Design of Dataflow

Tuğrul Yazar

Published online: 5 December 2014
© Kim Williams Books, Turin 2014

Abstract Design scripting is initially based on secondary syntactic mechanisms that control sequential executions of code. This model of computation, called control flow programming, was introduced to designers as the only way of creating algorithms. Dataflow represents a widespread alternative to control flow by utilizing concurrent streams of data. Dataflow programming languages are also considered to be pedagogical tools in fields where the coders are not expert computer programmers. In the last decade, new parametric modeling tools introduced this conception to design scripting, shifting its practical and educational foundations. Dataflow programming represents a promising platform for studying contemporary parametric modeling techniques in design, while architects and designers increasingly use it as their algorithmic sketchbooks. This paper describes a purpose to reflect this shift in design education, pursuing four pedagogical patterns derived from in-class experiments: “explication”, “kit of components”, “objectification”, and “re-generation”.

Keywords Design scripting · Dataflow · Pedagogical patterns · Parametric modeling · Visual programming · Design education

Introduction

Generative algorithms have been integrated into design in a sequential form where text-based scripting languages are accepted as mainstream tools. This approach to

Presented at the conference Nexus 2014: Relationships Between Architecture and Mathematics, Ankara, Turkey, 9–12 June 2014.

T. Yazar (✉)
İstanbul Bilgi Üniversitesi, Mimarlık Fakültesi KD.6-202 santralİstanbul,
Eyüp, 34060 İstanbul, Turkey
e-mail: tugrul.yazar@bilgi.edu.tr

design scripting gave rise to “control flow”, the dominant paradigm of programming languages. Methods of control flow eventually lead designers to seek secondary syntactic mechanisms in order to control program states, such as iterations, branchings, and conditionals. Developing sequential executions of such statements mostly created a cognitive layer in design praxis, and an esoteric skill for design research. In some cases, this leads to suppressing other design domains and can even become the de facto purpose of the design process. Moreover, it might be argued that control flow computing has inevitably become the primary conception upon which a whole body of knowledge generally known as “Algorithmic Architecture” (after Terzidis 2006) was constructed.

For less than a decade, casual coders of architecture and design have been working with a different way of planning and writing algorithms (Çolakoğlu and Yazar 2009). Pioneered by Grasshopper for Rhinoceros, and later Dynamo for Revit, the new generation of Visual Programming Languages (VPLs) proposed an alternative approach to design scripting. The collective social network of design computing reacted to these tools and methods faster than academia, where the novelty of this approach did not attract much attention. Among a few publications about design VPLs, (Leitão and Santos 2011) explain fundamental differences between textual and graphical approaches in scripting, and compare them according to usability in various design conditions. Celani and Vaz (2011) also study VPLs in order to compare different scripting languages for design education. Both of these studies conclude with an argument that, since visual construction of algorithms are perceptually limited, they are more successful in relatively simple and pedagogical uses. The discussion of design education via VPLs is further extended by Wurzer and Pak (2012), who present the first cognitive surveys about VPLs in design, stressing the necessity to develop tools and methods for design education.

The perceptual limitation of VPLs (generally called the “Deutsch Limit”) is a controversial argument as even extreme modularity (or clustering) could be implemented in visual interfaces just as effectively as in textual ones. Cognitive qualities of VPLs and their comparisons with other programming environments have been the subject of research in computer science for at least 35 years. Readers are referred to Myers (1989), Nickerson (1994) and Menzies (1998) for in-depth studies and bibliographies about visual programming, program visualization, and related studies. Today, most of the experiments about the use of VPLs in architecture and design are conducted by the collective social network of design computing mostly via workshops. The expansion behavior of this network is a significant subject to study in order to understand (non)academic development of the field, but this is out of the scope of this present paper. Here we focus on the education regarding the computational model underlying VPLs mentioned above. It is not only textual or graphical interfaces, but the concept of “dataflow” make such languages exceptional and more exploratory for designers. After the definition of dataflow from the perspective of computational design research, various educational experiments are introduced in this paper, categorizing them according to causal connections called as “pedagogical patterns”.

Dataflow Design Computing

Since its definition as a novel computational model in mid-1970's, dataflow has found use in a wide range of programming languages. In a broad perspective dataflow represents a deterministic solution to the abstract modeling of concurrent systems, offering an alternative to traditional von Neumann machines and languages (Agerwala and Arvind 1982; Davis and Keller 1982). The concept of dataflow is generally traced back to William Sutherland's Ph. D. dissertation (Sutherland 1966), in which he describes graphical representations for algorithms (Sousa 2012). The term dataflow is also used in other fields such as business management and finance, when there is a need to model relationships that include parallel events. Although dataflow is generally related to parallel computing research on computer hardware, William Ackermann (1982: 15) argues that many of its properties existed long before dataflow computers were conceived. The bibliographic work of Wesley Johnston, Paul Hanna and Richard Millar (2004) explains the development of dataflow concept, which started as a study on computer architecture, and evolved into today's programming language and compiler research. Dataflow programming is alternatively called "stream-based programming" (after Uustalu and Vene 2006: 135) and "program graph execution model" (after Najjar et al. 1999: 1907). This paper focuses on the dataflow programming environments utilized within the VPLs mentioned in the previous section.

Jill Larkin and Herbert Simon (1987), in their article titled, "A Diagram is (Sometimes) Worth a Thousand Words" describe forms of representation as constructive components of the knowledge itself:

...the fundamental difference between our diagrammatic and sentential representations is that the diagrammatic representation preserve explicit information about the topological and geometric relations among the components of the problem, while the sentential does not. A sentential representation may, of course, preserve other kinds of relations, for example temporal or logical sequence... (Larkin and Simon 1987: 66).

In control flow programming, the idea to represent algorithms in two dimensions is generally realized by flow charts which are used to explain the order of functions to be executed, whereas dataflow programming treats these charts as not only representational, but also generative. Johnston et al. (2004) describes dataflow in a different relationship with its representational medium, stating that "...the name dataflow comes from the conceptual notion that a program in a dataflow computer is a directed graph and that data flows between instructions along its arcs..." (Johnston et al. 2004: 1).

This description shifts the focus of flow charts from a process management to the evolution of data itself. While looking at the graph, it is expected to see the flow of information between the nodes via arcs, but not the sequential order of functions to be executed. Alan Davis and Robert Keller clarify the same idea from the opposite direction: "...this flow concept gives data flow languages the advantage of allowing program definitions to be represented exclusively by graphs..." (Davis and Keller 1982: 26).

The graphs mentioned above are mostly directed and acyclic diagrams, preferably similar to Petri Nets commonly called “dataflow diagrams” (DFD). This diagrammatic representation reveals a design continuum that creates a suitable environment in which to work directly on the evolution of design itself, rather than the secondary mechanisms: “the code” that generates designs. This alone introduces a different understanding for solution spaces in architecture and design. The concept of dataflow is effectively utilized within VPLs of architecture and design today, as they allow creating and manipulating formation processes via DFDs.

Tilak Agerwala and Arvind (1982) define two fundamental characteristics of the classic von Neumann model of computation that are distinctive from the dataflow model: the lack of a global updatable memory, and a single program counter. We will now provide a brief explanation of these two characteristics from the perspective of VPLs of design;

No Global Updatable Memory

...Professional programmers with years of experience in writing Fortran code have become very good at writing Fortran-like solutions to problems. The change to Algol, Cobol, Pascal, etc., is not a large conceptual step, in that the structural styles of these languages are not radically different from Fortran's. However, data flow languages require and support very different styles. Programmers trained only in conventional languages might be unwilling to try problem-solving techniques based on graphical or even functional program structures... (Davis and Keller 1982: 40).

In a design process driven by control flow scripting, designers are expected to create variable containers in order to use work with digital data. Generally these variable containers should be defined at the beginning of the code, and used with strict syntax rules of the language; there are therefore side-effects for the overall running of the algorithm. The design process is likely interrupted by an optimization, and/or debugging activity, and possibly becomes one of them literally. In the dataflow approach, designers are expected to deal with the evolution of data itself, but not the data containers. Hence a designer does not need to plan variable names; he/she directly controls how data streams should flow along the functions, transform into different forms, and fire different branches of the graph, further re-forming itself in order to achieve a desired solution space. A number, a coordinate, a shape, or any other source of data can be defined as a reference for another one, or transformed from/into geometric forms with the desired complexity. This formation process can be expanded or contracted by joining or cutting parts of the DFD, or creating clusters. As each node of the DFD is working independently without side-effects, the flow of data can easily be re-directed to another path to extend the exploration. This is why dataflow algorithms can easily be composed into larger ones, or manipulated in real-time to test different solution spaces. In design VPLs, this characteristic encourages designers to sketch directly on the geometric formation process, providing an opportunity to sketch parametric designs in real-time.

No Single Program Counter

...in dataflow programming, there is no locus of control, nothing corresponding to the program counter of a conventional sequential computer... the dataflow model describes computation in terms of locally controlled events; each event corresponds to the “firing” of an actor... in a dataflow execution, many actors may be ready to fire simultaneously (locally controlled by their operand availability) and thus these actors represent many asynchronous concurrent computation events... (Najjar et al. 1999: 1907).

Dataflow does not have a single program counter that manages the execution of algorithms in a sequential manner. It is described as a data-driven process continuously fired by interconnected events, re-calculating its branches only when any related input is changed by another event. This creates a naturally optimized workflow, enabling a real-time running. Designers do not hit a “run” button to see the result of an algorithm; instead, they manipulate an already running process, and see the outcomes of the actions immediately. As a result, a close interaction between designer and the subject becomes more possible, resembling a design cycle closer to a form of “reflection-in-action” (after Schön 1983). However the lack of a sequential program counter has its own disadvantages. Logically, by its nature an acyclic DFD cannot execute iterative or recursive events, where data is expected to feed back to itself. Mauro Mosconi and Marco Porta (2001) explain various iteration constructs for dataflow languages, its most addressed disadvantage compared to control flow counterparts.

Figure 1 shows an example of “dataflow analysis” (after Ackermann 1982), investigating which parts of a sequential algorithm can be executed concurrently within a dataflow program definition. In order to utilize the DFD in Fig. 1, a designer must focus on the availability of data streams and their potential concurrencies, rather than the variable containers (A, B, C, etc.) or their calculation sequences (1, 2, 3), while knowing that the DFD might “fire” any of the nodes as soon as their input values are fed.

Fig. 1 On the *left* A control flow algorithm including variables and a sequential execution order. After defining B and C, this sequence can be executed both in (1, 2, 3) and (1, 3, 2) orders. On the *right*; the same algorithm in a graphical representation. Reading from *left to right*, the DFD reveals that steps (2) and (3) could be executed concurrently

Fig. 2 Dataflow diagrams of contemporary design VPLs, reflecting the algorithm on Fig. 1. On the *left* Dynamo for Revit (Autodesk) On the *right* Grasshopper for Rhinoceros (McNeel & Associates)

VPLs of architecture and design mentioned in the previous section utilize DFDs (as seen in Fig. 2); similar to Yourdon/deMarco graph notation. Like in most of the DFDs, common characteristics of this notation are nodes and arcs that connect nodes. In a classic DFD, each node represents an instruction that manipulates its incoming data and outputs the result of its computation event. VPLs of design specifically utilize geometric creation and manipulation functions as nodes. On the other hand, the arcs represent paths of data that flow between the nodes. This construct allows non-sequential computation events to be modeled as multiple arcs that might be connected to multiple nodes. VPLs of architecture and design treat numbers, points, curves, surfaces, etc., as data types, thus the construction of a DFD becomes the construction of an associative formation process at the same time.

The hardware side of dataflow is not purely utilized in VPLs of design today; DFDs are mostly rendered into sequential executions in runtime. Although the computing paradigm of dataflow is a promising development in design scripting, there are serious challenges in its integration to the design education. The style of programming is effected by how VPLs manage data matching issues, and the final sequences of execution. This is why the integration of VPLs in design education is closely related with the pedagogical approaches that introduce design students to DFDs. The following section will describe four strategies to address these challenges.

Pedagogical Patterns for Dataflow Design

The term “pedagogical pattern” was introduced by a computer scientist, Joseph Bergin who describes the concept inspired from Christopher Alexander’s *A Pattern Language* (Alexander 1977). Bergin and the Pedagogical Patterns Project team define an aim to record and share the experiential knowledge of instructors in a compact form (Bergin et al. 2012) with the possibility to relate them as a “language” between different fields of education. In this description, a pattern is regarded as a problem solver, each time the problem recurs in a different context. This is why pedagogical patterns are mostly explained as proactions to common problems of education, generalizing the motivations but not the outcomes.

In the emerging field of design scripting, it is not possible to define a single body of knowledge regarding how to introduce concepts and methods of programming to designers. An instructor with good scripting skills is usually believed to be good at teaching, too. On the other hand, teaching programming to architects and designers require a number of parallel experiences in addition to a perspective about the designerly use of algorithms. This is why the pedagogical patterns of computer science do not always overlap with the educational challenges of design scripting.

The “Parametric Modeling” course that is the subject to this research was conducted in various universities between 2009 and 2012. It is conducted with graduate and undergraduate students of architecture, interior design and industrial design with no previous experience on any programming language. The course was based on the notion of learning-by-doing, a common strategy for the active construction of a design knowledge. It is suitable to define behaviors and patterns within a constructivist learning environment, where a course is generally divided into clusters and sub-clusters with clear aims and exercises. It was composed of short-term and contextually limited design exercises, aiming to introduce basic concepts of dataflow parametric design. These exercises were derived from the challenges of dataflow mentioned in the previous sections. As the number of experiments on these exercises increased, various repeating causal connections started to emerge. Below are the initial patterns derived from these exercises: “explication”, “kit of components”, “objectification”, and “re-generation”. Each pedagogical pattern is explained by a general context, the educational challenge, the exercise used to approach to the challenge, student works, and feedbacks results.

In a wider perspective, the course introduced in this study was a part of a curriculum supported by architectural geometry classes and computational design studios that focus on material studies and design tectonics. The experiments introduced in this paper include algorithmic form-finding exercises on a particular programming environment. Readers may refer to the author’s blog (<http://www.designcoding.net>) for information about some of the other courses of the curriculum.

Explication

Contextual background: Algorithms are generally seem to be complicated and puzzling to designers. They are believed to limit creativity, dominating designerly intentions with predefined components and forms. This point of view reveals the importance of controlling algorithmic tools just as effectively as any other design tool. In education, sketching with algorithms could be introduced to students by the explication of own design actions.

Pedagogical pattern: One of the most significant challenges about introducing dataflow is gaining a fundamental literacy in DFDs. Students should be able to conceptualize diagrams as associative process explications. In essence, DFDs could be introduced by recording sequences of a design activity as a graph.

Dataflow design exercise: After the introductory example about developing an associativity graph from a geometric modeling process (Fig. 3); students are asked to develop graphs for their own design processes. In this particular exercise, they

Fig. 3 Introductory in-class assignment for the explication of DFDs

Fig. 4 Above semi-regular surface tessellation (student: Derya Karaduman). Below an adaptive component design (student: Ayşen Duman)

are expected to design a parametric surface subdivision by using only native modeling commands. Each command they choose is represented by a node in the DFD, while the input object of the command (an output of a previous command) is represented by connections between these nodes. Finally, students finish their own parametric design workflows by hand-made DFDs, without using any VPL or scripting. Students Ayşen Duman and Derya Karaduman experimented with regular and semi-regular surface subdivisions (Fig. 4). First, they were asked to develop their designs by sketching them on computer. As they drew, they were encouraged to pay attention to the sequence of commands and parameters they chose to use. Then they developed explicit definitions by writing down all of their actions as a graph. Finally, they used these graphs to “rewind” their design processes, changing input parameters as well as the graph itself in order to explore variations.

Feedback results:

- This approach is effective in introducing the practical use of DFDs in a design process, without implementing a sophisticated syntax. Similar to the algorithm in Fig. 1, these graphs reveal that geometric formation sequences produce potentials of concurrency. Furthermore, they are useful in perceiving and controlling the effects of each action step on the overall design system. Students write and repeat their own diagrammatized algorithms, testing different results while altering the parameters or the connectivity of the process without any concern about syntax, performance or optimization.
- As the method does not utilize VPLs, it is possible to include iterative and recursive events (which are common deficiencies of VPLs), if students describe them on their DFDs. This shows the hand-made explication of algorithms are also effective in by-passing some of the technical issues of VPLs.
- This pedagogical pattern has limitations beyond the introduction of DFDs. Dataflow languages have more complicated issues that cannot be captured by the process recording explained above, such as data trees and non-geometric data types.

Kit of Components

Contextual background: Constructive reduction is a common strategy for studio exercises. It could be the reduction of design elements or context, such as used in the popular exercises of the “Nine-Square Grid” (Hejduk 1999: 23), or the “Bridge” (Candido 1989: 22). Mostly developed in the 1950s and made popular between 1970s and 1990s, the “kit of parts” exercises utilized reduction as a pedagogical pattern. According to Love (2004), the kit of parts problem deflected attention away from reasons for architecture by focusing on the basic principles of design as an autonomous discipline. These exercises mostly aim to develop a spatial reasoning by pure geometric relationships derived from the combinations of elements. In some examples, students are asked to set forth their own restrictions while in other cases the kit of parts is defined by the instructor. This strategy of generative design matches today’s design computing, exemplified by contemporary digital design exercises developed from the Nine Square Grid (Yazar 2009).

Pedagogical pattern: The pedagogical pattern called “kit of components” is motivated by the classic concept explained above. In a dataflow environment, constructing correct nodes and connections on a DFD requires deductive reasoning. Different connectivities of the same node set would result in different solution spaces. Therefore this pattern aims to explain the role of innovation in generating DFDs. Students are asked to derive potential solution spaces from a predefined DFD cluster.

Dataflow design exercise: In the particular exercise, a ready-made definition of a surface subdivision is given to students, asking them to manipulate it in order to meet a specific design requirement, for example a formal study for a roof covering (Fig. 5). Students are allowed to add or remove a limited number of nodes, or alter the DFD connectivity, while keeping the base cluster unchanged. Student Deniz

schools of basic design would support instructors of design scripting in developing methods.

Objectification

Contextual background: There is a common misconception in design scripting that complex algorithms are needed in order to explore diverse solutions. In contrast, diversity is revealed by simplicity in most cases. This idea would be implemented in design scripting by using a strategy mostly common to other fields of education; the use of direct or indirect metaphors to explain a concept.

Pedagogical pattern: The pedagogical pattern of objectification denotes the use of physical metaphors for introducing dataflow concepts. Data trees are one of the most addressed challenges in learning dataflow, as substantial amount of values can flow from a single path of the DFD. Matching correct data groups from different paths is a complicated subject to study especially in the beginnings of the education. Multiple input values connected to a node require a kind of ordering in order to decide which value from an input stream is matched with the other one. The data management procedure of Grasshopper uses data lists and data trees to overcome this issue. This makes data trees a significant subject to understand in order to use the particular VPL efficiently. It is especially important to organize data trees in VPLs in order to maintain exploration with them.

Dataflow design exercise: The objectified introduction of “data trees” can be realized by superimposing data matching structures to the geometric modeling process of trees. In this particular exercise, students are first introduced to the recursive modeling technique of the VPL. Then they are asked to create fractal tree shapes, in which each branch is to be organized properly within a data tree (Fig. 7). In fact, data trees were already organizing while students modeled different trees, as the metaphor naturally fit well with the intention.

Fig. 7 Above an introductory assignment utilizing data trees and recursion. Below “data tree” exercise (student: Saadet Sezer)

Feedback results:

- The fractal algorithms required a special recursion component (called Hoop-snake), although some students developed their own data trees without using it.
- The motivation of using direct metaphors in design exercises is based on the fact that the student's reasoning with software notions could be improved faster by matching them with more familiar phenomena.
- The exercise would also be tested by adding well-defined details to the assignment, such as parametric leaves. The objectification pattern could also be implemented by introducing other components of the particular VPL, such as graph functions and image processing methods.

Re-Generation

Contextual background: Built examples of contemporary parametric design are becoming increasingly documented, and even historical enough to track a kind of “heritage”. They can be regarded as single instances of mostly “hidden” generative algorithms. After the construction of material outputs, these algorithms are generally forgotten; although they represent significant values beyond any single physical instantiation. Types of input data, programming languages, uses of geometry and mathematics, and the general motivation of design process (genetic, agent-based, etc.) make these experiments valuable assets of the cultural heritage of design computing. But beyond such constructive elements, these algorithms reveal the “phase space”, all material and immaterial potentials of the visible instance. It is beyond the scope of this paper to address the conservation issues of a parametric design heritage, and while surveys and restorations of such algorithms are not common practices of today, they have been found to be useful as pedagogical patterns.

Pedagogical pattern: Recoding an existing parametric design simply by looking at its single instantiation reveals an ill-defined problem: The real aim of this pattern is not to reconstruct the unique algorithm but rather to make students develop their own conceptions over an existing relational system. While the subject of this pattern is chosen from a specific design intention, it helps students to see the potential parameters easily.

Dataflow design exercise: In the particular regeneration exercise conducted in 2012, students were asked to analyze a material system named SPEC (Fig. 8) designed by architect Nilüfer Kozikoğlu and professor Fulya Akipek of Yıldız Technical University (Kozikoğlu 2004). In their analyses, students extracted geometric proportions and formal associations from the material output, mostly by measuring, taking photos and drawing diagrams. After that, students developed various DFDs of the SPEC, generating variations by their own conceptions. The DFDs should be able to generate the original design, and an “animate” solution space of it.

Feedback results:

- Students are encouraged to derive the initial data from a physical context in order to recreate its algorithm. This reverses the common cycle of parametric

Fig. 8 *Above SPEC*; designed by Nilüfer Kozikoğlu and Fulya Akipek in 2004 (image: N. Kozikoğlu).
Below “Surveying SPEC” exercise (student: Can Görgün)

design routine, which generally starts on-screen, and finishes with material outputs.

- The exercise based on the regeneration pattern reveals that it might be a harder, or even impossible to develop the exact algorithm from only one instance; thus creating an opportunity for an open-ended educational setup. This exercise helped students to develop their own understandings on physical parameters such as material properties and joint details that could be re-generated.

Conclusions

This paper describes an experiment on the education of dataflow programming tools to design students. Technical properties of dataflow paradigm suits well with some aspects of designing as a kind of learning and “curiosity-driven production of new knowledge” (after Nowotny 2011). It might be predicted that future design tools (such as programming languages) will continue to re-orient design, while design will, and should continue to create new tools. This is why methods to introduce new tools should be experimented in design education, and should remain discussed and kept updated within academia. The exercises explained in this paper form general strategies called pedagogical patterns. These patterns do not present strict definitions for how to handle a dataflow integration to design scripting; but reveal the common challenges and show practical paths to overcome them. These patterns should also be considered as chaotic, in the sense that they are likely occurrences of infinite possibilities. Therefore, all individual feedbacks presented here are not subject to generalization. One pedagogical pattern could produce multiple design exercises, while one exercise could match multiple patterns. In essence, tracking and sharing pedagogical patterns and exercises contribute not only to the personal experiences of instructors, but to the academic field as well.

To sum up the general outcomes from the relationships between design, education and dataflow model of computing, the following challenges and discussions emerge:

- Design education in dataflow should stress the definition of a relationship; not a comparison, as *no programming paradigm can be enough to cover all aspects of designing*. Combined and rational use of different computing models would gain from all.
- Practical and pedagogical intentions of design computing do not always overlap. A powerful, fast and optimized algorithm is not necessarily educational. As the theories and methods of design computing are integrating into the beginnings of education, *this raises the need for a computation-based basic design education today*.
- Computational design sketching has found a practical path of development in dataflow, because of the possibility of real-time geometric manipulation guided by any source of digital data. The collective social network of design computing is pursuing methods that extend the uses of different forms of data in design. *One of the most significant consequences (or risks, or potentials) of a data-driven design idea is the reduction (or redefinition) of design into data, and design process into a “flow of data”.*

Acknowledgments Author would like to thank to graduate students Anıl Bayburtluoğlu, Ayşen Duman, Seyla Enbatan, Ebru Eryeli, Derya Karadağ, Sertaç Karsan, Sibel Kırca, Erdem Köymen, Gözde Küçüköğlu, Merve Ulucan and Deniz Yazıcı at Yıldız Technical University Faculty of Architecture, and undergraduate students Ahmed Onkaş, Burak Özaktaç, Can Görgün, Derya Yıldız, Elif Ustaoglu, Gizem Sümbül Salar, Kaan Kortuncan, Nurcan Bayrakçı, Saadet Sezer, Sacha Nokay, Yağız Özkan, Yeşim Yıkılmaz, and Yunus Emre Dokumacioğlu at İstanbul Bilgi University Faculty of Architecture, for their invaluable explorations and feedback reports. The experiments presented in this paper were realized at Computational Design Graduate Programme at Yıldız Technical University Faculty of Architecture in 2009, and Parametric Modeling undergraduate course at İstanbul Bilgi University Faculty of Architecture in 2012.

References

- Ackermann, William. 1982. Dataflow Languages. *IEEE Computer* 15(2): 15–25.
- Agerwala, Tilak, and Arvind. 1982. Dataflow Systems. *IEEE Computer* 15(2): 10–13.
- Alexander, Christopher. 1977. *A Pattern Language, Towns, Buildings, Construction*. Oxford: Oxford University Press.
- Bergin, Joseph et al., eds. 2012. *Pedagogical Patterns: Advice for Educators*. Joseph Bergin Software Tools.
- Candido, Anthony. 1989. Bridge. In: *Education of an Architect: The Irwin S.Chanin School of Architecture of the Cooper Union*, eds. E. Diller, D. Lewis and K. Shkapich, 22–23. New York: Rizzoli International Publications.
- Celani, Gabriela, and Carlos Eduardo Verzola Vaz. 2011. CAD Scripting and Visual Programming Languages for Implementing Computational Design Concepts: A Comparison from a Pedagogical Point of View. *International Journal of Architectural Computing* 10(1): 135.
- Çolakoğlu, Birgül, and Tuğrul Yazar. 2009. Designer as Casual Coder: Overview of an Experimental design Studio. In *27th eCAADe Conference Proceedings*, 449–454. http://cumincad.scix.net/cgi-bin/works/Show?ecaade2009_091. Accessed 1 Nov 2014.
- Davis, Alan, and Robert Keller. 1982. Dataflow Program Graphs. *IEEE Computer* 15(2): 26–41.

- Hejduk, John. 1999. The Nine-Square Problem, Education of an Architect: A Point of view. In *The Cooper Union School of Art & Architecture 1964–1971*, eds. U. Franzen, A.P. Gomez and K. Shkapich, 23–38. New York: The Monacelli Press.
- Johnston, Wesley, Paul Hanna, and Richard Millar. 2004. Advances in Dataflow Programming Languages. *ACM Computing Surveys* 36(1): 1–34.
- Kozikoğlu, Nilüfer and Fulya Akipek. 2004. SPEC: From Digital to Physical. <http://www.tuspa.net/spec/index2.htm>. Accessed 2 Oct 2014.
- Larkin, Jill, and Herbert Simon. 1987. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science* 11(1): 65–99.
- Leitão, António and Luís Santos. 2011. Programming Languages for Generative Design: Visual or Textual. In *Respecting Fragile Places*, 29th eCAADe Conference Proceedings, 549–557. http://cumincad.scix.net/cgi-bin/works/Show?ecaade2011_118. Accessed 1 Nov 2014.
- Love, Timothy. 2004. Kit-of-parts Conceptualism: Abstracting Architecture in the American Academy. *Harvard Design Magazine* 19: 1–5.
- Menzies, Tim. 1998. Evaluation issues for Visual Programming Languages. In Eleventh Workshop on Knowledge Acquisition, Modeling and Management. Reprint in *Handbook of Software Engineering and Knowledge Engineering*. Vol. 2, 2002. Preprint available at <http://menzies.us/pdf/00vp.pdf>. Accessed 2 Oct 2014.
- Mosconi, Mauro, and Marco Porta. 2001. Iteration Constructs in Data-Flow Visual Programming Languages. *Computer Languages* 26: 67–104.
- Myers, Brad. 1989. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing* 1(1): 97–123.
- Najjar, Walid, Edward Lee, and Guang Gao. 1999. Advances in the Dataflow Computation Model. *Parallel Computing* 25: 1907–1929.
- Nickerson, Jeffrey. 1994. Visual Programming. PhD diss., New York University.
- Nowotny, Helga. 2011. Foreword. In *The Routledge Companion to Research in the Arts*, eds. Michael Biggs and Henrik Karlsson, xxvi–xxvi. Abington: Routledge.
- Schön, Donald. 1983. *The Reflective Practitioner: How Professionals Think in Action*. New York: Basic Books.
- Sousa, Tiago Boldt. 2012. Dataflow Programming: Concept, Languages and Applications. In *7th Doctoral Symposium on Informatics Engineering*. Oporto, Portugal.
- Sutherland, William. 1966. The On-Line Graphical Specification of Computer Procedures. PhD diss., Massachusetts Institute of Technology, Cambridge, MA.
- Terzidis, Kostas. 2006. *Algorithmic Architecture*. Architectural Press.
- Uustalu, Tarmo, and Varmo Vene. 2005. The Essence of Dataflow Computing. In *Revised Selected Lectures from 1st Central European Functional Programming School (CEFP 2005, Budapest, July 2005)*, ed. Z. Horváth, 135–167. Heidelberg: Springer.
- Wurzer, Gabriel and Pak, Burak. 2012. Lawnmower: Designing A Web-Based Visual Programming Environment That Generates Code To Help Students Learn Textual Programming. In *30th eCAADe Conference Proceedings*, 655–663. http://cumincad.scix.net/cgi-bin/works/Show?ecaade2012_144. Accessed 1 Nov 2014.
- Yazar, Tuğrul. 2009. Digital Exercises in the Design Studio. PhD diss., Yıldız Technical University, Computer-aided Design Unit, İstanbul

Tuğrul Yazar received his B. Arch at Yıldız Technical University in 2001, and has gone forward with an academic career on the field of Computational Design. Along with graduate education, he worked as a teaching assistant at YTU Faculty of Architecture Computational Design Unit between 2002 and 2011. In the master thesis, he developed a tool for architectural education, regulating a specific building typology. This shaped his initial orientation as a researcher and educator on the relationship between computational methods and architecture. His Ph. D. study was a search for a deeper understanding on the integration of computational design pedagogy, specifically focusing on the evolution of beginnings in architectural education. Since 2011, he has been working as an assistant professor at İstanbul Bilgi University Faculty of Architecture, conducting computation-based basic design, parametric modelling, design computing and architectural geometry courses within the first year curriculum. Since 2011, he has maintained a digital diary at designcoding.net, presenting fragments of his academic studies on geometry, education, scripting, patterning and parametric design.