Introduction to GenerativeComponents

A parametric and associative design system for architecture, building engineering and digital fabrication.

by Robert Aish, Ph.D
Director of Research
Bentley Systems, Incorporated

BENTLEY

BENTLEY BUILDING | Solutions for Buildings and Facilities

Table of contents

Overview of GenerativeComponents

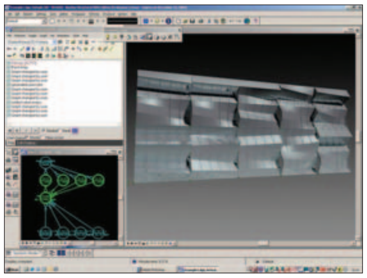

Image courtesy of Dan Strube

GenerativeComponents is a parametric and associative design system. Parametric design is equally applicable to all stages of the contemporary design: first, in concept formation and the exploration of building form; second, in the development and use of parametric and adaptive components within such a conceptual framework; third, in the control of the rapid prototyping and digital fabrication of these components; and fourth, to manage the extraction of conventional drawings, which automatically adapt to changing building configurations. Essentially, parametric design, as implemented with GenerativeComponents, opens new possibilities to efficiently explore alternative building forms and fabrication technologies, while at the same time addressing key issues in the efficient management of conventional design and documentation processes.

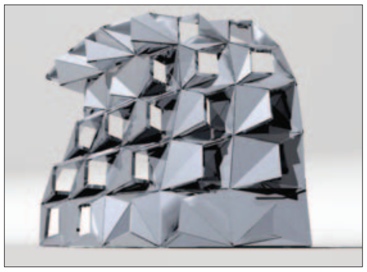

Image courtesy of Matthew Jogan

GenerativeComponents uses an advanced parametric engine to unify all aspects of design. It allows designers to build geometric models based on components and inter-component relationships. A component can be as simple as a line (a geometric primitive) or as complex as a double curves adaptive glazing panel arrayed over a complex building façade. A component can also be a numeric value (a single parameter) or a complex expression linking a number of parameters to the driving properties of geometric components. The system captures and graphically presents components and the abstract relationships between these components. This enables the GenerativeComponents to make explicit, not just geometry, but design intent as well.

Editing both the geometry and the inter-component relationships allows such models to be used to conveniently explore alternative design scenarios. Here, changes to the key geometry and parameters automatically ripple through the rest of the design models. This enables the designer to explore alternatives without having to manually rebuild the detail design model.

GenerativeComponents supports a unique combination of creative exploration and precise management control. Although designers are working graphically, based on intuition and experience in architectural design, the work is being captured in logical form, in what is effectively a program. GenerativeComponents provides an appropriate series of transitions between conventional design based on graphical interaction to a more algorithmic approach to design involving end-user scripting and software development.

GenerativeComponents is the design tool of choice for creative architects and engineers who are working at the intersection of design and computing, and who appreciate that good design emerges only by combining intuition and logic.

The distinguishing aspects of the GenerativeComponent system are that it:

- ***Captures design intent***

  The value of GenerativeComponents stems from the fact that designers can use the arrangements of the components and their interdependencies to capture their design intent in a way that is independent of any particular dimension or initial geometric configurations. This process of defining the model is valuable because it makes explicit what would otherwise be unrecorded in a non-parametric design tool.

- ***Allows designers to explore alternative design scenarios without manually rebuilding the design model***

  Once the model and relationships have been defined, the model can be used to experiment with alternative design approaches, eliminating the need to rebuild the complete model at every iteration. Thus, GenerativeComponents enables designers to concentrate on controlling their design strategy, rather than the mechanics of modeling.

- ***Create a user-defined component by modeling relationships, not programming***

  While each component has its own specific behavior, the dependencies between the components and inter-component relationships define the overall geometric behavior of the model. Designers can use such models to create their own components, without writing any scripts. Therefore, GenerativeComponent encourages customization and extension, so that the whole system can be tailored to the way an individual designer likes to work.

- ***Is delivered with a library of standard components from MicroStation core geometric functionality as well as with discipline-specific components from Bentley Architecture and Bentley Structural***

  GenerativeComponents is delivered with a rich set of predefined components, each of which implements a range of alternative behavior. This set of predefined components implements both abstract geometry and solid modeling functionality from the MicroStation platform, as well as application-specific components and functionality from the Bentley Structural and Bentley Architecture applications.

Illustrative GenerativeComponents scenarios[1]

Create a parametric building linked to conventional drawing extraction and reporting

In this scenario, a designer might construct a parametric building using only the predefined components from Bentley Structural and Bentley Architecture. By strategically building a parametric model, the designer can explore various alternative configurations—alternative floor-to-floor heights, column spacing, room layouts, and so on—without being slowed down by having to revise the building model with conventional direct manipulation commands.

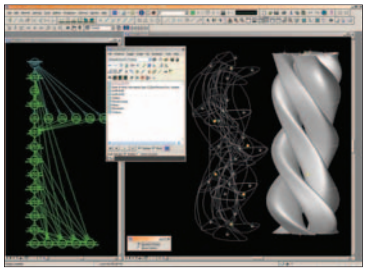
Image courtesy of Mark Cichy

Drawing extraction volumes can also be associated with the model, so that up-to-date and consistent documentation can be automatically generated as a byproduct of the modeling process.

Significantly, the parametric building modeled with GenerativeComponents is identical in data structure to an equivalent model created directly with Bentley Architecture or Bentley Structural, and can be analyzed by standard Bentley reporting tools. The difference between the GenerativeComponent model and the regular model is that the design intent has been explicitly captured in a way that is editable and re-executable.

This first scenario demonstrates that GenerativeComponents is completely at home with modeling conventional architecture and with the conventional documentation process.

Explore alternative unconventional building forms linked to digital fabrication

In this second scenario, the designer uses abstract geometric components to design a unique atrium roof. The designer constructs separate models to define the special components he requires. These components are designed to respond to the changing curvature and boundaries of the theoretical surface that defines the atrium roof.

The designer can test the validity of his or her design by unfolding the roof panels into a separate geometric model. This model can be used to prepare the geometry for digital fabrication. The unfolded model is completely associative to the design model, and updates as the design model is manipulated.

[1] A number of early adopter validation projects have been conducted with firms including Foster and Partners, Gehrey Partners, Morphosis, KPF, Arup, Grimshaw, and GLform. A beta release of GenerativeComponents is planned for the spring of 2005 and interested users can find more information about the Early Adopter program and availability at www.bentley.com/generativecomponents.

In this way the designer can explore variations in the overall concept while always being informed of the technical validity of his design from the perspective of digital fabrication.

This second scenario demonstrates that GenerativeComponents is completely at home with modeling unconventional architecture and with digital fabrication processes.

Combine both the convention and unconventional in the same parametric model

In the third scenario, the designer revisits the parametric building from the first scenario and decides to change the design concept into a courtyard configuration. He then uses the canopy designed in the second scenario to complete the enclosure. Because both the regular architectural components and the special roof components are all modeled in GenerativeComponents, they can be linked into a single associative parametric model.

The designer can now vary the building parameters and the atrium roof automatically responds to these changes.

The third scenario demonstrates that GenerativeComponents is completely at home combining conventional and unconventional architecture.

[1] *A number of early adopter validation projects have been conducted with firms including Foster and Partners, Gehrey Partners, Morphosis, KPF, Arup, Grimshaw, and GLform. A beta release of GenerativeComponents is planned for the spring of 2005 and interested users can find more information about the Early Adopter program and availability at www.bentley.com/generativecomponents.*

Conceptual Framework

Dependency and change propagation

GenerativeComponents is based on the dependency relationships between individual components, such that a change to an upstream component triggers consequential changes to all dependent or downstream components. Such a change is propagated through all the dependent components, and therefore is often referred to as a *change propagation mechanism*.

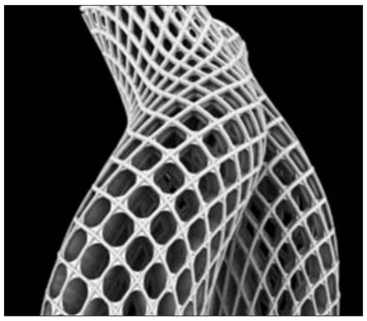
Image courtesy of Neri Oxman

Graph model

The components and the relationships between the components in GenerativeComponents form what is called a *graph*.

Previous computer-aided design systems have used hierarchical tree structures to represent relationships between components. For example, a model might consist of branching system of nested coordinate systems that define a series of geometric transformations. In this usage, each coordinate system (or graph node) can have only one upstream node (or parent coordinate system), but many downstream nodes (or child coordinate systems). Another use of the tree structure is found in mechanical modeling applications to represent the CSG (constructive solid geometry) relationships or feature trees in solid modeling applications. In this usage, each feature can have many upstream (or parent features), but results in a single node (or geometric body).

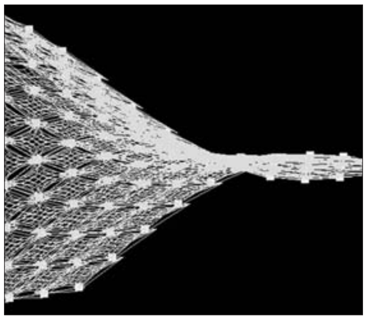
Image courtesy of Neri Oxman

While these uses of tree structures are interesting, they are completely geared to their respective theoretical bases (either hierarchical transforms or CSG operations). Essentially, this approach fails to capture the complexity inherent in design, where any component (or node) might be required to take inputs from many different upstream components and, in turn, influence many different downstream components.

The dependency graph used in GenerativeComponents is a generalization of the tree structure. Any tree structure can be represented as a dependency graph.

The GenerativeComponents graph is *directed*, because all the relationships are from the upstream or independent components to the downstream or dependent components. Many components in the middle of the graph have both inputs from other upstream component and, in turn, influence other downstream components. A component with no inputs is referred to as a *root component*. A component with no dependents is referred to as a *leaf component*.

In strict graph terminology, the components are nodes of the graph and the relationships between the components are the arcs of the graph. The relationship between two components is established by one of the input properties of the downstream components referring to an upstream component or to a property of an upstream component.

Typically, there are no cyclic paths through the graph in which the output of one component is either directly or indirectly related to one of it inputs. So we refer to this kind of graph as a *directed, acyclic graph*.

Symbolic model

One of the key advantages of computer systems is that they can be used to represent (and to graphically present) abstract or intangible relationships. Computers can, therefore, make explicit that which was previous tacit. GenerativeComponents uses a *symbolic model* to convey to the designers the relationships they have established, so that he can understand and manage this representation, and can use it to edit the model or share his ideas with colleagues.

Multiple alternative component behavior

Each component can have multiple alternative behaviors. For example, a point may be defined by Cartesian coordinates within a coordinate system or at a distance along a curve. These are not different types of point, but rather the same type implementing different behaviors, using different input relationships from other upstream components.

Each behavior is implemented with a unique *update method* (see below). The designer can switch update methods at will. Changing update methods would (in the point example) move the point, but it does not affect the way subsequent downstream components refer to that point.

Bi-directionality

The apparent limitation of a graph is that it is directed (i.e., it does not allow for bi-directional relations between components) and it is acyclic (i.e., it does not allow for cyclic paths between components).

We can argue that the majority of design conditions are essentially hierarchical. For example, the height of an individual stair riser is a function of the floor-to-floor height and possibly the overall height of the building and the number of floors. While a change in the riser height might affect the number of risers in a

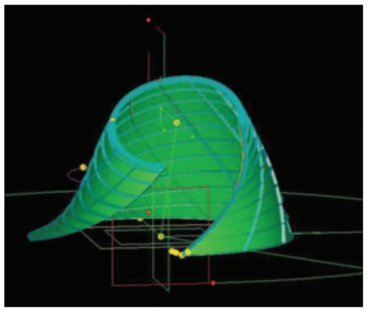
Image courtesy of Katherine Liapi

particular stair flight, it is unlikely that a designer would want to jack up the whole building when increasing the riser height.

Once we allow bi-directional relationships between components, the cognitive complexity explodes. Even with a reasonably-complex directed graph model, we are challenging the cognitive limits. So in the main, bi-directionality is not essential and is difficult to deal with.

There are, however, important aspects of design where design intent cannot be modeled as simple dependency relations. Therefore, bi-directionality is an important requirement that needs to be addressed.

Constraints

One technique to control the complexity of a bi-directional graph is to use a constraint manager to automatically switch update methods on components to reflect changing directions of dependency. This relies heavily on each component having a matched series of update methods, where each combination of relationship with the other components are implemented and can be systematically switched from being input to output.

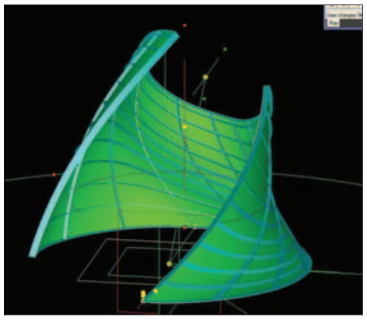
Image courtesy of Katherine Liapi

The requirement of matched series of update methods is very difficult to consistently satisfy. Even when it is satisfied, the resulting models are unpredictable. This approach is incredibly complex, but neither universal nor really useful.

Another more favorable technique is to use a *constraint solver*, or indeed a number of different constraint solvers tailored to specific type of modeling operations. Each constraint solver is restricted to a defined set of components and relationships. The designer can modify any of the components known to the solver, and the solver adjusts all the other components accordingly

This approach requires a different approach to defining components and relationships used in the previous graph dependency approach.

The graph dependency approach is often referred to as a *history-based system*, because it literally captures the sequence or history of the modeling operations of the designer, and faithfully re-executes these when an input is changed.

For example, the designer might define line A, then define line B as being parallel to an existing line, in this case line A. Moving line A causes line B to update to maintain itself parallel to line A. If line B is moved it has no effect on line A; indeed, as soon as the move is complete, the dependency system will move line

B back to its previous position, parallel to line A.

With a constraint solver, the designer defines both line A and line B, then declares that line A and line B are to be constrained to be parallel. The designer does not tell the system the order in which operations are to be applied, but declares a relationship (or many relationships) that the solver must satisfy. The designer can then move line A (and line B is moved to maintain the parallel constraint) or move line B (and line A is moved to maintain the parallel constraint).

It is envisaged that within an overall graph-based dependency model, the designer will be able to define (where appropriate) collections of components whose behavior will be controlled by constraint solvers.

Terminology

Component type / class definition

A *component type* (in software engineering terms, these are called class definition) defines all of the properties (characteristics) and *update methods* (functionality) for a particular type of geometric element or building component.

GenerativeComponents includes a library of abstract geometric components: Point, Line, Arc, BsplineCurve, BsplineSurface, Solids, and Operations (including Boolean operations) that create further components, based on existing components.

A user can create his own component types and add these to the library of available types. A new type can be defined from a graph-based dependency model (or a user-defined part of such a model) without any programming on the part of user.

Significantly, users can also define their own component types programmatically (using Visual Studio .NET to write their classes in C# or VB.NET), and/or by combining existing components into new component types. The latter process can be applied recursively, such that any arbitrarily complex project can be evolved in manageable stages.

Component instance

Whereas a component *type* defines a set of characteristics and behaviors, a component instance is an actual usage of that component type within a graph.

Each component instance has a unique *name* (assigned by the user) within the graph.

For example, a graph may include several instances of the Line component type; these instances may be *named line0001, line0002, myLine, yourLine,* and so on. Each of these is a unique instance of the Line type.

Update method

A component can comprise any number of update methods. An update method specifies the technique by which the component recalculates its location, size, and appearance, whenever such recalculation is necessary.

For example, consider a component that represents an arc (partial circle). An arc can be defined in several ways; for example:

- By three points on the arc
- By two points on the arc, and a center point
- By a center point, a radius, a start angle, and a sweep angle

The arc component has one update method for each of these ways of defining an arc.

At any given moment, only one update method is active within each component. However, the choice of active update method can change spontaneously, depending on current circumstances.

Property

A component can comprise any number of properties. A *property* is a characteristic of a component that controls its current state, and/or provides information about its current state. Each property must be of a defined type: either a built-in type such as *Double, Int, Bool*, or the type of a component. Each property must have a symbolic name.

A property is populated with an *expression*. A property has a value that is created by evaluating the expression.

For example, consider a component instance named myCircle that represents a circle. Its properties may include the following:

- **myCircle.CenterPoint** - The center point of the circle (of type point)
- **myCircle.Radius** - The radius of the circle (of type double)

The circle's location and size can be controlled by assigning new expressions to these properties; and these properties can be referenced within other components' property expressions.

Property expression

A *property expression* is a formula that defines a property's value.

For example, in the preceding example, the radius property may have any of the following as its expression:

- **myCircle.Radius = 3** - This expression happens to be the same as its resultant value, 3.
- **myCircle.Radius = 5+6*Sqrt(7)** - This expression is a mathematical formula, yielding the property value 20.8745…

- **myCircle.Radius = myLine.Length** - The value of this expression is the current value of the Length property of the component instance, myLine.
- **myCircle.Radius = 5+6*Sqrt(myLine.Length/yourRectangle[0].Height)** (Expressions can be arbitrarily complex.)

Each expression automatically recalculates whenever any of its dependent values change. For example, if myCircle.Radius's expression references myLine.Length, then myCircle will automatically change size whenever myLine's length changes.

Property value

At any given moment, a *property's value* is the result of the most recent recalculation of its expression.

Each property value change causes all dependent expressions to recalculate automatically; therefore, a change in one property value may trigger a ripple of changes throughout the entire graph.

When the user assigns a property, he is *assigning an expression*. When he references a property within another expression, he is *referencing* that property's value.

Types, update methods and properties

Each component type is characterized by a set of properties that are always guaranteed to be available for subsequent query. For example, an arc has a StartPoint, CenterPoint, EndPoint, Length, Radius and SweepAngle.

Different update methods use different properties as input to define the current configuration or behavior of the component. A property that may be used as input to one update methods, may be used as an output property of another update method. It is, therefore, easy to swap update methods, with the certainty that all dependent component using (or referencing) a particular property of that component will be valid. These conventions are strictly adhered to in the developed of the Bentley delivered component libraries.

Design Strategies

Model building

The opportunity offered by graph-based modeling systems such as GenerativeComponents is that these can be used to capture design intent as explicit dependency relationship between components. This process is neither accidental or incidental, but requires explicit operations on the part of the designer. The objective of GenerativeComponents is to support the design exploration both in conventional and unconventional building forms; therefore, the system avoids making any assumptions about the intent of the designer and avoids inferring any automatic response from the designer's actions.

With GenerativeComponents, it is possible to build extremely complex dependency models, with so many components and complex dependency relations that the designer can lose track of the intended roles or function of particular components. The GenerativeComponents system is quite capable of operating at this level of complexity; it is often the designer who is overwhelmed.

This is where the symbolic model is useful. If the symbolic model is starting to look too complex, it is telling the designer that it is time to think about structuring a single large and complex graph model into a series of smaller, less complex, and more manageable graph models. GenerativeComponents has the facilities to help implement this restructuring, because different parts of a single large model can be captured as different component definitions. The single complex model can then be reformulated as instances of these components. Each individual component encapsulates some essential behavior. The top level assembly model combines the behavior of all the different constituent components.

In a single, complex model the designer can be overwhelmed with all the functionality in an unstructured model. In the subsequent approach, he has carefully partitioned off the problem. He is either considering the behavior and functional implementation of a small part of the overall functionality, as captured in a single component definition, or he is considering how to use the functionality of this (and other components) in some higher level model. He has deliberately separated out implementation (how some functionality is achieved) from application (how some functionality is used). From a software engineering perspective, he has encapsulated this functionality for subsequent re-use. From a cognitive perspective, he has chunked a single complex task into a series of related tasks, each of tractable complexity.

This approach might change our understanding of what a component is. Conventionally, a component is a discrete physical entity: something that could be fabricated, that had geometric or material continuity and cohesion. In the future, a component might be defined on the basis of cognitive convenience, independent of physical partitioning.

User-defined components

At one level, creating a user-defined component from an existing graph model may seem a purely graphical operation, based on the designer's intuition and experience of architectural design. Although the designer is not programming, in fact his component is being captured as a program—one which is automatically generated by the GenerativeComponents application.

So although designers are not directly programming, they are nevertheless operating at the very interesting area of overlap between conventional design and software engineering. Of course, in this area of overlap, concepts and formalism from both disciplines converge and cross fertilize. In designing their new component, the designers need to plan not only its geometric configuration and behavior, but also how they or other designers expect to create instances of this component, or how they will use this component as the context for creating further components.

This means designers need to plan both the name and type of critical input properties required by this component. They may want to plan how the naming convention they adopt relates to other existing components or future components. Maybe this component creates not a single geometric primitive (such as a line or an arc), but an assembly of other components. In this case, they may also want to plan the type and name of these components. In addition, it may be desirable that the component report key measurements, such as area, or a count of the number of its constituent components. Again, designers need to plan what reporting properties are required, how these will be named, and what type they will be.

Essentially, designers of user-defined components are investing in the future ease of use or re-use of their work. They need to plan the scope and architecture of each component and its relationship to existing components or future components they are planning to create. Are they planning a few complex components, each of which span a range of behavior, or are they planning a larger number of simpler components, each specialized and focused on rigorously supporting a limited range of behavior? Are they building one component to be used within another component? Or are they building a set of interchangeable components, based on clearly defined combinatorial rules? In this approach, it is the exercise of these combinatorial rules that gives the desired variation in behavior, which might be far more powerful than a few complex components.

Conclusion

Using GenerativeComponents, designers have to operate simultaneously at different levels of granularity. They must understand the big picture, the complete system, but they must also identify suitable fault lines that can be component boundaries. They must understand how one component interfaces with it neighbors. They must define what information is exchanged across these boundaries and how each component contributes to the configuration and behavior of the overall system.

These are all important issues, and suggest that creating a really well-engineered and useful component library and using that library creatively requires a considerable sense of design. This is where well- established notions of systems engineering, object-oriented programming, and human computer interaction converge and complement emerging trends in architectural design.