

# DesignScript:

## a domain specific language for architectural computing

Robert Aish

Bartlett School of Architecture, UCL  
robert.aish@ucl.ac.uk

Emmanuel Mendoza

ARM Ltd  
jun.mendoza@arm.com

### Abstract

DesignScript is a multi-paradigm domain-specific end-user language and modelling environment for architectural and engineering computation. DesignScript implements both visual data flow programming and imperative programming. The novice user initially develops his data flow program through the familiar visual programming environment. This environment is effectively an intuitive user interface masking the underlying DesignScript language. The DesignScript language and its related user interface addresses three issues: the domain specific requirements of architectural and engineering computing, the scalability issues encountered when visual data flow programming is applied to complex design scenarios and the abstraction barriers encountered when users transition from data flow to imperative programming.

**Categories and Subject Descriptors** D.3.3 [Software and its engineering]: Software notations and tools – General programming languages – Language types – Imperative languages, Data flow languages, Multiparadigm languages; Compilers – Dynamic compilers; Context specific languages – Scripting languages, Domain specific languages, Visual languages; Applied Computing – Physical sciences and engineering – Engineering – Computer-aided design; Applied Computing – Arts and humanities – Architecture (buildings) – Computer-aided design.

**General Terms** Design, Human Factors, Languages.

**Keywords** exploratory design; scalability; extensibility; learning to program; end-user programming; abstraction barriers, abstraction gradient.

### 1. Introduction

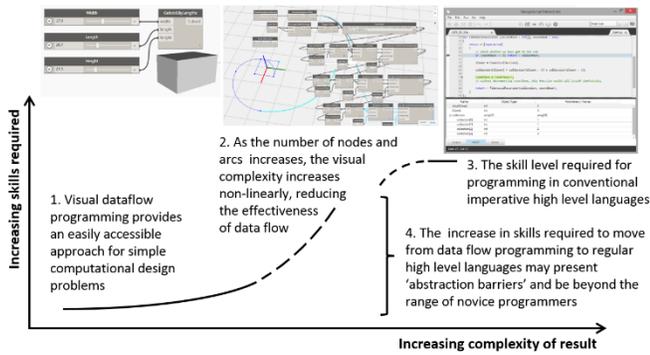
In this paper the term ‘architecture’ is used both in its original sense referring to the design of buildings and in the more general sense as the schematic design of engineering systems. DesignScript sits firmly within a branch of domain specific computing broadly referred to as Computer Aided Design (CAD). The earliest example of CAD is SketchPad developed in 1963 by Ivan Sutherland [1]. Since then there has been a proliferation of CAD systems for different application domains. It is possible to generally describe CAD systems and their use with three characteristic dimensions:

#### 1.1 ‘domain specific to general purpose’ dimension

At one extreme there are ultra-domain specific CAD systems where the user is presented with a predefined schema of domain specific components which can be assembled into models using pre-defined inter-component relationships. This schema typically represents a physical ‘kit of parts’, such as a pre-fabricated building system. These systems are intended for users who are domain experts. These systems represent the constraints and conventions of established engineering and construction practice.

At the other extreme there are completely general purpose systems with programming and geometry libraries which are intended for users who are both domain experts and accomplished software developers.

In between these two extremes there are hybrid systems which combine predefined domain specific components with more general purpose programming and geometry tools. These tools can be used by more proficient users to customise and extend the CAD system by defining new components. These hybrid CAD systems are used in more open-end application domains, such as advanced building architecture, where innovation beyond current conventions (and



**Figure 1.** The sequence of increasing programming complexity. Abstraction barriers and consequential discontinuity may be encountered with existing domain specific applications when transitioning from visual data flow programming to text based programming.

beyond a fixed schema) is important. For example, it is entirely feasible for a ‘one-off’ building to be based on a unique architectural form, custom components and specialised fabrication techniques.

The architecture of buildings is not a single application domain. Buildings are essentially the integration of a number of sub-systems (for example, structure, services, cladding, etc.) which are the responsibility of different design and engineering disciplines. Often each discipline uses separate design and engineering computer applications in what are effectively separate ‘vertical silos’.

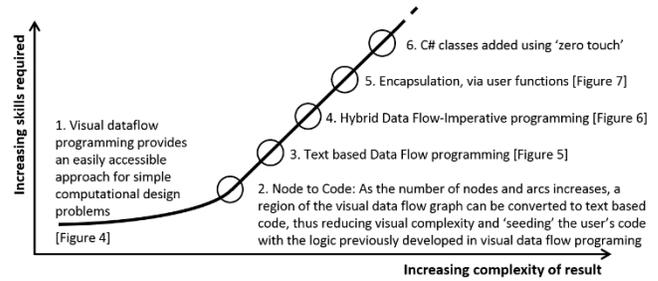
To create a truly integrated building requires that the contribution of each of these disciplines is integrated. This in turn requires that the different modelling tools are integrated.

Ideally the different domain specific applications should be based on a common set of computing abstractions and these abstractions should be exposed to the users so that inter-domain integration can be created to suit the needs of each new building project. These ‘building system integration’ issues are discussed by Rush [2].

## 1.2 Scaling from exploratory design to detail design

It is essential that an effective domain specific system supports a scale of use from simple exploratory model to complex detailed models. Architectural design often starts with exploratory sketch models, involving just a few abstract geometric elements and relationships. At the exploratory stage the architectural user may only be representing some speculative design intent without committing to particular dimensions, materials or construction processes.

During the course of the design process this model will be expanded to potentially hundreds of thousands components and the model will be developed from an initial abstract geometry to an assembly of highly specific construction components. While domain specific application based



**Figure 2.** The objective of DesignScript is help the user to harness computing abstractions, but without these becoming abstraction barriers. This is achieved by creating an abstraction gradient of easily assimilated intermediate steps between visual data flow and text imperative programming.

on visual data flow programming attract an initial use with small scale exploratory design models, this technique do not scale to complex real world design projects [3].

## 1.3 ‘Skills’ dimension

Domain specific data flow applications have enabled programming to be accessible to architectural users without any prior computing experience. However these applications are limited because they present a restricted subset of computing concepts. The paradox of domain specific computing is that it can help the user develop certain programming skills but the familiarity with these skills may trap the user within a simplified version of programming.

Scalability and usability issues arise when data flow programming has to be applied to more complex computational tasks found in architectural design. While regular programming and scripting languages offer a more complete set of computational concepts, learning these languages often presents abstraction barriers to novice end-user programmers such as architects [Figure 1][4].

In this sense the aim of a domain specific application as an educational tool is not to avoid abstractions, but to avoid abstractions becoming a barrier.

It is important to recognise that a domain specific computing system should not be designed around a fixed definition of the task or to anticipate a fixed set of user skills. The use of the system will change the user and in turn change how the system is used.

In practice all these dimensions (domain specific to abstract, scalability and skills) interact. It is not possible to address the scalability, extensibility and integration issues without first addressing the skills issue. More generally a domain specific computing system will only be successful if it is more than domain specific and introduces the user to more general purpose computing ideas and their application.

## 2. DesignScript

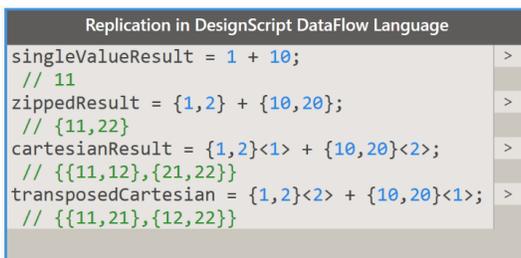
DesignScript implements a series of intermediate programming techniques between visual data flow programming and regular text based programming [Figure. 2]. This provides an abstraction gradient which allows the gradual introduction of more advanced computing abstractions and notation:

**Visual data flow programming:** This uses the familiar visual ‘graph-node-arc’ programming UI. It is an ideal entry point where the user can develop the fundamental programming skill that of expressing ideas in a logical and executable form. The visual graph-node-arc convention surreptitiously introduces the user to the concept of ‘type’ by requiring him to match the ‘type’ of the output of an upstream node with the ‘type’ expected as input to a downstream node. However, visual programming can become extremely verbose and does not scale to more complex programming tasks. [Figure 4]

**Node-to-Code:** Effectively the visual programming environment is a graphical UI where each node is represented by compiled DesignScript source code. The ‘node-to-code’ process merely exposes this underlying code in the selected nodes as a single ‘code block’ node. This dramatically reduces the visual complexity of the graph and ‘seeds’ the user’s code with the visual logic previously developed.

**Text based data flow language:** This uses conventional C style syntax but is simplified because there are no explicit flow control statements. Flow control is determined by the data flow dependencies between the variables. [Figure 5]

**Replication:** This allows the user to create and operate on collections without initially needing to understand iteration [Figures 3, 4 and 5]. This is a domain specific functionality of particular importance in the design of buildings, which are composed of collections such as floor, columns, beams, façade elements, etc. As the architect explores design options these collections may vary in size and in rank. In the example in Figures 4 and 5 we show a single point [the *controlPoint*] created using single values for the



```
Replication in DesignScript DataFlow Language
singleValueResult = 1 + 10; >
// 11
zippedResult = {1,2} + {10,20}; >
// {11,22}
cartesianResult = {1,2}<1> + {10,20}<2>; >
// {{11,12},{21,22}}
transposedCartesian = {1,2}<2> + {10,20}<1>; >
// {{11,21},{12,22}}
```

**Figure 3.** Replication in DesignScript enables the user to operate directly on collections. “Replication guides” use the syntax  $\langle n \rangle$  where  $n$  controls the order in which the input collections are used to build the output collection.

X, Y and Z coordinates and then the use of ‘replication’ where a 2D collection of points [the *pointArray*] is created as the cartesian product of a set of X and Y coordinates. Essentially ‘replication’ blurs the distinction between the ‘type’ of a single variable and the ‘type’ of a collection.

**Imperative language:** This uses the familiar C style programming syntax. It is a more powerful and precise programming paradigm where standard control flow constructs such as “if”, “for” and “while” statements can be utilised [Figure 6]

**IDE:** DesignScript includes a conventional IDE to support the development of iterative and recursive functions and programs which are text based and cannot be debugged by ‘tracing through’ a visual data flow program. [Figure 7].

**Liveness:** DesignScript is a dynamic language and supports ‘liveness’ which is essential for exploratory programming. Liveness includes directly responding to user actions such as changes in program logic, dynamic interaction using ‘sliders’ to changes in the value of variables and the direct manipulation of control points in the graphical model.

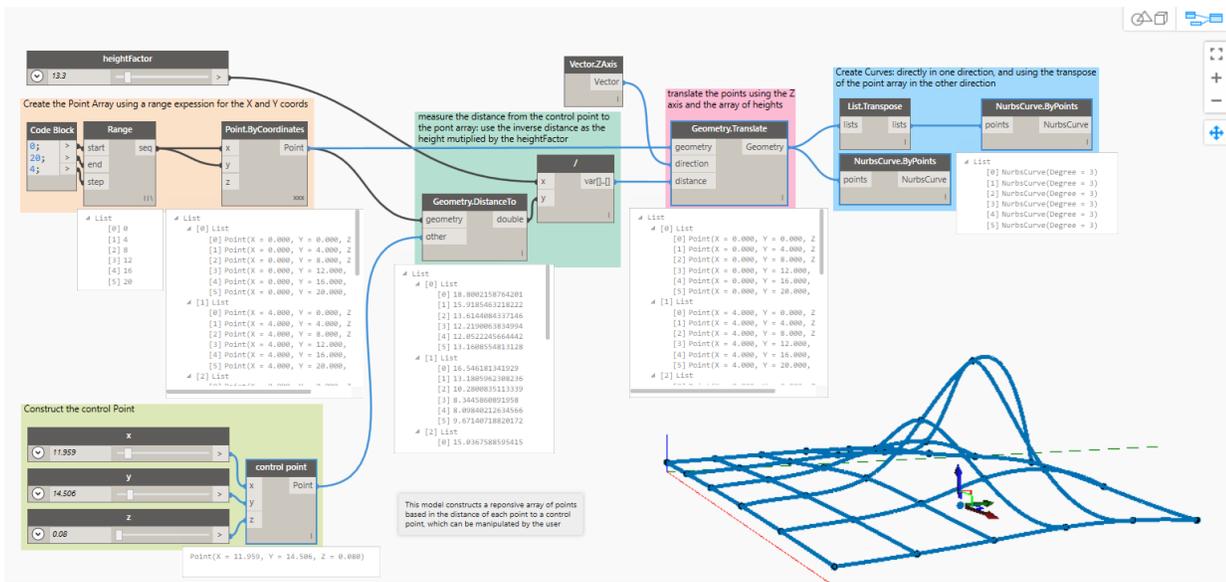
Liveness is synonymous with REPL [Read-eval-print loop] with live interaction between the user and the program [5]. Liveness/REPL behaviour is only applicable when DesignScript is continuously running in Automatic Mode in a visual programming environment such as Dynamo.

**Typing:** DesignScript supports optional typing. When variables are declared with a type, it aids the compiler to statically enforce type-safety. For variables with no assigned type, the compiler and the virtual machine will perform both static and dynamic type inferencing to determine the best type to assign to a variable.

**Domain specific syntax:** DesignScript avoids adding complex domain specific syntax. The text based data flow language combined with replication actually represents a simplification compared to the equivalent imperative language. The only additional syntax are replication guides [Figure 3]. Tutorial and reference documentation is available. [6] [7]

**Scaling:** DesignScript can scale from simple exploratory data flow models [Figures 4, 5, 6 and 7] to the full complexity of a completed building. [Figures 8, 9 and 10]

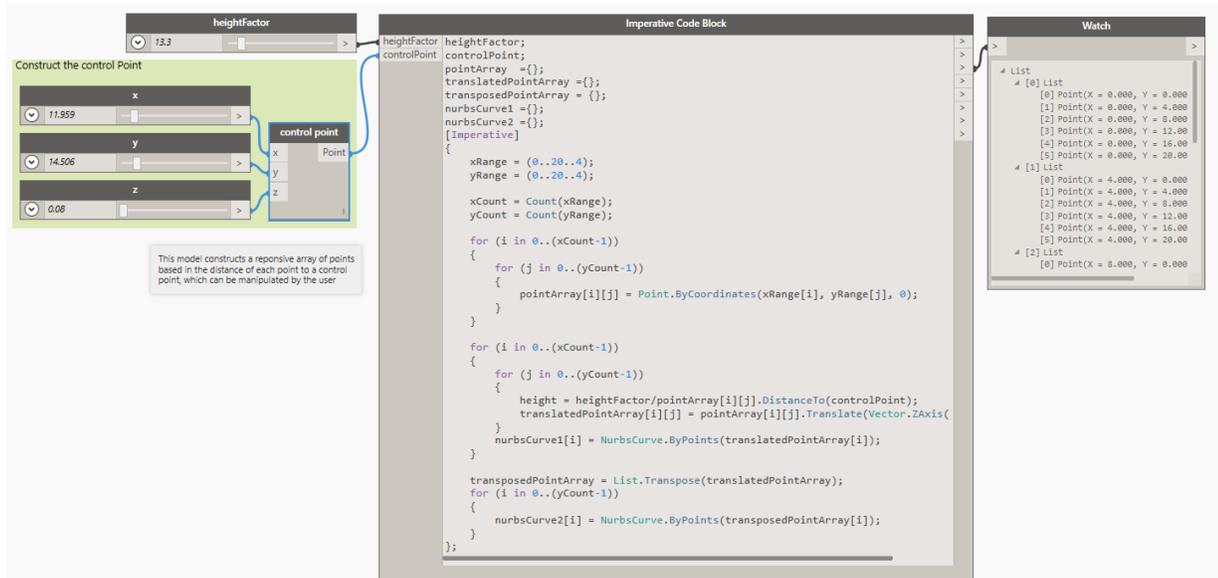
**From abstract to domain specific:** DesignScript can also be used as a completely general purpose programming tool to build other domain specific modelling applications. Figure 8 illustrates one such application which models the design and behaviour of the MIPS microprocessor pipeline [8]. In this application each codeblock node represents the identifiable regions of the processor and the data flow represents communication between these regions.



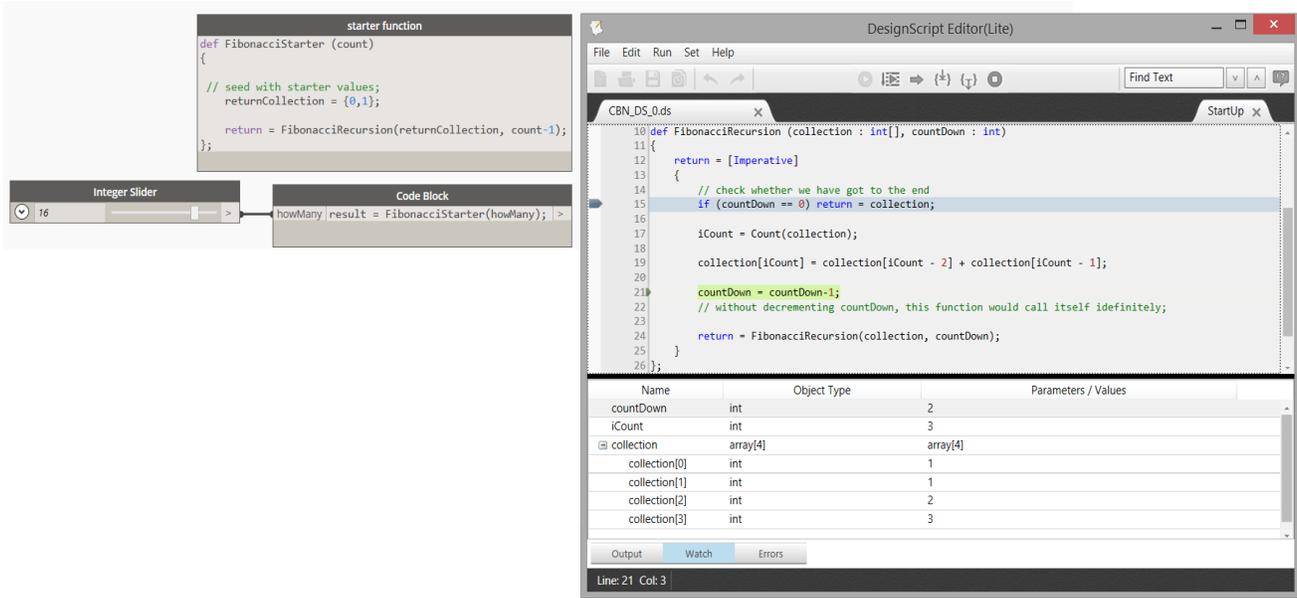
**Figure 4.** Visual Data Flow programming: here a set of curves is drawn through a 2D array of points, where the height of the points is based on the inverse distance to a control point.



**Figure 5.** Text based data flow programming in DesignScript. Flow control is based on the graph dependencies of the original data flow program. Notice the ability to directly operate on collections without iteration,



**Figure 6.** The equivalent program in the DesignScript Imperative language using familiar ‘C’ style syntax and flow control statements which allow the explicitly iteration through collections.



**Figure 7.** DesignScript supports a conventional IDE. In Data flow programming the program logic can be ‘traced through’ using the visual representation so there is no need for an IDE. However with Imperative programming where there is iterative and recursive logic, then it is important that the user can delve into the program execution.

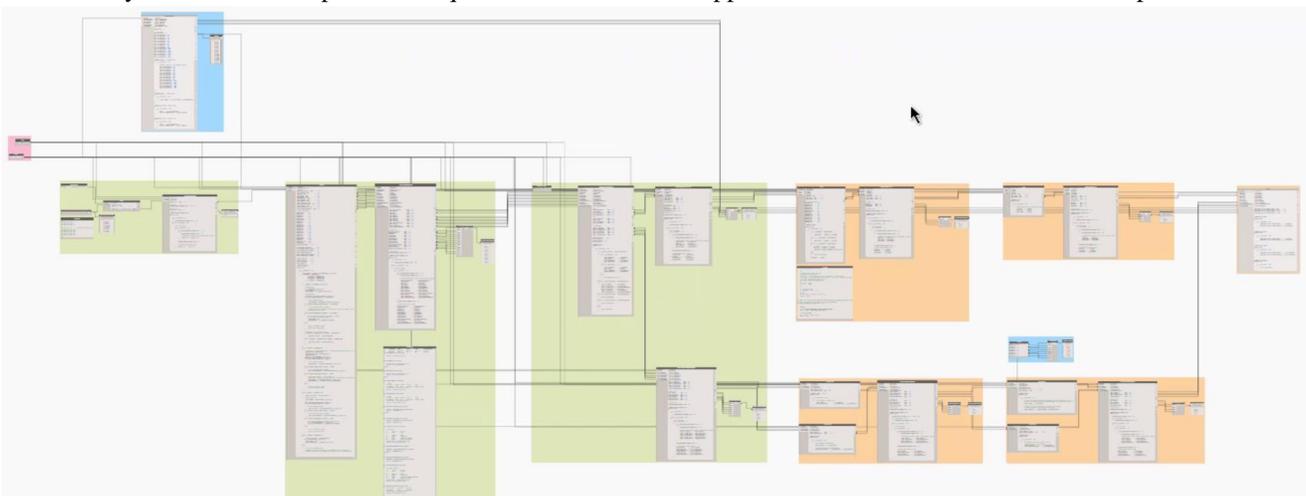
### 3. Design and implementation:

In visual data flow programming each node is represented by compiled DesignScript source code. Imperative code blocks are treated as nodes in the data flow graph. The DesignScript compiler then performs standard optimization to the input source code and emits an executable format that contains the following: (1) Symbolic information (2) Executable bytecode (3) Dependency graph nodes. A virtual machine loads this data and performs runtime execution while enforcing dataflow semantics. The virtual machine is driven by the data flow dependency graph where each node points to a set of bytecode to be interpreted in sequence.

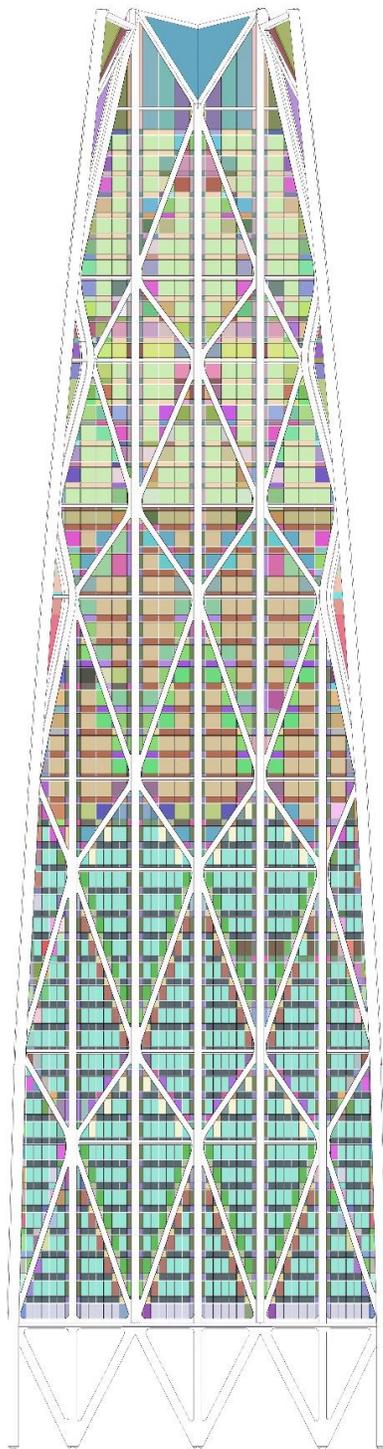
Only modified nodes and those that depend on the modified nodes are re-compiled and re-executed, thus giving highly efficient dynamic execution. Other DesignScript nodes/source code can include calls to C#, thereby giving access to extensive external libraries.

### 4. Conclusions:

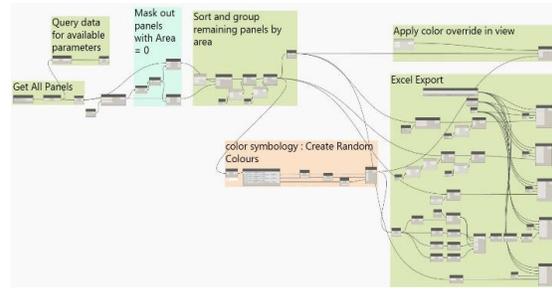
DesignScript addresses a particular form of domain specific computing which is required to be scalable and extensible while also acting as a learning environment. DesignScript illustrates the maxim that an effective domain specific application has to be more than domain specific.



**Figure 8.** Using the hybrid visual and text based programming to model the MIPS microprocessor pipeline



**Figure 10.** Façade geometry created using data flow programming, including the automatic creation of curtain wall gridlines based on the planning grid geometry and façade panel analysis colour coded based on fabrication criteria. Images courtesy of Foster + Partners



**Figure 9.** Data flow programming used in the design of the 'Oceanwide' high rise building by Foster+Partners.

We have found that a hybrid approach combining textual and visual programming and combining data flow and imperative programming allows the user to model the overall process as a visual diagram, while the logic of the individual processes can be programmed using the text based language.

## Acknowledgements

The authors would like to acknowledge the contribution of the DesignScript development team at the Autodesk Singapore Research and Development Centre. DesignScript is an open source project and is the core computational engine within the 'Dynamo Studio' application. The authors would like to thank Foster+Partners for permission to use of the illustrations in Figures 9 and 10.

## References

- [1] Ivan Sutherland "Sketchpad" MIT (1963).
- [2] Richard Rush "The Building Systems Integration Handbook" AIA (1986)
- [3] Margaret M. Burnett et al., "Scaling Up Visual Programming Languages," in Computer 28 no. 3 (1995): 45 [[ftp://ftp.cs.orst.edu/pub/burnett/Computer-scalingUp-1995.pdf](http://ftp.cs.orst.edu/pub/burnett/Computer-scalingUp-1995.pdf)]
- [4] Thomas Green and Alan Blackwell "Cognitive Dimensions of Information Artefacts": a tutorial, Version 1.2 (1998) [[www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf](http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf)]
- [5] REPL [[https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print\\_loop](https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop)]
- [6] DesignScript summary user manual: [[http://au-cache.autodesk.com/au2012/sessionsFiles/3286/5471/handout\\_3286\\_DesignScript\\_summary\\_user\\_manual.pdf](http://au-cache.autodesk.com/au2012/sessionsFiles/3286/5471/handout_3286_DesignScript_summary_user_manual.pdf) ]
- [7] DesignScript tutorial as implemented with the Dynamo UI [[http://dynamoprimer.com/en/07\\_Code-Block/7\\_Code-Blocks-and-Design-Script.html](http://dynamoprimer.com/en/07_Code-Block/7_Code-Blocks-and-Design-Script.html) ]
- [8] John Hennessey, David Patterson "Computer Architecture, a Qualitative Approach 5<sup>th</sup> Edition", Morgan Kaufmann (2014)