a different type of geometric entity, such as a line, an arc, or a BSpline curve. A geometry collection can therefore contain a large number of geometric entities related to one another in complex ways.

| Shape | Description |
|---|---|
| Vertex | Shape corresponding to a point. |
| Edge | Shape bounded by a vertices. |
| Wire | Sequence of edges connected by vertices. |
| Face | Part of a surface bounded by closed wires. |
| Shell | Set of faces connected by edges. |
| Solid | Part of 3D space bounded by shells. |
| CompSolid | Set of solids connected by their faces. |
| Compound | Set of any other shapes described above. |

Table 2
The eight typological shapes available in OpenCascade.

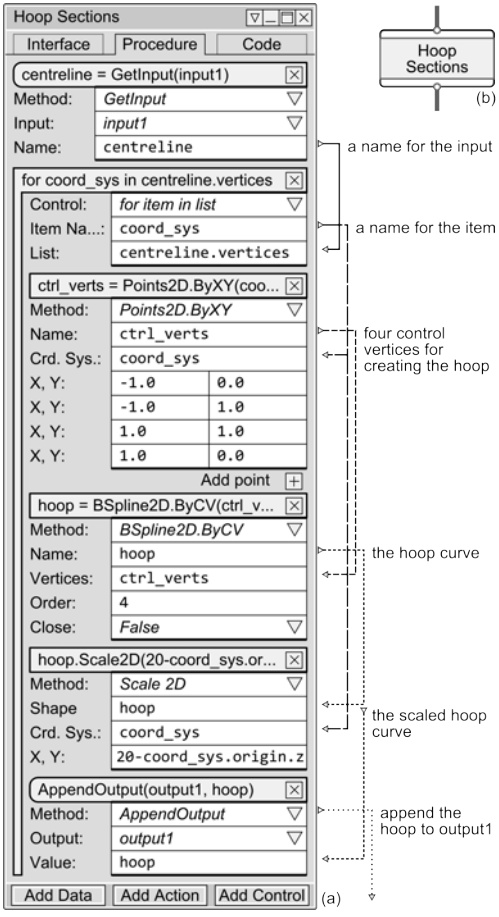
For the 'Roof Surface' node described above, the input and output data are both geometry collections. The input collection consists of vertices and edges, with the vertices being points and edges being the BSpline curves. The output collection consists of five topological levels, from vertices to shells. The vertices are points, the edges are straight lines, the wires are sets of four connected lines, the faces are polygons, and the shell is a set of connected polygons.

The topological data structure allows lower level entities to be extracted from higher level entities using simple expressions. The expressions specify data queries using the topological level as a way of navigating through the data structure. For example, the expression `input1.edges[1].vertices[0].z` will return the z coordinate of the first point of the of the second curve in the collection `input1`. This ability to extract specific pieces of data from deep within the geometric collection means that nodes often only need a single input, thereby significantly reducing the numbers of links that are required in the dataflow graph.

### *Explicit iteration*

In order to reduce the obfuscation of iteration, Vidamo allows iteration to be explicitly specified using loops, thereby giving users complete control over the iterative process. Loops fall under control flow instructions, which are one of the three instruction types that can be created for node procedures.

Figure 3
The node view for the 'Hoop Sections' node that includes a loop. (a) The procedure showing a list of instructions with expanded settings. (b) The node in the dataflow graph.




The ability to add control flow to the procedures makes this approach Turing complete, theoretically allowing procedures of arbitrary complexity to be defined. Figure 3 show the procedure for 'Hoop Sections' node, which includes a 'for loop'. As can be seen in Figure 1, this 'Hoop Sections' node actually precedes the 'Roof Surface' node sown in Figure 2.

In order to create the hoops, the node has to iterate over each of the coordinate systems in the input geometry collection. The coordinate systems are created and oriented by the 'Centreline Coord Sys' node. Since the hoop sections lie on a flat plane, 2D coordinate systems have been used, defined as a set of vertices in the topological data structure.

The 'for loop' is the second instruction in the procedure shown in Figure 3. The type of 'for loop' that has been used in this case is the 'for item in list' type, which iterates through a list of items. For the settings, the 'Item Name' defines the name of a local variable whose value will be equal to the item, and 'List' refers to the list of items to be iterated through.

Inside the loop, four instructions have been defined: three action instructions and one data instruction. The first action creates four 2D points on the 2D coordinate system. The second action create a 2D BSpline curve through these points and assigns it to a local variable called `hoop`. The third action scales the hoop by a factor that ensures that the roof ridge line will remain at a constant height. The final instruction is the data instruction, which appends the scaled hoop to the geometry collection for the node output.

### *Analysis of the dataflow graph*

The dataflow graph shown in Figure 1 has 4 nodes and 3 links, and therefore has a cyclomatic complexity of 1. Furthermore, even if more nodes were used, there would still be no need to create a non-linear graph and therefore the cyclomatic complexity would remain the same. For example, another user might decide to split the 'Centerline Coord Sys' node into two: one node to create the centreline and another to place coordinate systems along that line. This would result in 5 nodes with 4 links, which would still have a cyclomatic complexity of 1.

The reason that the linearity of the dataflow graph can be maintained is due to the fact that the parametric modelling task is itself liner. This is of course not always the case. More complex parametric models may require non-linear graphs. However, with Vidamo the user can ensure that the complexity of the dataflow graph remains commensurate with the complexity of the parametric modelling

task. This is not the case with existing VDM environments, where the complexity of the dataflow graph will typically be significantly higher than the complexity of the parametric modelling task.

## CONCLUSIONS

Four existing VDM environments were used to model the Kilian roof and the resulting dataflow graphs were analysed. Considering the simplicity of the task, the size and complexity of the dataflow graphs was found to be very high.

Vidamo is a new type of VDM environment that results in dataflow graphs that are are much much smaller and less complex. It achieves this by integrating two key features: a hybrid visual programming approach and a topological data structure. The hybrid approach uses the dataflow nodes-based style for defining a sequence of procedures and the procedural tile-base style for defining each individual procedure. The topological data structure defines a hierarchy of shapes that allows complex shapes to be modelled as assemblies of simpler shapes.

Vidamo is in the early design stages, and therefore the modelling of the Kilian roof was conducted as a theoretical exercise. The use of a hybrid approach combined with a topological data structure allowed the Kilian roof example to be modelled using significantly smaller dataflow graphs with a cyclomatic complexity of only 1. Furthermore, due to the explicit way that loops are represented, it is argued that the comprehensibility of the iterative behaviour is much better than in existing VDM environments.

The future development of Vidamo will focus on three areas: support for debugging, support for linking to third-party tools (such as simulation programs), and support for generating data-rich models (such as Building Information Models). These are all areas in which current VDM environments are lacking.

## REFERENCES


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

Davis, D 2013, *Modelled on Software Engineering: Flexible Parametric Models in the Practice of Architecture*, Ph.D. Thesis, RMIT University

Davis, D, Burry, J and Burry, M 2011, 'Understanding Visual Scripts: Improving collaboration through modular programming', *International Journal of Architectural Computing*, 9, pp. 361-376

Janssen, P and Chen, KW 2011 'Visual Dataflow Modelling: A Comparison of Three Systems', *Proceedings of the 4th International Conference on Computer Aided Architectural Design Futures (CAAD Futures 2011)*, Liege, Belgium, pp. 801-816

McCabe, T 1976, 'A Complexity Measure', *IEEE Transactions on Software Engineering*, 2, pp. 308-320

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

Senske, N 2014 'Confronting the Challenges of Computational Design Instruction', *Proceedings of the 19th International Conference on Computer-Aided Architectural Design Research in Asia*, Kyoto, Japan, pp. 821-830

Woodbury, R, Aish, R and Kilian, A 2007 'Some Patterns for Parametric Modeling.', *27th Annual Conference of the Association for Computer Aided Design in Architecture*, Halifax, Nova Scotia, p. 222–229

[1] http://www.grasshopper3d.com/

[2] http://dynamobim.org/

[3] http://www.bentley.com/en-US/Promo/Generative%20Components/default.htm

[4] http://www.sidefx.com/index.php?option=com_content&task=view&id=1000&Itemid=266

[5] http://www.esri.com/software/cityengine/features

[6] http://www.3dvia.com/studio/documentation/user-manual/ai-artificial-intelligence/pathfinding/case-example

[7] http://wiki.blender.org/index.php/Extensions:2.6/Py/Scripts/Nodes/Sverchok

[8] http://www.opencascade.org/doc/occt-6.7.1/overview/html/occt__tutorial.html