Multi-Level Interaction in Parametric Design

Robert Aish[1] and Robert Woodbury[2]

[1] Bentley Systems, Incorporated
robert.aish@bentley.com
[2] Simon Fraser University, School of Interactive Arts and Technology, 14th Floor, Central City Tower, 13450 102 Avenue, Surrey, BC, CANADA, V3T 5X3
rw@sfu.ca,
WWW home page: http://www.siat.sfu.ca

Abstract. Parametric design systems model a design as a constrained collection of schemata. Designers work in such systems at two levels: definition of schemata and constraints; and search within a schema collection for meaningful instances. Propagation-based systems yield efficient algorithms that are complete within their domain, require explicit specification of a directed acyclic constraint graph and allow relatively simple debugging strategies based on antecedents and consequents. The requirement to order constraints appears to be useful in expressing specific designer intentions and in disambiguating interaction. A key feature of such systems in practice appears to be a need for multiple views onto the constraint model and simultaneous interaction across views. We describe one multiple-view structure, its development and refinement through a large group of architecture practitioners and its realization in the system Generative Components.

1 Architectural practice and parametric design

Conventional CAD systems focus design attention on the representation of the artifact being designed. Currently industry attention is on systems in which a designed artifact is represented parametrically, that is, the representation admits rapid change of design dimensions and structure. Parameterization increases complexity of both designer task and interface as designers must model not only the artifact being designed, but a conceptual structure that guides variation. Parameterization has both positive and negative task, outcome and perceptual consequences for designers. Positively, parameterization can enhance search for designs better adapted to context, can facilitate discovery of new forms and kinds of form-making, can reduce the time and effort required for change and reuse, and can yield better understandings of the conceptual structure of the artifact being designed. Negatively, parameterization may require additional effort, may increase complexity of local design decisions and increases the number of items to which attention must be paid in task completion.

Parametric modeling has become the basis for most mechanical CAD systems and now is beginning to emerge as a tool for architectural design. While there is a general appreciation of the concepts and advantages of parametric modeling,

application to projects at the scale and complexity of buildings raises important theoretical and practical issues. In a deep sense, parametric modelling is not new: building components have been adapted to context for centuries. What is new is the parallel development of fabrication technology that enables mass customization. Building components can be adapted to their context and parametric modelling can represent both context and adapted designs. In a design market partly driven by novelty, the resulting ability to envision and construct new architectural forms rewards firms having such expertise. There are relatively few such firms, most of which have had long experience and have built substantial reputations on distinctive form and construction. But many firms and students (future practitioners) are interested. The confluence of technology and interest appears as exploration in a new design space: architecture and its supporting technologies of parametric design and fabrication are experiencing both co-development and rapid change.

Emblematic of this change is the SmartGeometry group. It comprises senior practitioners, a CAD system developer and academics. Its website leads with the declaration:

> Architecture is fundamentally about relationships. Many of those relationships are geometric in nature or find a geometric expression. The SmartGeometry group has been created in the belief that Computer Aided Design lends itself to capturing the geometric relationships that form the foundation of architecture....The group is dedicated to educating the construction professions in the new skills which will be required to use these new systems effectively....The group conducts a series of schools and seminars where this new technology is explored in the context of highly experienced professionals. *(Minor verb tense changes made to the quotation.)* (`www.smartgeometry.org`)

To date, several such workshops have been conducted and workshop participants have continued to develop both designs and ideas for needed system support. The outcomes of the workshops and work that follows from them comprise designs, some intended for actual construction, some as explorations of design possibility. Figures 1 and 2 show work done by two workshop participants.

SmartGeometry works, as it must, through using (and developing) systems that support design relationships. Much of its work parallels and is influencing the development over the last eight years of a specific parametric design system Generative Components (GC) by Bentley Systems, Inc. The primary designer and developer of GC is Robert Aish. At the time of writing, GC was not on the commercial market and its development path lay outside of the Bentley product process. Its design and development have been extensively affected by members of the SmartGeometry group, the professional members of which have acted largely in addition to their professional roles inside their respective firms. Academics and graduate students have been involved in design, review and trial workshops to an extent unusual for a corporate project. The existence of a motivated independent user community and a relatively open and relatively

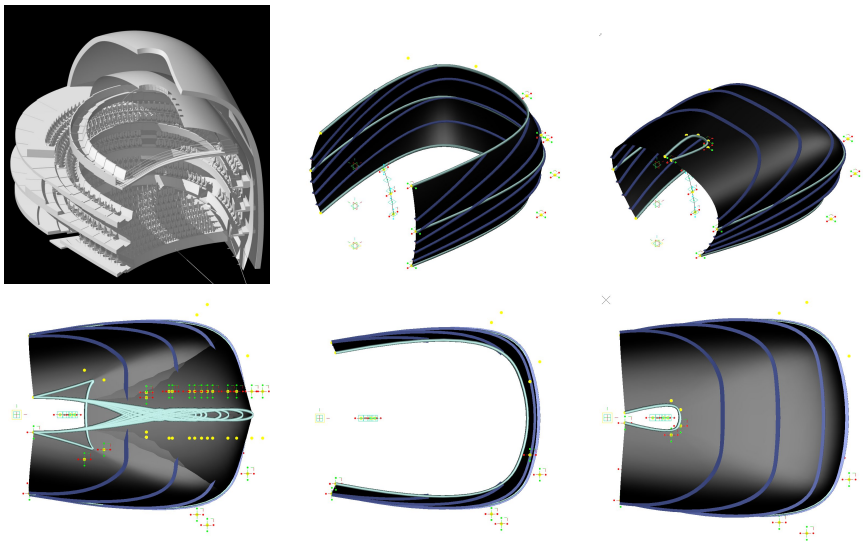


Fig. 1. Lars Hesselgren is a Senior Associate Partner and Director of R&D at KPF Architects. The images represent form explorations for a project currently (as of Spring 2005) under development at KPF. *(by permission of the author)*

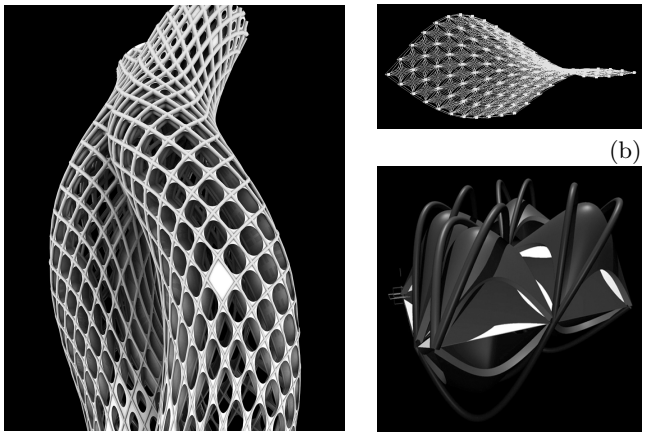


Fig. 2. Neri Oxman is a recent graduate of the Architectural Association in London and currently (as of Spring 2005) is at KPF Research in London. (a) Design prototype for a helical high-rise structure first modeled in a conventional 3D software package and post-rationalized in GC as a set of helical strands and tiling elements set in a cylindrical coordinate 3D space. The work represented was awarded a FEIDAD 2004 Design Merit Award. (b) & (c) Explorations in GC towards new graph nodes and an idiom of use employing a topologically-defined 'hosting environment' that dictates the position and geometrical articulation of any 4 point defined component to populate the mat. (c) A four-point based component developed to populate its hosting environment. *(by permission of the author)*

well-resourced system development process provides opportunities for early and frequent verification of design choices against actual need and for research connecting task to system at a scale largely not possible in research labs using systems built from scratch.

GC is a propagation-based system – implyting that part of a user's task is determining *how* particular relationships are processed in addition to specifying the relationships themselves. There are practical reasons for this. First propagation-based systems are efficient and sound. They are predictable, in that choices of what controls and what is controlling are required and are often important to designers. They are easily extensible at the local (node) level and provide multiple points of entry for more sophisticated extension. Although algorithmically simple, they are complex in use – the task of simultaneous model creation and design demands sophisticated language-level and user interfaces. The interface appears to be the principal technical obstacle to further practical use.

2 Propagation-based constraint systems

At the representation level, a propagation-based constraint system comprises an *acyclic directed graph* and two algorithms, one for *ordering* the graph and one for *propagating values* through the graph.

The nodes of the graph are schemata, that is, they are objects containing variables and constraints amongst the variables. Variables within a node are either *independent* or *dependent* and both variables and nodes are typed. Every type of node provides an efficient *update algorithm* for updating specific values for the dependent variables subject to the constraints and given values for the independent variables, as well as *display algorithms* for displaying the node symbolically and in 3D. The arcs of the directed graph denote uses of nodes or variables from within nodes as the independent variables in a node's constraints. An arc from node A to node B exists if a variable in A is identified with an independent variable in B. An incoming arc to a node *binds* one or more independent variables in the node. Graphs are constrained to be acyclic: any operation that introduces a cycle has undefined effect. Graphs may have arbitrarily many nodes with unbound independent variables; these are the *independent nodes* of the graph and their unbound independent variables are the *independent variables* of the graph. All other nodes and variables are *dependent*. The *antecedent* nodes of a node are those that bind its independent variables. The *consequent* nodes of a node are those that are bound by any of its variables.

A graph models a usually infinite collection of *instances*, each of which is determined by assigning values to the independent variables of the graph. Graphs are typically presented to the user through one or more of their instances.

To compute an instance, graphs are first ordered and then values are propagated through the graph. Both algorithms are theoretically simple and both are efficient. The ordering algorithm is topological ordering, which has worst case time complexity of $O(n+e)$ where n is the number of nodes and e is the number

of edges in the graph. The propagation algorithm is also linear, with complexity $O(n + e)$, presuming that the internal node algorithms are $O(1)$.

The representation is well-known as the basis for such common tools as spreadsheets, project management tools and dataflow programming languages. In design applications, the graphs tend to be large and the nodes represent schemata of arbitrary complexity. These issues are typically addressed through *constraint languages*, which provide tools for expressing node algorithms and for building complex graphs from nodes and less complex graphs.

Propagation-based systems are the most simple type of constraint system. The first CAD system was a constraint system. Sutherland's Sketchpad [Sut63] provided both a propagation-based mechanism and a simultaneous solver based on relaxation. It was the first report of a feature that became central to many constraint languages – the *merge operator* that combines two similar structures to a single structure governed by the union of the constraints on its arguments. *Constraint management systems*, for example, Borning's Delta Blue [SMFBB93] provide primitives and constraints that are not pre-bundled together and with which the user can overconstrain the system, but is required to give some value (or utility) for the resolution of different constraints. A constraint manager does not need access to the structure of the primitives or the constraints. Rather its algorithm aims to find a particular directed acyclic graph that resolves the most highly valued constraints. *Constraint solvers* represent primitives and constraints over them with algorithms specific to the class of constraints being solved, for instance, differential equations. *Constraint logic programming* systems integrate constraint satisfaction within a logic programming framework and use constraints to limit the search process of the logic program. Different constraint domains define different classes of constraint logic programming. For instance, in CLP($\mathbb{R}$), the constrained variables are real numbers and the solver is typically limited to linear relations with a delayed binding mechanism that admits solution for some non-linear problems. *Algebraic constraint* systems such as Magritte [Gos83] apply symbolic algebra either directly or to rewrite sub-graphs so that a graph suitable for propagation exists. *Constraint languages* such as ASCEND [PMW93] address the problem of building (and solving) large sets of equations in which the equation structure has significant regularity.

3 Parametric modeling as a task

A parametric model amplifies the effort of building a representation by providing an interpretation of a model as a typically infinite set of instances, each determined by a particular selection of values for the model's independent variables.

The parametric modeling task proceeds in concert with the task of creating a design. At the end of the process there exists both a graph structure and a specific instance that represents a concrete design. The main effect of separating graph and instance is to defer decisions. The graph embodies decisions about chosen relationships and defers computation of precise values (and in some cases structure) that depend on the relationships. For example, a graph representing

a roof structure may have the roof's support lines as its inputs and may produce different roofs designs depending on the location of the support lines. In contrast, non-parametric modeling systems tend to invert such a decision structure. The precise location of an object is required to model it at all and objects that depend on other objects must be explicitly modeled in precise context.

The cost of decision deferral is a higher level of abstraction in work. Users of parametric systems must explicitly develop relationships between objects and, at some level, code those into a node or graph. When a system does not support a particular relationship, it must be developed. Figure 3 shows an example of computing the shortest line between two skew lines. If such a relationship is not already part of a system it must be coded. If the needed sub-relationships are not supported, they too must be coded. Such work is necessary in practice. It turns out that the relationships that designers need to model their work comprise a set far too large and idiosyncratic to anticipate and provide as node types in a modelling environment. Designers using parametric systems must develop, and must be able to develop, relationships specific to the task at hand.

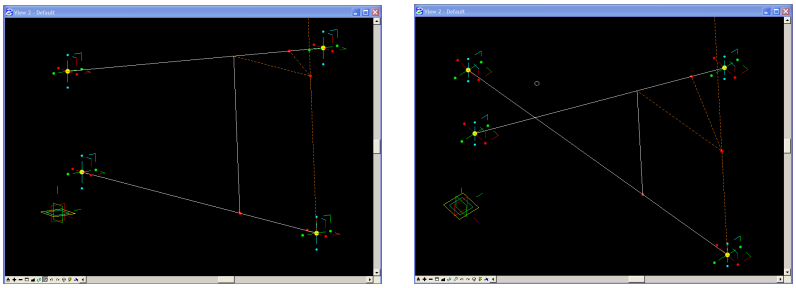
Another cost of parametric model is complexity of both representation and interface. At the representation level a designer must understand new concepts such as the graph itself, node compilation, intensionality and a set of mathematical ideas related to descriptive geometry and linear algebra. The present state of interfaces for parametric modeling systems dictates multiple, related interaction devices for different aspects of the representation. Using GC as an example, we demonstrate several novel aspects of task complexity and supporting interface.

4 Representational features

4.1 Object compilation

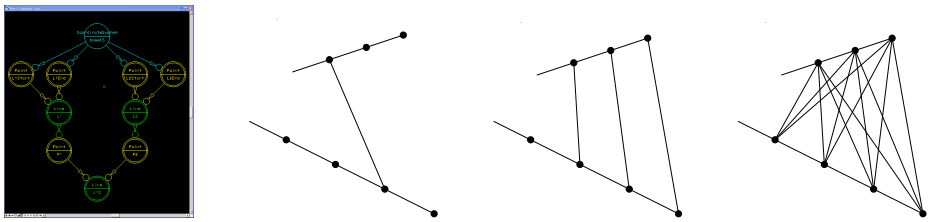
Nodes are typically defined by specifying independent and dependent variables and an update algorithm for the node. In GC the specification is a C# method where the method signature contains the independent variables, the method returns the dependent variables and the method itself specifies the update algorithm. The system uses code reflection to compile user-provided node specifications into update and display methods.

Conceptually, a node can be arbitrarily complex. To the ordering and propagation algorithms of a parametric design system a node provides independent and dependent variables and an update algorithm. A subgraph can be considered as a single node whose independent and dependent variables are those of the subgraph and whose algorithm is that of the global propagation algorithm applied to the subgraph. GC provides a second level of node compilation called *feature compilation* that, through code reflection, collapses a subgraph into a new object with its own update method. Figures 5(a) and 5(b) show the effect of compiling a graph representing a Bezier curve into a single node. Such compiled nodes can be reused in other contexts where, to a user, they represent a single modelling concept.



(a) (b)

Fig. 3. Two views of a shortest line segment between skew lines. The graph (not shown) comprises a cross product, three vector projections, a converse vector projection and three construction lines.



(a) (b) (c) (d)

Fig. 4. Single points, each expressed as a parametric point on a line. (a) A symbolic model representing a line between parametric points each on their own line. The same symbolic model represents (b), (c) and (d). A line between the two parametric points with single index values (1 & 2) for the parameters of its defining points. (c) Each of the defining points of the line has multiple parameter values. The number of lines is equal to the length of the shortest collection. (d) A line collection under the Cartesian product interpretation of a collection.

4.2 Objects as collections

In GC a node's independent variables may be either *singletons* or *collections*. A collection has the interpretation that each object in the collection specifies a node in and of itself. When multiple independent variables are collection-valued, collections propagate to dependent variables in two distinct ways as shown in Figure 4. The first produces a collection of objects, of size equal to the shortest of the input collections, by using the i^{th} value of each of the input collections as independent inputs. The second produces a collection by using the Cartesian product of the input collection as arguments. In both cases, the collection finds interpretation as a single node in the graph, while its elements are accessed through an array-indexing convention. The identification of singletons and collections supports a form of programming-by-example whereby the work done to create a single instance can be propagated to multiple instances simply by providing additional input arguments.

4.3 Intensionality

Intensionality means that two symbol structures can be identical in all aspects yet remain distinct. From a representational perspective a symbol is a declaration that there exists an object with the properties given in the symbol. There may exist many such objects and multiple symbols may refer to the same object. Changing the properties of a symbol does not affect the existence of the symbol – representationally it modifies the represented object, while maintaining object identity. Intensionality appears to be a necessary property in parametric modelling. In GC, it appears as *merge operator* and *variable update methods*. The merge operator is analogous to Sutherland's [Sut63] and essentially identical to Borning's [Bor81]. The design of update methods is modelled on Delta Blue [SMFBB93]. Update methods are uniquely distinguished by their signatures and a group of suitably designed update methods forms a clique through which a node with a method belonging to the clique may circulate, that is, users may change a node's update method from within its clique.

5 Representational views

In this section we sketch several interface views in GC. Such features have either been developed independently several times or are otherwise recurrent features of parametric design systems. It appears that multiple views and the attendant complexity in use is, at least at present, a necessary feature of parametric design systems. We include three views: 3D, symbolic graph and object (the latter being itself a composite of other views). Due to space, we omit a fourth view, namely the programmatic view through which elements may be directly programmed using the system's API.

5.1 3D interaction

Most demonstrations of parametric modelling systems are made in a *3D interactive view*. Such views comprise a 3D scene in which graphical objects are displayed. Graph nodes are represented by displaying the instance of the node associated with the current settings of the graph independent variables. The node display algorithm is determined by the node type: nodes of type *Point* display as a geometric point, nodes of type *Line* as a line. Some node types, for example *global variables,* do not have a direct representation in a 3D interactive view. Further, not all nodes need be displayed at once – it is common for nodes to have display properties such as *hidden* and *construction* that simplify a view while retaining the underlying model structure. Key node display algorithms are crafted with intent to support user intuitions about how a node is computed. For example, a point constrained to be on a curve may be displayed as a point plus a handle for dragging the point along the curve. Nodes in a 3D interactive view may be constrained by other nodes by identifying their independent variable with either nodes their contained variables.

Users engage with a 3D interactive view to define graphs, and to manipulate independent variables to find useful graph instances that may be concretely realized. Expressing even relatively simple ideas as a graph turns out to be labour intensive and saving labour has motivated a number of interaction devices.

5.2 Object view: names, expression entry, functions and scripting

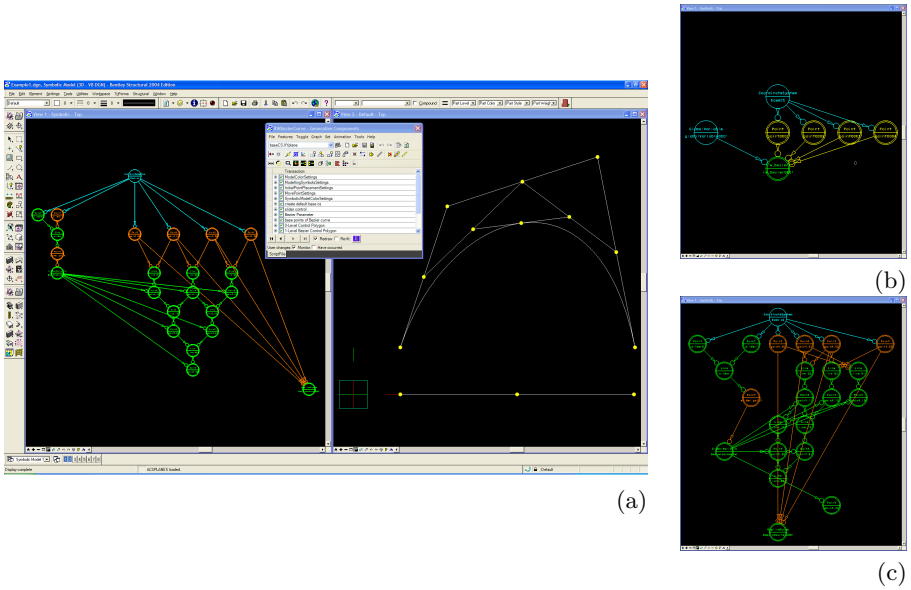
Individual variables within a node can be bound by identifying nodes or variables directly, through expressions over other nodes and variables, through user-provided functions or through statements in a high-level scripting language. Expressions, functions and scripts introduce the requirement that each node and each variable has a name. Names can be automatically generated but it turns out that naming and name re-factoring are critical aspects of model building. Variables may have nodes as their values and this introduces the need for *pathnames* from a node recursively through nodes held in its variables. A pathname from a node may trace *upstream* through antecedent nodes or *downstream* through consequent nodes in any combination. A node or variable thus may have multiple names either as a global name (nodes only) or a path beginning at some node. The hierarchy of direct identification, expressions, functions and scripting was not an *a priori* design decision. It emerged through sustained conversation with the SmartGeometry group (and others) which established the need for a learning scaffold from the GUI to a full procedural language. Designers move up this scaffold progressively, using new features as both need and skill develop. Each step in the hierarchy necessarily has its own interface.

Expressions and their attendant pathnames introduce a need to understand and interact with the structure of a graph. This is typically supported with *name completion* (not described here) and a *symbolic graph view*.

5.3 Symbolic graph view

A symbolic graph view presents a 2D or 3D visualization of the graph. Its chief uses are to select nodes for editing and as arguments to bind other nodes and variables, to explain the structure and expected behaviour of a graph, and as an aid to debugging. It is also important in selecting subgraphs for compilation into a node (see Section 4.1 above). Nodes may be represented in the symbolic graph view and not in a corresponding 3D interactive view, which makes the symbolic graph view the sole locus for interactive selection of such objects.

The layout of a symbolic graph greatly affects its utility. In a strictly parametric modeller the fact of data propagation establishes the useful convention that a graph may be laid out so that the visual flow never reverses. This is not enough: Figure 5 shows that a well laid out graph can add information critical to understanding a model. Such is important in reuse, which has subtasks of examining and editing graph structure. *Good* layout is task- and model-specific and a matter for authorial intent. Symbolic graph views have specific and limited utility yet occupy a considerable portion of available screen space.



(a) (b) (c)

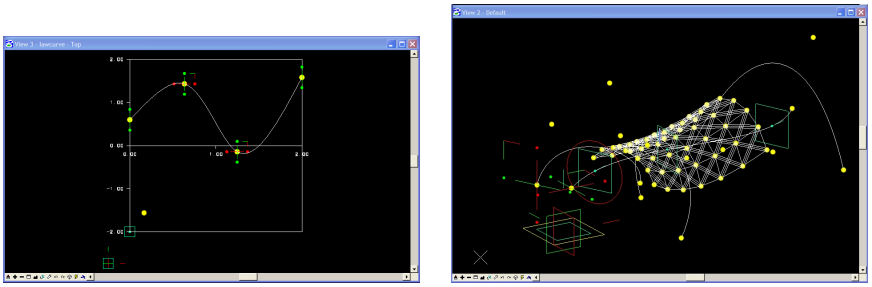
Fig. 5. (a) Symbolic graph (left) 3D interaction (right) views in GC. The small window in the centre is the general interface to GC. An order 4 Bezier curve implemented with the deCasteljau algorithm. The symbolic model has four conceptual parts: the *base coordinate system* at the top, the *slider control* on the left, the *systolic array* in the middle and a single node *Bezier curve* on the right. Each has a corresponding 3D display. As the point on the slider is moved from left to right, the point defining the Bezier curve traces out the curve. The nodes in the symbolic graph view have been manually laid out to correspond with both the deCasteljau algorithm and the layout of the control points in the 3D view. (b) A symbolic graph view of the Bezier curve graph after it has been compiled to a single node. The inputs are preserved in the form of four points and a single global variable for the curve parameter. (c) A symbolic graph view in which relatively minor changes to the layout have been made, resulting in a significant degradation of clarity.

6 Emergent features

The fact of independent variables that can be directly manipulated in an interface affords the emergent feature that a parametric design system can, to some extent, be its own interface. It turns out that users often spend a considerable portion of their effort building graphs that control other graphs. Such patterns of use have been the impetus to develop generic node types that are, in effect, elements of the GC user interface.

6.1 Law curves

Figure 6 demonstrate the so-called *law curves* in GC that support the mapping from an *independent control variable* to a *dependent control variable*. The law



(a) (b)

Fig. 6. (a) A *law curve frame*. The ordinate values of the curve become inputs for the parametric model shown in (b).

curve frame is nothing more than a node in the graph. It provides a curve controlled by independent variables and a collection of independent control variables that specify abscissa values at which the curve is to be sampled. Its output is an equal-sized collection of dependent control variables that specify the respective ordinate values of the control curve.

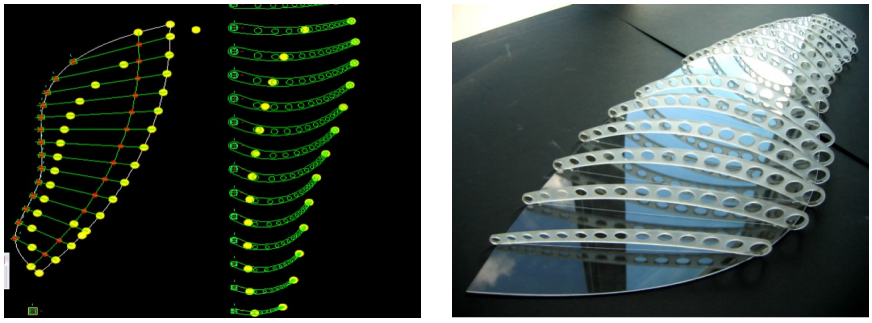
6.2 Fabrication planning

As mentioned in Section 1, a principal enabler for parametric design in practice has been the development of fabrication technology by which models, prototypes and entire components can be created by computer-controlled machinery (CNC). Fabrication planning can be brought into a parametric design system by nodes and graphs that transform a design to objects suitable for input to a CNC machine. The shape of objects to be fabricated need not be separate from the design process, but can become an aspect of the interaction with a model. Figure 7 shows windows to a GC model where a specialized node computes the layout for fabrication and an illustration of the resulting laser-cut model.

7 Summary

> The advantage of using Generative Components is that it helps me think about what I am doing. The disadvantage is that it forces me to so think. – *Lars Hesselgren*

The structure of design work using parametric systems remains poorly understood. Designers must simultaneously attend to both the specific, concrete design instance and the graph structure that captures its conceptual and mathematical structure. At the same time they must attend to the mulitfaceted design task at hand. We should neither under-estimate nor under-value the change to the structure of work and design process required. It is crucial to recognize that nothing can be created in a parametric system for which the designer has not



(a) (b)

Fig. 7. Wilson Chang is a graduate of the Master's program in Interactive Arts and Technology at Simon Fraser University and is currently (Spring 2005) in private practice. (a) A model comprising a layout for a structural system and the collection of structural members laid out on a plane. The structural members are parameterized by the layout. (b) A physical model whose components have been laser-cut using the structural member descriptions. *(by permission of the author)*

explicitly externalized the relevant conceptual and constructive structure. This runs counter to the often-deliberate cultivation of ambiguity that appears to be part of healthly design processes. Abstract concepts such as *intensionality* and *compilation* appear to be essential to effective use. Multiple views appear to be necessary. Such concepts, their attendant tasks and the complexity of working across views provide benefits and impose cognitive costs. Some of these costs may be overcome by clever graphical interventions such as guided layout of a symbolic model. However, any such move must be tested against a robust user community – it far too easy to work on a sophisticated solution to an unimportant problem. In contrast, problems such as the need for a hierarchy from the GUI to the algorithm demand both long conversation with sophisticated designer/users and apt solutions to the appropriate problems that emerge.

References

[Bor81] A. Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, Oct. 1981.

[Gos83] J. Gosling. *The Algebraic Manipulation of Constraints*. PhD thesis, Computer Science Department, Carnegie-Mellon University, May 1983.

[PMW93] P. Piela, R. McKelvey, and A. Westerberg. An introduction to the ascend modeling system: It's language and interactive environment. *Journal of Magagement information system*, 9(3):91–121, 1993.

[SMFBB93] M. Sannella, J. Maloney, B. N. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Softw., Pract. Exper.*, 23(5):529–566, 1993.

[Sut63] I.E. Sutherland. Sketchpad: A man-machine graphical communication system. Technical Report 296, MIT Lincoln Lab., 1963.