A CASE FOR ARCHITECTURAL COMPUTING

Computing Using Architectural Constructs

GANAPATHY MAHALINGAM
North Dakota State University, U.S.A.
Ganapathy.Mahalingam@ndsu.edu

Abstract. This paper is about the potential of architectural computing. Architectural computing is defined as computing that is done with computational structures that are based on architectural forms. An analysis of works of architecture reveals the embedded forms in the works of architecture. A uniform, connections-based representation of these architectural forms allows us to derive computational structures from them. These computational structures form the basis of architectural computing. In this paper a case is made for architectural computing, ideas are provided for how it could be done, and the benefits of architectural computing are briefly explored.

Keywords. Architectural computing: architectural programming language; intentional programming; connections-based paradigm.

1. Introduction

Researchers in the field of computer-aided architectural design have pondered the computability of design for the past 3 to 4 decades. While this inquiry may seem moot now, given that most design activities can be performed on the computer using various pieces of software, it has masked what can now be considered as a unique form of computing, architectural computing. Claude Shannon (1937), in his influential master's thesis, *A Symbolic Analysis of Relay and Switching Circuits*, literally founded modern digital computing by integrating Boolean algebra, binary arithmetic and electromechanical relays into an effective device to perform computations. What if we now recast the inherent devices of architecture as effective machines? Can we build architectural computers from them? What would these computers do and what would be their unique characteristics? What if we derive an architectural programming language from the operations of architectural design? Not too long ago, Charles Simonyi (1995) hailed the death of computer languages and the birth of intentional programming. What is the potential of architectural intentions when we consider them as effective devices? What are the kinds of 'logic gates' that we could derive from architectural constructs? Besides opening up the world of 'architectural computing,' this inquiry would make us reconsider the world of architecture with a renewed rigor. It is time now to make the case for architectural computing. This paper is an attempt to do that effectively.

2. Architectural Computing

What is architectural computing? Architectural computing is computing that is done with a computational structure that has as its basis an architectural form. Architectural forms are embodied in works of architecture. They are essentially intrinsic. They include the forms of

the building envelope, the forms of the structural system, the forms of the mechanical and plumbing systems, the forms of the circulation system, the forms of the electrical system, the forms of the life safety and communication systems, etc. These architectural forms are manifest in the finished works of architecture. They can be derived from the finished works of architecture by careful analysis. These are the manifest forms of architecture.

However, the process of creating a work of architecture has embedded in it various inherent devices as well. These are not immediately available from a cursory visual analysis. These include datums, proportional systems, ordering diagrams, etc. These inherent devices can also be represented in such a way that they can become the basis of computational devices. The challenge lies in the creative mapping of these inherent devices into computational structures.

In recent research a case has been made for the uniform representation of architecture (i.e., architectural forms) using a connections-based paradigm (Mahalingam, 2003). A case has also been made to derive computational structures from these connections-based representations of architecture (Mahalingam, 2007). Earlier a case was made for an architectural programming language (Mahalingam, 2000). These three approaches can now be integrated into a case to be made for architectural computing.

Why architectural computing? Computer scientists often talk about computer architectures, which refer to the organization of computational devices. Though the term used is architecture, these computational devices seldom approach the complexity of works of architecture in the built environment, viz. buildings. Building designs are the result of some of the most exacting neuronal processing in the brains of designers. The synthesis of building designs represents the complex structuring of our neuronal systems. It may be said that complex works of architecture reveal human neuronal underpinnings more accurately than any other cultural artifact that humans produce. Architectural computing is proposal to tap this neuronal richness that is manifest as complex architectural constructs. The first step in this process is to see if, at the heart of architectural creation, there is a programming language.

3. An Architectural Programming Language

This section of the paper is adopted from an earlier paper on the topic (Mahalingam, 2000). It is absolutely necessary to integrate it in this paper to make the case for architectural computing.

The potential success in developing a programming language for architectural design depends on a careful mapping of the fundamental operations in the creation of architectural designs onto a set of computable operations. A characterization of architectural design at a fundamental level is needed before a programming language can be defined to enable the creation of architectural designs. Architecture has been defined as the art and science of designing buildings and supervising their construction. The creation of a work of architecture is the result of a complex interaction of diverse processes. However, the complexity in the creation of an architectural design belies a set of simple, fundamental operations.

A programming language is defined by its syntax and semantics. The syntax of the language describes the rules for creating structures (programs) using the language, and the semantics of the language reveals the meaning of valid structures (programs) that can be created with the language. Of these, the syntax is formally represented. Examples of formal description systems for the syntax of a programming language are the Backus-Naur notation and syntax diagrams.

To create a programming language for architectural design, one has to define the starting symbol, terminal symbols, non-terminal symbols and production rules for the creation of architectural designs. This may seem a daunting task, but, if we realize that the fundamental entities in architecture consist of form and space, solids and voids, the definition of a language for architectural design becomes viable.

3.1 THE DEFINITION OF AN ARCHITECTURAL PROGRAMMING LANGUAGE

This section presents the definition of an architectural programming language, complete with the Backus-Naur form (BNF) for the language. The purpose of developing this language is to provide a tool to write programs that generate architectural designs when executed. A complete syntactical description of the language including its starting symbol, its non-terminal symbols, its terminal symbols, and its set of production rules is provided.

A complete syntactical description of a language is called a grammar. A grammar can be considered a tuple of the following elements:

Starting symbol (S)

Terminal symbols (T)

Non-terminal symbols (N)

Production rules (P)

The notation for a grammar is thus: G (S, T, N, P)

A language (L) based on a grammar is defined thus: L (G) = L (S, T, N, P)

The task of creating a programming language for architectural design starts with the definition of a grammar for the creation of architectural designs. Using the 4-tuple form for the definition of a grammar, G (S, T, N, P), architectural design can be mapped thus:

Starting symbol (S): Architectural form (f)

Terminal symbols (T): Solid polyhedron (p_s), Void polyhedron (p_v), Union (U), Difference (\)

Non-terminal symbols (N): Architectural form (f), architectural space (s)

Production rules (P):

f -> p_s | f U f | f \ s

s -> p_v | s U s

The union operation (U) has precedence over the difference operation (\) in the production rules. The vocabulary (V) of the grammar or language is defined as N U T, that is, the union of the non-terminal and terminal symbols. The use of the symbol * after V, N or T indicates all possible strings over the sets of V, N and T.

These production rules defined give rise to other production rules of the form:

f -> p_s U p_s

This production rule allows an architectural form to be created by unioning a solid polyhedron with another solid polyhedron.

f -> p_s U f

This production rule allows an architectural form to be created by unioning a solid polyhedron with another architectural form.

f -> p_s \ p_v

This production rule allows an architectural form to be created by differencing a void polyhedron from a solid polyhedron.

s -> p_v U p_v

This production rule allows an architectural space to be created by unioning a void polyhedron with another void polyhedron.

s -> p_v U s

This production rule allows an architectural space to be created by unioning a void polyhedron with another architectural space.

If you visualize the creation of an architectural design, an architect starts with an existing architectural form, the site of the design. The architect then synthesizes a new form by creating a solid polyhedron, combining solid polyhedra (material) or removing void polyhedra (empty space) from the solid polyhedra (material). The production rules defined to create architectural forms are both recursive and non-recursive. Since there are an infinite number of solid and void polyhedra, this grammar does not preclude any architectural form.

In this programming language, only the Boolean operators of union and difference are used. Now can we visualize an architectural design operation that creates, in essence, a different 'logic gate'?

The grammar presented above is context-free like most programming languages. The actual grammar to create specific types of architectural forms will be a refined version of this grammar. This grammar captures the essence of a real grammar that creates an architectural form. Since polyhedra are themselves complex entities, a nested grammar can be defined to generate polyhedra. This series of nested grammars can then be used to develop a comprehensive programming language for architectural design.

3.2 THE POTENTIAL OF AN ARCHITECTURAL PROGRAMMING LANGUAGE

Kalay (1989) calls computer models of real-world phenomena "languages of representation." What if this language of representation is a programming language? Symbols sets used in computer programming languages include the binary set (1,0) or the number set (1,2,3,4,5,6,7,8,9,0) or the English alphabet set (a,b,c,d…z). Such sets allow for programs to be written in an alphanumeric language. The traditional language of architectural design is graphical. Therefore, a programming language for architectural design should probably use graphical symbols instead of alphanumeric symbols. This would make an architectural programming language a visual programming language. What if the symbol sets in an architectural programming language are graphical? Can one then draw a program instead of writing one? The equivalent of a sentence in an alphanumeric programming language would be a drawing in the visual programming language. What are the problems or benefits related to checking the validity of a program if it is drawn using graphic symbols? Actually, the problems related to checking the validity of a program written in a visual programming language should be no different than syntax checking in an alphanumeric programming language, if the graphic elements directly correspond to alphanumeric elements.

In the grammar for an architectural programming language presented in this paper, if the alphanumeric symbols are replaced with graphics representing the polyhedra, then the string of alphanumeric symbols generated by the production rules has a graphical equivalent. The architectural programming language can generate different strings based on the production rules. These strings can then be converted into graphics by substitution. Each sentence in the language will then become a spatial composition. When the substitution is made, there may be invalid forms created by some of the production rules. This is because the alphanumeric symbols are not spatial. For example a void polyhedron that is larger than a solid polyhedron cannot be differenced from it. Similarly, two solid polyhedra that do not overlap cannot be unioned to create a single architectural form. A mechanism is needed for checking spatial parameters of the polyhedra when implementing the production rules.

Drawing an architectural design may not be essentially more complex than programming an architectural design except for the visual immediacy of the drawing and the unstructured (or very complexly structured, depending on your viewpoint) nature of the drawing process. If graphical symbols are used in the architectural programming language, then programming an architectural design can become another form of drawing, a shorthand graphical notation of the design that reveals its full visual form when the program is executed. Even symbols for operators in the architectural programming language can be given graphical equivalents. ***A drawing will then be a computer program.*** This will be possible if the sequence of elements and operations used to create the drawing is accessible in order to map it onto a program. A finished drawing on paper using traditional media does not have a record of the sequence of graphic elements and operations used to create it, but a computer-based drawing does! Computer-based drawings can then provide a computational medium for the generation of architectural designs in a completely different sense.

With a well-defined architectural programming language, architectural designs can be generated by executing programs written as you would with a general-purpose programming language like Smalltalk. Programs can then be written (drawn?) to generate programs that generate architectural forms. This can lead to a powerful form of automation in the creation of architectural designs.

4. Intentional Programming

Charles Simonyi, who used to be one of the chief software architects at Microsoft, has been working on what he calls Intentional Software. According to his team, Intentional Software simplifies software creation by separating the software contents in terms of their various domains from the implementation of the software and by enabling automatic regeneration of the software as the contents change (Simonyi, Christerson and Clifford, 2006). In traditional software development, programmers had to take tasks that were to be completed in the domain of an application, for example, the creation of a cuboid in 3D modeling software, and represent the process in a form that the computer could understand, that is, using a general purpose programming language. The program was written to facilitate execution on the computer and not to articulate the task being performed. This disconnection between a machine executable representation and an 'intentional' representation in performing a task is the gap that is being closed by Intentional Software.

In the implementation of Intentional Software, domain experts can work in parallel with programmers in their respective areas of expertise; and the repeated intermingling can be automated. Intentional Software is supported by a Domain Workbench tool where multiple domains can be defined, created, edited, transformed and integrated during software creation (Simonyi, Christerson and Clifford, 2006).

Domain experts first define domain schema, where terms of the domain code are defined. Domain experts then define the domain code using the Domain Workbench tool and a domain specific language. This domain code takes the form an intentional tree in its parse structure. The domain code can also be converted to other forms of notation such as a finite state machine diagram (see Figure 2). A generator then processes the domain code to generate target code that is executed on the computer. The target code is the software program or application that a user needs to perform a particular task. These domain schemas and codes are defined by the domain experts, for example, the architectural programming language described in this paper would be a high-level 'intentional' domain defined by a domain expert, in this case, the architect.

Using this domain definition many programs for the creation of various architectural designs could be generated (see Figure 1 for an overview of the Intentional Software system where the domain code for some of the production rules in the architectural programming language presented in this paper is shown in yellow). This is not always easy. In complex systems, the domain vocabulary, domain relationships and domain rules may not lend themselves to be easily mapped onto a programming language to generate the target code. However, this is not as difficult in the synthesis of architectural forms as the architectural programming language shows.

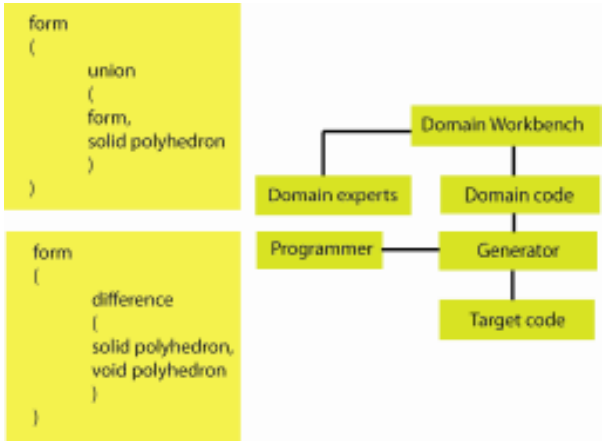

Figure 1. Overview of the Intentional Software system

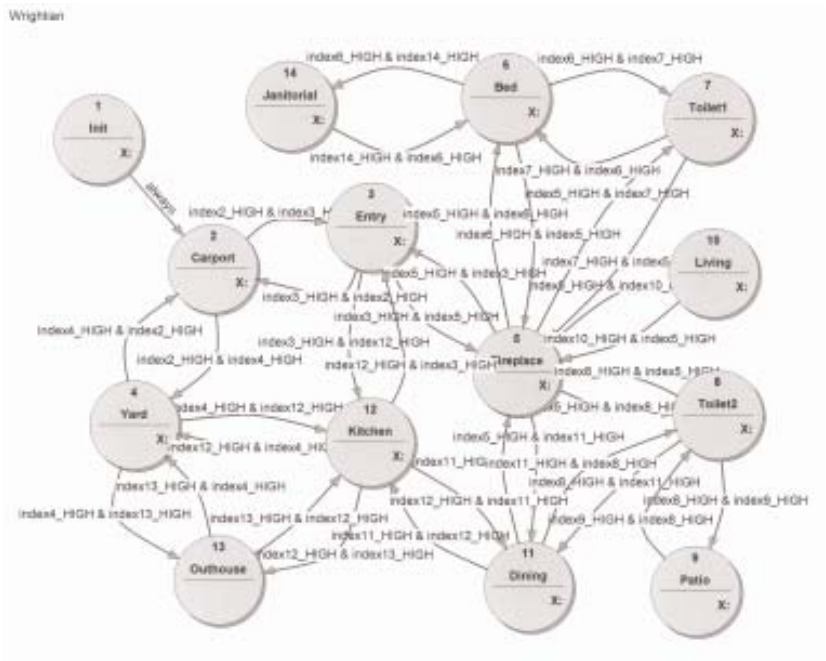

Figure 2. Domain code in a finite state machine notation for a system to predict fire spread in a Wrightian style house

Other key features of Intentional Software include a uniform representation of multiple interrelated domains, the ability to project the domains in multiple editable notations, and simple access for a program generator (Simonyi, Christerson and Clifford, 2006). This rich environment seeks to exploit domain schema and codes from innumerable sources for diverse applications. These domain schemas and codes serve as engines for the software development process. A critical role is played by the generator in this process. The domain schemas and codes only specify the data structure; the behavior of the data in the target code is generated by the generator using a process that translates the domain code into target code.

The world of architecture is rich in domain schema and domain codes. Hitherto, the world of architecture has not been seen as a valuable source of domain schema and domain codes for software design. With the implementation of Intentional Software, the opportunity has arisen for the use of architectural schema and architectural codes in the process of software design. Consider a structural design schema in architecture that can be used to create software for the design of the structural elements involved (Mahalingam, 1999). Consider a spatial layout schema, where the spatial layout and the interconnection of the spaces is the engine for a fire spread and control software for the building (Mahalingam, 2007) (see Figure 2). Consider a spatial design synthesis schema in the manner of a master architect, for example, software that can be used to design in the manner of a Palladio or a Wright (Hersey and Freedman, 1992). Consider a circulation system schema that can be used to create a program that automatically generates spatial layouts for buildings. The possibilities are endless. Where this research avenue can be taken is as limitless as the world of intentional forms.

5. Conclusion

Architectural computing is a new frontier. There is enough structure in the process of creating architectural designs and in architectural products to allow us to derive architectural programming languages and other complex computational structures from them. These programming languages and computational structures will initially inform the process of architectural design and expand its potential. They would then migrate to other disciplines and engage worlds such as engineering and biology. Architecture is a universal phenomenon. Form is its central ingredient. Architectural computing mobilizes computing with architectural forms. The future is wide open.

References

Hersey, G. and R. Freedman, *Possible Palladian Villas*, MIT Press, Cambridge, Massachusetts, 1992.

Kalay, Y. E. ***Modeling Objects and Environments.*** John Wiley & Sons, New York, 1989.

Mahalingam, G. *Discovering Computational Structures in Architecture*, CAAD Futures 2007 Conference, Sydney, Australia, July 2007.

Mahalingam, G. *Representing Architectural Design Using a Connections-based Paradigm*, ACADIA 2003 Conference, Indianapolis, Indiana, October, 2003.

Mahalingam, G. *Computing Architectural Designs Using An Architectural Programming Language*, eCAADe 2000 Conference, Weimar, Germany, June 2000.

Mahalingam, G. *A Parallel Processing Model for the Analysis and Design of Rectangular Frame Structures*, ACADIA 99 Conference, Snowbird, Utah, October 1999.

Shannon, C. *A Symbolic Analysis of Relay and Switching Circuits*, Master's Thesis, Massachusetts Institute of Technology, 1937.

Simonyi, C., M. Christerson and S. Clifford. *Intentional Software*. *OOPSLA'06* October 22-26, Portland, Oregon, USA, 2006.

Simonyi, C. *The Death of Computer Languages, The Birth of Intentional Programming*, Microsoft Technical Report, MSR-TR-95-52, Microsoft Research, 1995.

Teufel, B. ***Organization of Programming Languages.*** Springer-Verlag, New York, 1991.