A Complex-Based Building Information System

Norbert Paul
Institut für industrielle Bauproduktion (ifib),
University of Karlsruhe, Germany
http://www.ifib.uni-karlsruhe.de/de/mitarbeiter/Paul_Norbert/
norbert.paul@ifib.uni-karlsruhe.de

Volume modeling, finite element computation and spatial information systems have a common basis in topology, especially in the theory of complexes which are higher dimensional generalization of graphs. A complex is a topological space having a certain algebraic structure called "chain complex" and due to its similarity to real world buildings a special kind of complexes are even called "buildings" in mathematics. So it would be consequent to base a building information system upon this theory or at least try to do so.
Some manipulations and queries of such spaces can then be expressed by mappings which are similar to complex-morphisms and generalize the well known Euler operators. Other practical useful operations like assigning a detail to a spatial element ("refinement"), hiding details at lower scale views ("coarsening") or spatial versioning, however, need a somewhat different kind of mappings. This paper shows a simple relational database representation of a finite complex as a basis for a database backed building information system. The implementation of operations on such a relational complex in Java will also be shown.

Keywords: *Building information modeling; spatial modeling.*

Introduction

Computer based building information models either tend to be extremely complex like the IFC or COMBINE2 if they are to represe nt planning information such as the material elements are made of and the spatial relations between them, or the models tend to be very simple volume models with some texture attached to surfaces representing only the optical appearance of a building and lacking relevant planning information.

These simple volume models, however, carry an interesting spatial structure which is very similar to spatial relations in architecture. A polyhedron for example consists of a volume bounded by faces, edges and vertices connected by some spatial relations on them. A floor plan has a similar structure, where the rooms represent the faces, the walls, doors and windows stand for the edges and their connections on the floor plan replace the vertices. An architectural space, however, is tree-dimensional and its spatial structure therefore is somewhat richer than the surface of a polyhedron. This difference, however, is not essential: We always have a chain of relations "bound by" stepping down dimension by dimension. In the three-dimensional case we start with the volumes

which are bound by faces. Faces are bound by edges, edges are bound by vertices. So boundary relations behave the same fashion as in the two-dimensional case. In mathematics such a space is called a *complex*, and sometimes such a complex is even called building.

Care must be taken, however, that in the three-dimensional case an edge may connect more than two faces, and therefore data structures which rely on the fact, that each edge connects one left-hand-sided face with exactly one right-hand-sided face cannot be used. Most volume modelers and Geo-Information Systems (GIS), however, use such data structures like the Dual Independent Map Encoding (DIME) in GIS or its counterpart, the winged-edge-structures in volume modeling. Every finite complex has a simple straightforward realization in a relational database, which gives a common generalization of GIS, volume modeling and all other complex based data structures. So this database realization can be used as a common basis for all spatial data modeling including Building Information Modeling (BIM).

Complexes

We will start with a very simple example of an "architectural complex" as shown on figure 1 to present the basic mathematical concept of "complex".

We just take two cubes of equal size piled atop. The space occupied by these cubes is divided into volumes, faces, edges and vertices, the so-called cells which are drawn exploded on the right hand side of the figure. An element is called n-cell according to its dimension n. Now each cell of dimension n+1 (or n+1-cell) is circumscribed by n-cells. A volume (3-cell) is bound by faces and so on. Vertices (0-cells) have no boundary cells. The property of "circumscribing" a cell can formally be expressed by assigning a sign to each pair *(c, d)* of cells if *d* is a boundary cell of *c*. This sign expresses which side of *d* touches the cell *c*. Then we have a function side(*c*, *d*), defined as

$$+\mathrm{side}(c,d) := \begin{cases} +1 & \text{if } c \text{ is touched by the front side of } d \\ 0 & \text{if } c \text{ is not touched by } d \text{ at all} \\ -1 & \text{if } c \text{ is touched by the rear side of } d \end{cases} \quad (1)$$

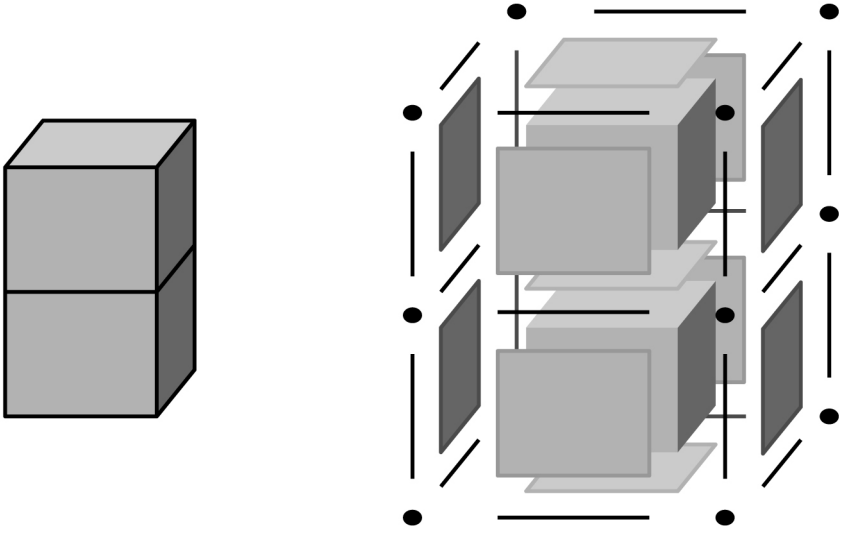

Figure 1
A simple example complex on the left hand side and its cells on the right hand side. Note that even where three faces are connected by one common edge, exactly two of them cover a common volume. The same is true for faces, edges and vertices: Exactly two of all the edges at a common vertex also have a common face.

A "degenerated" case would be, if *c* is touched by *d* from both sides like a protrusion of a wall which has the same room on either side. Then side(*c*, *d*) would also be zero, but we will discuss this later.

Now, if this side-function is properly defined then a cell d can be "flipped" by saying: "The front side of $+d$ shall be called the rear side of $-d$." which gives

$$+\,\mathrm{side}(+c,-d) = -\,\mathrm{side}(+c,+d) = +\,\mathrm{side}(-c,+d) \qquad (2)$$

as if one said he is constantly trapped in his hamster's cage because his hamster's name is Minus (and because interior side(–cage) = exterior side(cage)).

Now we only consider the boundary of one particular volume which we call *x*. Then the faces circumscribing that volume *x* meet at common edges and an edge connects exactly two of these faces – one to the left and one to the right. We first flip each boundary face such that the volume is only touched by front sides of these boundary cells. This flipping is easily done by replacing a face *a* by $\pm a$=side(*x*,*a*) · *a*. Then each edge *e* of $\pm a$ has an opposite face $\pm b$ at its opposite side. In this case side($\pm a$,*e*) also is the opposite sign than side($\pm b$,*e*) and so we get for the edge *e*: side($\pm a$,*e*) + side($\pm b$,*e*)=0. So if we "sum" up the edges of all signed boundaries of *x* they cancel to zero. Now with formula (2) we also have

$$+\,\mathrm{side}(\pm a,e) = \mathrm{side}(\mathrm{side}(x,a)\cdot a,e) = \mathrm{side}(x,a)\cdot\mathrm{side}(a,e) \qquad (3)$$

Now take a volume *x* and an edge *e*. Then the sum of all these expressions for all faces *a* returns zero because these faces circumscribe the volume. The property that a boundary's boundary is always zero is the *fundamental property* of a complex.

The side function above now sets up a square matrix M which is also called the incidence matrix and the fundamental property is equivalent to M multiplied to itself yielding the zero matrix. Usually a chain of linear functions from the volumes to the faces, from the faces to the edges and from the edges to the vertices is used instead of one single matrix. These functions are then called *boundary-operators* and the boundary of the boundary of each element must then be zero. As our definition differs somewhat from standard literature, we also give it a somewhat different name:

Definition (Finite Relational Complex) *We call a series of finite sets S_n,... ,S_0 together with a series of partial integer matrices*

$$\begin{aligned} M_n &:\subseteq S_n \times S_{n-1} \to \mathbb{Z} \\ &\vdots \\ M_1 &:\subseteq S_1 \times S_0 \to \mathbb{Z} \end{aligned}$$

such that $M_i \cdot M_{i-1}$ only contains mappings to zero a finite relational complex. *The* dimension *d of this complex is the greatest distance of two non-empty sets S_{d+i} and S_i.*

The attribute "relational" will become clear when we show the database realization of such a complex. A *partial* matrix is a matrix where some entries are left out. These can be padded by zeros until the result is a *total* matrix with no missing entries. Our above example is a relational complex of dimension 3. The set S_3 is the set of volumes, S_2 consists of the faces, S_1 contains all edges and S_0 the vertices.

The partial matrices are handy, because they save storage space and they provide us with two different kinds of zero: An explicitly stored zero value indicating the spatial relation bounded "at both sides" and a missing value side(*c*,*d*) = "undefined", if there is no spatial connection between them. Zero entries and missing entries must therefore not be confounded with.

Database realization

The term "relational complex" is used because we have a relational database implementation in mind. Each finite set *S* has a straightforward database schema *S[id:integer, ...]*, a table named *S* with a primary key attribute *id* as an identifier for each element and probably further attributes indicated by the ellipsis in the schema. If *S* is the set of the volume elements of a complex then we can name the table by "*Volume*". The faces can be stored in another database

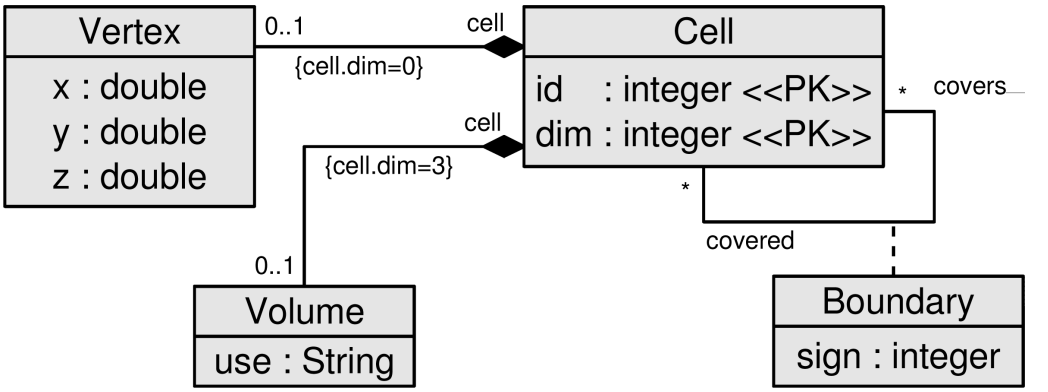

Figure 2
The UML Notation of the relational database schema of a complex with unbounded (complex) dimension. The stereotype <<PK>> stands for primary key attribute. In this example the geometric dimension of the vertices' space, however, is tree.

table *Face[id:integer, ...]* the same fashion. Then we have a partial matrix *FaceSide* of type *Volume×Face* with a database table schema

FaceSide[volume:Volume.id, face:Face.id, sign:integer]

where at least all matrix entries that are different to zero are stored and eventually some zero entries, too. So our complete database schema is

Volume[id:integer, …]
FaceSide[volume:Volume.id, face:Face.id, sign:integer
Face [id:integer, …]
EdgeSide[face:Face.id, edge:Edge.id, sign:integer]
Edge[id:integer, …]
VertexSide[edge:Edge.id, endPoint:Vertex.id, sign:integer]
Vertex[id:integer, x:double, y:double, z:double, …].

Note that this schema has a repeating pattern. Atop the volumes there can be appended hypervolumes of dimension 4 and even more. But this approach always fixes a static upper bound for the dimension of the complex. We can also define an even simpler database schema with unbound dimension if we collect up all the cells into one relation and all matrices into another relation by simply adding one attribute "*dim*" to the cells table and two additional dimension attributes "*dimA*" and "*dimB*" with two binary foreign keys to the boundary relation *Side*:

Cell[id, dim, …]
CellSide[idA, dimA, idB, dimB, sign :integer].

It can be shown, that this simple database schema is also an effective model for every finite topological space, not only for complexes. Additional Information can be attached to each cell by using relations such as *Volume* or *Vertex* like on figure 2. We see that the static version somehow reappears here.

In our first approach, geometric information is only attached to the vertices by simply specifying their coordinates in the three dimensional Euclidean space. Richer geometric information like spline specifications of edges and surfaces can also be attached to the cells.

Levels of detail

If there are different views at a common spatial situation and one of them contains more details than the other we call the less detailed plan the *sketch* of the other. Then each element of the detailed plan "comes from" an element in the sketch. This "comes from"-relation is a function: The bricks layer *b* in

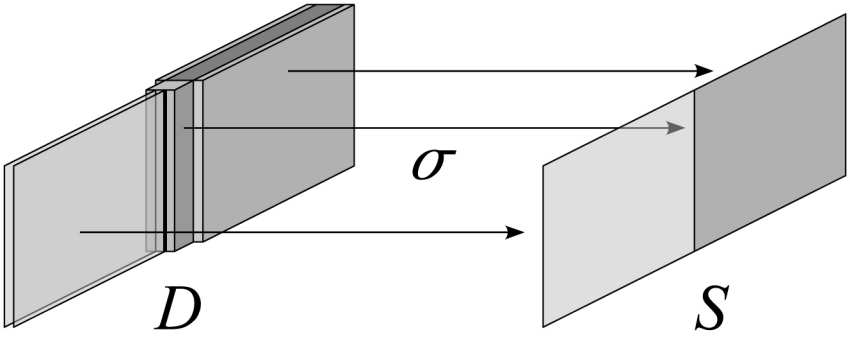

Figure 3
*A sketching function σ : **D** → **S** from a detailed space **D** to its sketch **S***

some multi-layered wall "comes from" some wall element *w* in the sketch. So we have a function $\sigma : D \rightarrow S$ (like "σketching") from the detail to the sketch which we call a *sketching function*. Figure 3 shows such a sketching function from a detailed joint between a multilayered wall and a double glazed window to a rough sketch of the same situation.

A sketching function, however, must maintain most of the spatial structures between both plans involved: It must not separate elements which are connected, hence be *continuous* from the topological space (D,T_R) to its sketch, the space (S,T_Q). The topologies T_R and T_Q are defined by the incidence matrices *R* and *Q* on the cells (Paul & Bradley 2005). On the other hand, the sketching function must not overdo pasting: Everything connected in the sketch must "detail into" something connected, too. The relation "details into" is the inverse image of "comes from". Continuous functions of such property are called *monotone* (Kuratowski 1968). Figure 4 gives an example of a monotone function σ and another continuous function τ which is not monotone as it overdoes pasting: It pastes interior and exterior together.

Implementation

Our work is still concerned with implementing cell complexes in Java. We have a MySQL database of the above schema with the specification of the location

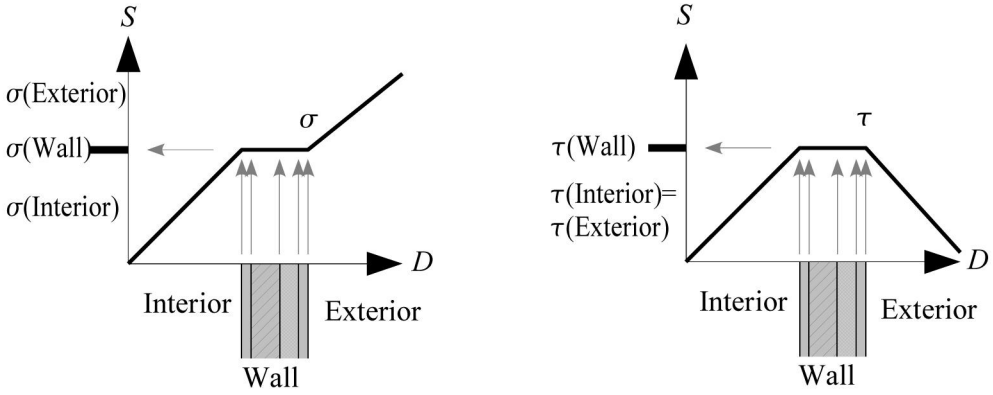

Figure 4
A monotone function σ from a detail D to a sketch S and a non-monotone function τ. The interior maps to the same element as the exterior, but their connecting wall is mapped to a different element.

of the vertices by attributes *x*, *y* and *z*. First we defined some helper types for the linear algebra we need. A linear combination such as "3 Apples – 1 Pie" is of type Map<Fruit,Integer> which maps "Apple" to the Integer 3 and "Pie" to –1 (always assuming the import "java.util.*;"). In fact we defined an interface

```
IntegerMap<T> extends Map<T,Integer>{...}
```

with the additional module operations such as addition, scalar multiplication and so on. Another type is Matrix<R,C>, an R×C-Matrix with row type R, column type C and entry type Integer which extends IntegerMap<Pair<R,C>> with the usual matrix operations. The type Pair<X,Y> is a trivial helper type, similar to Map.Entry<X,Y>.

Java Cell-Complex

Using the above helper types we declare a cell complex by a Java interface like

```
interface Complex<T extends Cell>{
  Complex<T> coComplex();

  Set<T> cellSet();
  IntegerMap<T> boundary(IntegerMap<T> chain);
  Matrix<T,T> getBoundary();
  IntegerMap<T> coBoundary(IntegerMap<T> chain);
  Matrix<T,T> getCoBoundary();

  boolean isEditable();
  Set<T> editableCellSet() throws
UnsupportedOperationException;
  Matrix<T,T> editableBoundary()
    throws UnsupportedOperationException;
  boolean addCell(T cell, IntegerMap<T> cellBoundary)
    throws IllegalArgumentException
        ,UnsupportedOperationException;
  boolean removeCell(T cell)
    throws IllegalStateException
        ,UnsupportedOperationException;
  // ...
}
```

The methods starting with "add" and "remove" are generalization of the so called Euler operators (Mäntylä 1988). The add-methods stand for the MAKE-operators and the remove-methods for the KILL-operators. They all throw an Unsupported-OperationException if the Complex is not editable. The add-methods may throw an IllegalArgumentException if the specified cellBoundary violates the fundamental property. The remove-methods throw an IllegalStateException if the cell which is to be removed is a boundary cell of some other cell.

We strictly tell zero values from null values and we have a helper class Integers where the ring operations multiplication, addition and subtraction are extended to null references, in a way that null is treated as a zero which is "even more zero than zero". The operations with null are defined as:

```
null+a == a == a+null    == Integers.add(a,null)
null*a == null == a*null == Integers.mul(a,null) != 0
-null == null            == Integers.minus(null)
```

Therefore we always use Java's Integer class instead of its primitive int type. Note that if a Java map does not contain a mapping from some object, the value returned for that object is null. So for some complex c, and two cells v, f of c the Java expression

```
c.boundary(v).value(f) != null
```

defines the relation which generates the topological space of the complex c. The set c.sellSet() together with this relation gives a topological database which is discussed in more detail in (Paul & Bradley 2005).

The "editable"-methods return modifiable views of the set and the (partial) boundary matrix on this set. Direct modification of them, however, is not allowed so they should be declared protected which, however, cannot be done in Java interfaces. Modifying the set and the matrix hides the underlying database or memory access and it can also be file access, network transmission, card punching or any other means of data storage that is available. Of course

storage of a matrix does not mean, that for each pair of cells there must be allocated memory space: "Thou shall store a null by de-allocation." Waste-of-memory-expressions like

```
new Integer[cells.size()][cells.size()]
```

can only be done in small fixtures for test cases. Our "normal" way of matrix implementation is a database table like the one above and the database access is hidden by appropriate implementation of the Set and Map interface.

Morphisms

In algebraic topology a complex-morphism is a family of linear functions between two complexes, one such function for each cells' dimension. This function takes linear combinations of n-cells (or IntegerMap<Cell>) in one complex to linear combinations of n-cells of the other complex respecting the boundary operator. This definition, however, seems not very useful for a building information system as it is both too strict and too lax. It is too strict on the one hand, because then a mapping from a volume (wall) to a face (sketch of that wall) would not be possible, because the wall's volume is a 3-cell whereas a face is of dimension two. On the other hand the definition is too unconstrained, because one cell can be mapped to a combination of many cells and so at the cells level we rather have a relation instead of function. We therefore define a new kind of morphism suitable to our needs: We adopt continuous mappings to complexes. We propose a definition of this notion where using Java declarations instead of mathematical formulae to demonstrate the strong link of our implementation to the underlying mathematics. The last two expressions are abbreviations like 'f(x)' for the quite lengthy correct Java expressions like 'f.value(x)' and the image of an IntegerMap under f is also still undefined:

Proposed definition (relational complex morphism II) *Let cx be a Complex<X> with Set<X>x=cx.cellSet() and Matrix<X,X>dx=cx.getBoundary() and, accordingly, let Set<Y>y and Matrix<Y,Y>dy be the cells and boundary of a Complex<Y>cy. A* relational complex morphism *is a continuous Map<X,Y>f, from cx to cy considered as topological spaces, together with a mapping IntegerMap<Y>iy if the image f(dx.row(x).mul(iy(f(x)))) equals dy.row(f(x)).mul(iy(f(x))), if zero and null is considered equal.*

The definition above differs from our original definition in (Paul & Bradley 2005, p. 47), so this is version II and possibly not the final version as we still work on that. The sketching function we have already mentioned is a special kind of a complex morphism. Another such morphism is the relation between a geometric realization of a complex and its rendering on the screen. Note that for every continuous function such an 'iy' always exists: the constant zero function.

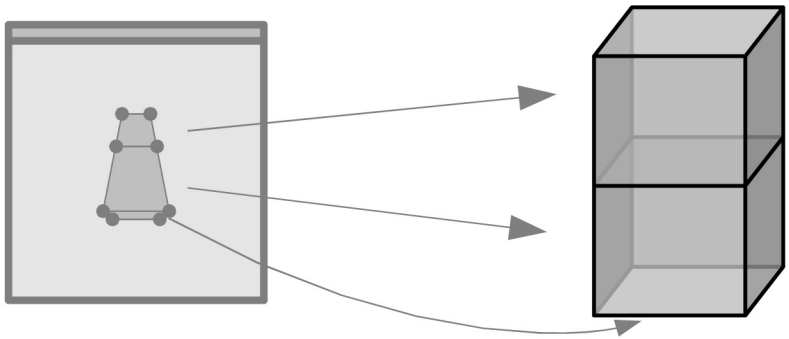

Figure 5
Two-dimensional awt. Shape objects (Area, Line2D, and – for the vertex bullets – Ellipse2D) as ShapeCell objects in a Complex<ShapeCell>. ShapeCell implements Cell and Shape. The morphism (arrows) is made up by the ShapeCells' references back to the rendered complex to the right side.

We have already seen that the morphisms' directions often are somewhat against intuition. A sketching function goes from the detailed plan onto the sketch – the inverse direction planning usually goes. The same is true with rendering. Its morphism goes from each shape cell on the screen back to the rendered cell as figure 5 illustrates. This is inverse to the rendering process.

Conclusion

Topology is an important foundation for spatial modeling. The comprehensive building information model is not finished yet but this work demonstrates a generalized concept for spaces at arbitrary dimension and levels of detail. So GIS, CAD and BIM can learn from mathematics. Other important aspects of spatial modeling like spatial versioning or using stock details from libraries or a user interface to construct complexes on a desktop do not fit into this paper and will be discussed elsewhere.

Acknowledgements

This work is part of the project KO-1488/8-2 *Architektonische Komplexe* funded by the German Research Foundation DFG.
(http://ww1.mathematik.uni-karlsruhe.de/~bradley/Arch/Archkomp/: May 2007).

References

Kuratowski, Kazimierz: 1968, Topology II, Acad.Pr., New York.

Mäntylä, Martti: 1988, An Introduction to Solid Modeling, Computer Science Press, Rockville.

Paul, Norbert and Bradley, Patrick Erik: 2005, Relationale Datenbanken für die Topologie architektonischer Räume, in Schley, Frank and Weber, Lars (eds.), Junge Wissenschaftler forschen / Forum Bauinformatik 2005, Lehrstuhl Bauinformatik, Brandenburgische Technische Universität Cottbus, Cottbus, pp. 42-51.