A multi-floor topology to geometry transformation procedure based on shape functions

G. Zimmermann
University of Kaiserslautern, Kaiserslautern, Germany

G. Suter
Department of Building Physics and Building Ecology, Vienna University of Technology, Vienna, Austria

ABSTRACT: Building floor plans are often outlined as schematics with zero wall thickness. Using shape functions, wall dimensions and circulation areas can be inserted and more detailed drawings generated. This topology-to-geometry transformation does not solve vertical constraints in multi-floor topologies, as for example the alignment of load bearing walls. The paper shows how dissection based rectangular floor plans can be modified by introducing space area extensions to resolve the vertical constraints. Penalty values can be introduced to control the distribution of area extensions. The algorithm is based on defining constraints between nodes of the dissection tree. In the presence of multiple constraints, several passes through the dissection tree may be necessary. The topology-to-geometry transformation is used in every pass to check if the constraints can be fulfilled. The algorithm is demonstrated with an example.

1 INTRODUCTION

The process of generating and modifying floor plans is an important design activity in both architectural and VLSI (Very Large Scale Integrated circuits) domains. A significant body of knowledge has been accumulated and various commercial systems have been developed to provide computational support. Liggett (2000) gives a survey of the approaches to automated layout of facilities. Typical layout systems have a modular structure and are based on a generate-and-test search paradigm (see, for example, Flemming 1989). Generators usually implement a floor plan representation based on the concept of dissection. Topological as well as geometric constraints on dimensions, areas, or aspect ratios of rectangles play an important role in filtering the number of alternative solutions down to a number that is manageable for human designers. Most of these constraints may be characterised as permanent, that is, they are either included explicitly in a space program or defined by a designer up-front, remaining relatively stable throughout layout synthesis.

In this paper, we describe how less static, ad-hoc constraints can be dealt with computationally. Also we are interested in how these could be maintained in the transition from schematic to detailed design. Such constraints are specific to particular floor plans and are informed by structural, energy use, or other performance considerations. For example, in multistorey buildings it is often desirable to align load-bearing walls vertically on top of each other (*vertical alignment constraint*), or to eliminate minor vertical recesses, which may occur due to discrepancies in floor layouts (*perimeter constraint*). These constraints and related conflicts arise first in schematic and later in detailed design. The transformation from schematic (topology) to detailed (geometry) design representations is non-trivial because the introduction of wall thickness information in detailed design typically causes multiple violations of the mentioned constraints. These might be satisfied in a schematic design, where walls are merely represented as lines.

The paper is organized as follows. First, we review related work, followed by a problem definition and an outline of an algorithm that addresses the problem. We illustrate the description of the algorithm with a concrete example and conclude with a discussion of the limitations of the proposed algorithm as well as potential enhancements.

2 RELATED WORK

The vertical alignment constraint introduced above is related to ‘nail’ constraints in graphics systems, where the location of an entity may be fixed. Harada et al. (1995) use nail constraints to fix the location of individual walls. Wall alignment across floors, however, is not considered. In the same work, numerical continuous constraint solution is used to determine exact locations, aspect ratios and areas of space rectangles. This approach may be computationally expensive, particularly in large, hierarchical layouts, where analytical area calculations need to be aggregated from leaf nodes upwards toward a root node. The computational effort for the derivation of such analytical equation systems appears realistic only in small problems or in situations limited to local search.

In contrast, shape functions could be useful for more complex layout problems because their derivation and solution is more scalable. In the context of VSLI design, a shape function has been defined "as the lower area bound of all possible rectangles of the cell" (Otten 1983, Zimmermann 1988). A more detailed definition of shape functions and a discussion of different types of shape functions in layout planning is given in Chapter 4.2. Shape functions have been implemented in VSLI and architectural design environments (Zimmermann 1988, Zimmermann and Suter 2004).

In stacking algorithms, the shapes associated with allocating an activity to a floor are usually not considered. Bozer et al. (1994) introduce an algorithm which generates multi-floor layouts. Among the known algorithms for stacking or multi-floor layout, none appears to address the types of constraints mentioned and the consistency problems that occur during the transformation from schematic to detailed design.

In the context of performance-based bulding design, Suter and Mahdavi (2004) describe a procedure that is based on space-boundary offset parameters to control the mapping from schematic to detailed design representations. In that work, however, space area constraints are not considered.

In summary, the work presented in this paper should be viewed as complementary to these related efforts. We believe that the proposed topology to geometry transformation algorithm could be a useful enhancement to automated layout generators or building editors for performance-based design. For conceptual clarity, however, we will explain the algorithm as part of a manual layout generation process.

3 PROBLEM DEFINITION

3.1 Terminology

We limit ourselves to orthogonal buildings. L-, U-shaped buildings or other shapes we complement by open spaces to achieve rectangular shapes. Therefore, for the rest of the paper we will assume rectangular buildings.

For floor planning purposes we distinguish two models with different resolution: the *schematic model* describes the *topology* of the spaces of a building with true net areas. The *detailed model* represents spaces with true wall and other dimensions. Geometric representations of schematic models as for example Figure 2 show dimensions for visualization purposes and give an idea of a possible geometry. By changing the aspect ratio, different geometries can be derived from one topology.

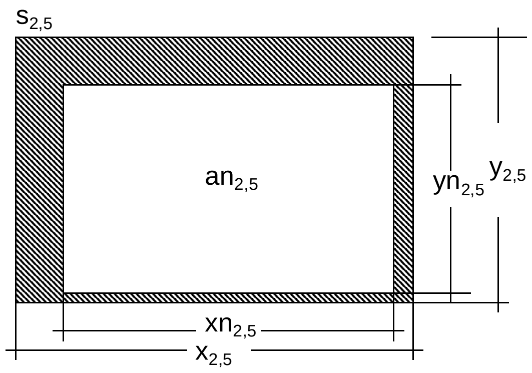

Figure 1. Single room dimensions of a detailed model.

We use an *x*-*y*-*z* coordinate system with *z* as vertical axis. *a* expresses floor area. Suffix *n* denotes net dimensions. Suffix *p* denotes space program dimensions. The space program is a list of all *ap* of net activity spaces plus circulation and other general spaces. The latter spaces are typically not defined by area but by one or two horizontal dimensions. In Chapter 4.2 possible differences between *an* and *ap* are explained.

All dimensions have two indices *i*, *j*. *i* denotes major *sections*, e.g. wings, storeys, vertical shafts, storey, *j* a *space* in a section. Figure 1 shows a single room example.

3.2 Partitioning

To generate a schematic model of a building, we interactively partition the whole space of the building recursively by horizontal or vertical *cut-sheets*. We also call this action a *cut*. The cut-sheets are denoted by the direction of the normal vector. The vertical sheets are denoted *x* and *y*, the horizontal *z*. Each cut-sheet dissects a space s_a into two spaces s_b and s_c with

$$an_a = an_b + an_c \tag{1}$$

We call s_b and s_c siblings of parent s_a. The two siblings have a positional relation to each other, depending on the cut-sheet direction. The following list shows the possible positions with abbreviations in parenthesis:

x:	left(l)	right(r)
y:	front(f)	back(b)
z:	down(d)	up(u)

Partitioning continues until all activity spaces of the space program are positioned. Through careful planning of the positions and directions of the cut-sheets, floor plans for all storeys can be achieved, which even without wall dimensions give a good impression of the final layout. Correct net areas for all spaces can be entered into the dissection tool we have described in Zimmermann & Suter 2004, creating a correct topology as starting point for the shape-function based geometry transformation.

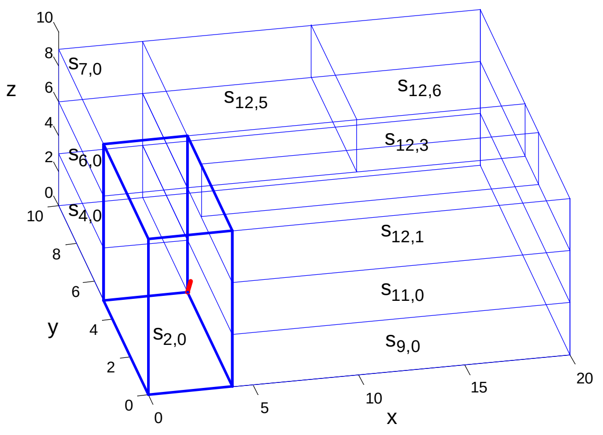

Figure 2. Topology example of a building space $s_{0,0}$ with 3 storeys and a staircase shaft $s_{2,0}$. The top storey $s_{12,0}$ is partitioned into a hallway $s_{12,3}$ and 3 rooms $s_{12,1}$, $s_{12,5}$, and $s_{12,6}$

The partitioning process leads to a binary *dissection tree* representation. Each node in the tree represents a space. We distinguish internal and leaf nodes. Internal nodes represent abstract spaces that are composed of other spaces. Leaf nodes represent spaces required by the space program. Each internal node is also uniquely related to a cut-sheet. Leaf nodes have no cut-sheets. Vertices represent parent-sibling relations. If we denote the nodes in the graphical representation by the positions from the above list, as shown in Figure 3, the cut-sheet direction of the parent nodes can be derived uniquely. A node identifier denotes the space as well as the associated cut-sheet. Figure 3 gives an example. This tree is the basis for all further procedures.

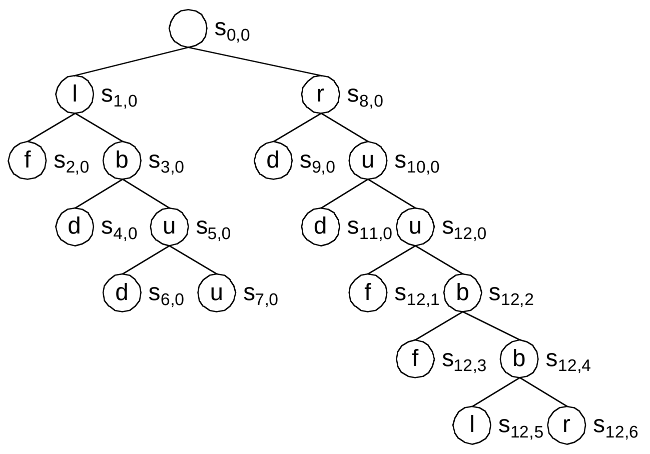

Figure 3. Dissection tree example of the building example in Figure 2

3.3 Constraints

Floor planning of multi-floor buildings may be performed floor by floor and by finally matching the layouts where necessary (see Figure 6). We call these necessary matches *vertical constraints*. Horizontal constraints are defined within one floor and are resolved during the topology to geometry transformation (see Chapter 4.2). We distinguish three types of vertical constraints:

The first constraint is caused by the need to align the outer walls of all storeys of a typical building. This means that the perimeters of different storeys have to have the same dimensions. We call this the *perimeter dimension constraint*. If we assume that during floor planning the total area of a storey is determined by the space program for this storey and by the additional areas for walls and other structural objects, the resulting perimeter dimensions will differ from storey to storey, even if a carefully balanced partitioning of the total space program into programs for each storey has been performed. What looks perfectly aligned in Figure 2 will change when the storey spaces $s_{9,0}$, $s_{11,0}$, and $s_{12,0}$ are partitioned and wall dimensions inserted. In this example the constraint can be expressed as:

$$\begin{aligned} x_{9,0} &= x_{11,0} = x_{12,0} \\ y_{9,0} &= y_{11,0} = y_{12,0} \end{aligned} \qquad (2)$$

A similar constraint could be expressed between the three storeys to the left of the example building.

The *wall alignment constraint* stems from the necessity to align load bearing and other internal walls between storeys. Typically the dimensions of such spaces has to be the same in all storeys. In the example this could mean that between the storeys on the left and on the right load-bearing walls exist. The alignment constraint would be expressed as:

$$x_{4,0} = x_{6,0} = x_{7,0} \qquad (3)$$

Shaft constraints guarantee the vertical alignment of shafts for staircases, elevators, chimneys, or utility networks. In the example in Figure 2 the alignment is automatically guaranteed by making the staircase a leaf-space $s_{2,0}$. In other cases this constraint has to be expressed by equations.

It should be pointed out that in all constraint examples dimensions of internal nodes or spaces have been used. The same could be expressed by leaf node dimensions, but the number of equations would increase drastically for most constraints. This is an advantage of the dissection approach and the resulting tree representation. In the following, we outline an algorithm for topology to geometry transformation. The algorithm is especially suited for configurations with floors separated by single cut sheets, such as the configuration shown in Figure 6.

4 SOLUTION OUTLINE

4.1 *Idea and basic equations*

If the space program would be strictly enforced, the constraints may not always be fulfilled. Some slack is necessary to enlarge or shrink total floor areas or other dimensions. For the purpose of this work we assume that the space program defines minimal net areas or dimensions that have to be fulfilled in any case. On the other hand, spaces can be enlarged if necessary. This will increase the cost but also the utility. We assume that the cost will exceed the utility increase, but this relation will depend on the activity of a space. We will introduce penalty factors p for leaf spaces to control the increase. In general, the total area increase should be minimized.

The basic idea is best explained with the perimeter constraint. Let us assume the example building with three storeys. The equalities of Equation 2 have to be fulfilled. The space areas are nearly balanced across storeys, but the floor plans are different. For each storey the topology has been transformed into a geometry with the same dimension y. This condition fulfills two of the equalities. But the x dimensions differ by a small percentage. If the difference is larger, the space program partitioning should be modified.

Since we assume that all dimensions are minimal, we can not shrink the largest x. We can only extend the x of the smaller storeys. We do this relative to the area of each space and also controlled by the penalty factor p.

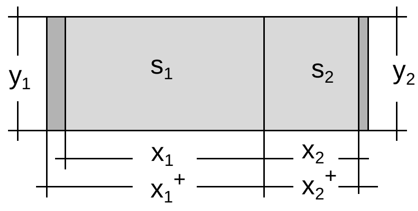

Figure 4. Area extension example

Let us assume two adjacent spaces that are separated by an x-cut-sheet, as shown in Figure 4. If a^+ denotes the extended area, then we define the extension d_i by

$$a_i^+ = a_i \cdot (1 + d_i) \tag{4}$$

If y_i is fixed, the following also holds (for x respectively):

$$x_i^+ = x_i \cdot (1 + d_i) \qquad \text{for } y_i = const \tag{5}$$

Let s_0 be the parent s_1 of and s_2 with

$$a_0 = a_1 + a_2 \tag{6}$$

then for the extended areas we postulate

$$\frac{d_1}{d_2} = \frac{p_2}{p_1} \quad \text{and} \tag{7}$$

$$a_0^+ = a_1^+ + a_2^+ \tag{8}$$

If d_0 is known, we can calculate d_1

$$d_1 = d_0 \cdot \frac{a_1 + a_2}{a_1 + a_2 \cdot p_1 / p_2} \tag{9}$$

and d_2 from Equation 7. This allows us to calculate all extensions starting from the top or from any internal node of the dissection tree down to the leaves of the corresponding subtree. We can imagine this process as a penalty controlled propagation of parent extensions to child extensions. Both children together provide the necessary extension for the parent.

In order to simplify the calculation of the d_i during resolving constraints, the fraction in Equation 9 is calculated and stored in the geometry translation process for all nodes. Since *an* and *at* can be shape dependant, we use *ap* for these calculations. In the case of spaces with x or y dimension constraints, these are used instead of *ap*.

Since we have defined the penalties for the leaves only, we have to give a rule for calculating internal node penalties. We do this from the leaves up, using the same indices as above

$$p_0 = \frac{p_1 + p_2}{2} \tag{10}$$

This basic approach works fine as long as only one constraint is defined in a subtree. If more than one is present, we have to refine our idea. Here we will just outline the idea and explain details in Chapter 5. We start with fulfilling the highest constraint in the subtree. Then we propagate the extensions d with Equation 9 and Equation 7 breadth first down the tree until we meet the next node with a constraint. This may require a larger d than was assigned to the node. In this case we propagate this new d up until it can be met. To get more slack, the extensions are no longer distributed according to the above proportions, but one-sided to the node that requires the larger d. In the extreme case this can mean to even extend the largest storey in the example. Once a node is found with sufficient slack, the top-down propagation continues until the next constraint is found. We assume that during early design phases the number of vertical constraints is small and this “Jo-Jo” process terminates quickly.

4.2 *Shape function based floor plan generation*

Before we can explain the algorithm for vertical constraint resolving, shape functions and their role in floor planning have to be explained. Shape functions have been introduced by Otten (1983) to define the relation between shape and layout area in VLSI designs. Zimmermann (1988) used this definition to estimate the effects of routing areas on layout area for chip planning purposes. This technique is closely related to considering wall and other areas in floor planning and led to the current research.

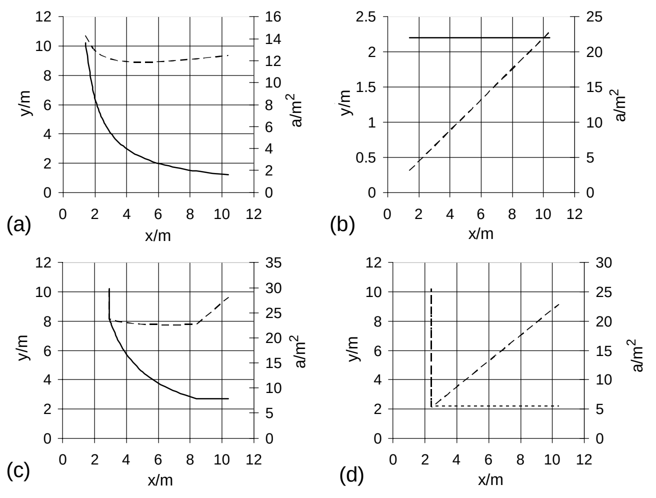

Figure 5. Different shape functions (dashed line = *a*): (a) constant *an*, (b) constant yn, (c) constant an in a limited range, minimum dimensions outside the range, (d) fixed *xn* and *yn*, the dotted line defines empty space

A *shape function* is defined as the relation of *y* to *x* for all possible aspect ratios *y*/*x*. Even if *an* is constant (horizontal constraint), *a* normally varies with different aspect ratios, because the volume of the walls changes with length. Figure 5(a) shows an example of this type.

Shape functions can also express other horizontal constraints. Hallways have typically one dimension fixed and the other variable. Figure 5(b) shows the shape function of a hallway that extends in the x-direction. Other constraints define minimum dimensions of rooms. For example, no office dimension should be smaller than a certain value. Figure 5(c) shows the corresponding shape function. In this case, *an* exceeds *ap* when one of the limits is reached. Instead of hard limits, an can also be continuously increased outside of an ideal aspect ratio. Finally, a space with fixed dimensions is defined by the shape function in Figure 5(d). An example is an elevator shaft. In order to fit such a space to a more flexible space, L-shaped spaces would be created. Here we will introduce empty space by extending the fixed space in one dimension as shown by the dotted lines.

All shape functions have a limited definition range. This is an additional constraint on possible aspect ratios. Also, discrete rather than continuous dimension increments can be expressed. During the topology to geometry transformation, aspect ratio constraints can lead to unsolvable problems. In such a case, the topology, the space program, or the overall aspect ratio have to be changed.

As explained in Zimmermann & Suter (2004), for the purpose of generating floor plans, shape functions have to be defined for all leaf nodes of the dissection tree. This is done automatically by using the space program data and the wall type information. Wall types are either assigned as defaults during the interactive dissection process or can be modified by the designer at any time. External walls are assigned fully to the adjacent space. Internal walls are normally split into two layers and these are assigned to the two adjacent spaces. For this purpose all walls are automatically partitioned into sections with exactly two adjacent spaces. For external walls the surrounding space is the second space.

Shape functions of internal nodes are calculated by adding the shape functions of the siblings. This is a bottom-up process that continues in all branches of the tree until a horizontal cut-sheet is reached. This limit is necessary because the height od spaces (*z*) can not be traded with *x* or *y*. Thus shape functions are always 2-dimensional.

The subtrees that are reached at this point are the major sections that are denoted by the first index (see Chapter 3.1). For each section one total dimension (*xt* or *yt*) for the top node has to be chosen. In the case of several vertically stacked storeys, the same dimension and value will be selected to align one direction of outer walls. With this selection all other dimensions in the subtree are derived from the shape functions and a floor plan geometry without holes or overlaps results. This is a top-down process. The algorithms have been described in Zimmermann & Suter (2004).

The problem remains that the section may not fit together in the free dimension and that other vertical constraints are violated. The basic idea for a solution has been explained in Chapter 4.1 and will be extended in Chapter 5.

5 ALGORITHM

After the introduction of the basic idea and the formulas in Chapter 4.1, the alignment process is straight-forward. We will describe the individual steps of the algorithm in textual form, because a formal language would hide more than it would explain:

Step 1: Plan major sections. Partition space program with the goal to evenly distribute net areas to stacked sections. Sketch floor plan for each section.

Step 2: Mark the cut-sheets in the sketches, compute sums of space areas, calculate total volume and net dimensions of the building. Dissect the building space, major sections first and then each section. This results in a geometric representation of the topology as for example in Figure 2 and the dissection tree as in Figure 3.

Step 3: Identify nodes that result from z-cuts, but do not contain z-cut-sheets in the associated subtree. For each subtree perform the shape-function based geometry transformation. Try to reduce the number of vertical constraints by using identical dimensions where possible. Assign penalty factors to leaf nodes.

Step 4: Identify vertical constraints. Enter relations between affected spaces and add constraint equations to the relations. Avoid redundant constraints (if *a*=*b* and *b*=*c*, *a*=*c* is redundant).

Step 5: Traverse the tree breadth-first until the first constraint is found. Find the second node of the constraint relation. Select the node with possible slack. Calculate the extension *d* for this node as defined in Equation 4 and apply *d* to all nodes of this subtree. If *d* extends x, Equation 9 and Equation 7 are used for nodes that result from x-cuts only. For nodes resulting from y-cuts, *d* is copied. For extensions in y, the above is applied respectively. Execute the geometry transformation to check if with the new dimensions a valid solution can be found. If not, which can happen if the valid range of a shape function is exceeded, other modifications as proposed in Chapter 4.2 have to be made.

Step 6: Repeat Step 5 until all vertical constraints at the highest level in the tree have been resolved. This can require repeating the process for nodes that have already been visited, because of the order of the visits. Because we do not reduce the minimal dimensions of the space program and if more than two nodes are directly related, the node with no slack may be visited after the first resolution step.

Step 7: Additional constraints are detected when the subtrees below a resolved constraint are also traversed breadth first. As in Step 5, the second node of the constraint relation has to be selected. Since all subtrees have been processed in Step 5, the extensions *d* of both nodes are known. They can be manipulated by modifying the distribution of the parent extension to the children. In Figure 4, for example, the cut-sheet position can be moved left or right within limits. If both participating nodes have extensions, both will contribute relative to their *d* value and penalty, using Equation 9 accordingly, replacing *a* by *d*. Otherwise, only one *d* is manipulated. In both cases, the *d* of the other child has to be adjusted accordingly.

If the range for modifying the *d*'s is large enough, the process stops here and all involved subtree are recalculated. Otherwise, if the required *d* is larger than what both parent nodes together can provide, the *d* is also distributed to both participating nodes as above, but the parents propagate this extension up the tree until it can be met by a node.

If during this up-propagation other constraint relations are met, these have to be reevaluated in the same way as before.

Step 8: Repeat Step 7 until all constraints have been found and resolved or no valid geometry can be generated.

6 EXAMPLE

In order to be able to show enough details, we had to make the example extremely simple. Let us take a building with 2 storeys, 8 rooms, and a load bearing internal wall in both storeys that have to be aligned. This creates a perimeter dimension and a nested wall alignment constraint.

Step 1: The space program consists of 3 rooms with 20, 30, and 50 m^2 in the first floor and rooms with 25, 30, and 3 rooms with 15 m^2 on the second floor. This sums up to 100 m^2 in each floor. Outer walls are 40 cm, inner walls 26 cm thick. Floor plan sketches are not shown here.

Step 2: The topology result of floor planning is shown in Figure 6 seperately for each storey.

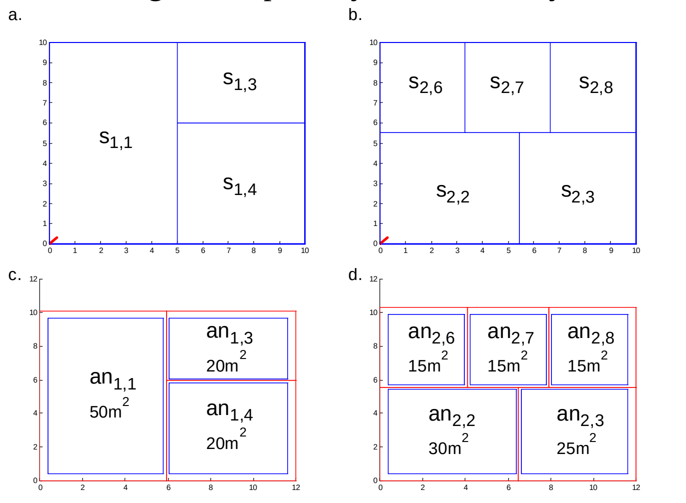

Figure 6. Test building before constraint resolution. a. First floor (schematic). b. Second floor (schematic). c. First floor (detailed), d. Second floor (detailed).

Figure 7 shows the resulting tree structure.

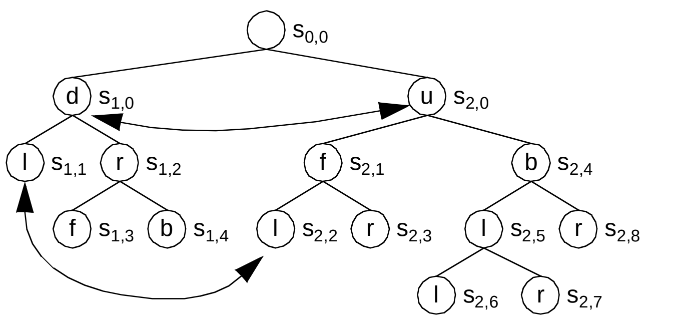

Figure 7. Dissection tree of test building. Arcs with arrow heads denote vertical constraint relations.

Step 3: For the two storeys s1,0 and s2,0 in the detailed model, the same *x*=12m is chosen and the geometry constructed with the results as shown in Figure 6. The transformation results in different y-dimensions of the storeys: $y_{1,0} = 10.076\text{m}$ and $y_{2,0} = 10.304\text{m}$. We can also extract the position and dimension of the spaces to the left of the load-bearing walls in the first storey between $s_{1,1}$ and $s_{1,2}$ as $x_{1,1} = 5.921\text{m}$ and in the second storey between $s_{2,2}$ and $s_{2,3}$ as $x_{2,2} = 6.497\text{m}$, both measured from the left side of the building. In this step we also assign penalty values for the rooms (leaf spaces): $p_{1,1} = 1$; $(p_{1,3}, p_{1,4}, p_{2,2}) = 2$; $p_{2,3} = 3$; $(p_{2,6}, p_{2,7}, p_{2,8}) = 4$. This means that necessary area increases will mostly apply to the larger rooms.

Step 4: We define two vertical constraints: The perimeter dimension constraint $y_{1,0} = y_{2,0}$ and the wall alignment constraint $x_{1,1} = x_{2,2}$. In Step 3 we have shown that both are violated in the geometry in Figure 6. The constraints are shown in Figure 7 as arcs.

Step 5: Traversing the tree in Figure 7 top down, we first find a constraint at node $s_{1,0}$ and the related node $s_{2,0}$. Since $y_{2,0} > y_{1,0}$, $s_{1,0}$ is the node with slack. With Equation 5 we get $d_{1,0} = 0.0226$. By applying Equation 9 and Equation 7 to nodes that result from y-cuts below node $s_{1,0}$, $s_{1,3}$ and $s_{1,4}$ in this case, we finally get new dimensions for these nodes and can apply the geometry transformation again. The result is $y_{1,0}$=10.304m and thus the first constraint is met.

Step 6: No more constraints at the top level are found.

Step 7: Traversing further down from $s_{1,0}$, we find a constraint at node $s_{1,1}$. This is still violated because the x-dimensions have not changed. Since node $s_{1,0}$ has been extended, there is slack in this storey and we try first to use this slack to extend $x_{1,1}$. By minimizing the net area of node $s_{1,2}$ to $ap_{1,2}$, we get $x_{1,1}$=6.061m. To get this result we have to use the shape function of $s_{1,2}$. Figure 8 gives an idea of this process. Still, the constraint is not met. Therefore, we extend $y_{1,0}$ and $y_{2,0}$ in parallel not to violate the first constraint. Again, we have to use the shape functions of the appropriate nodes as above. As a result we get

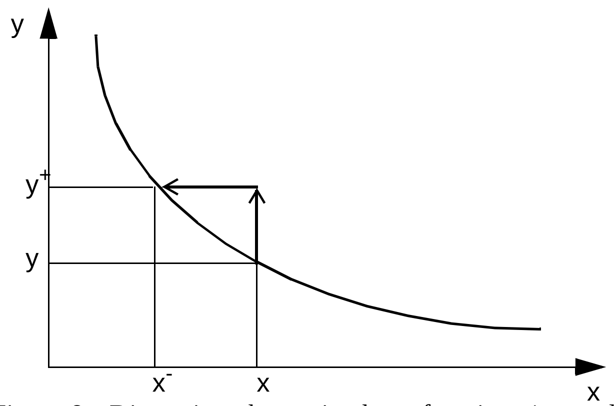

Figure 8. Dimension change in shape function: 1st modification y->y^+; 2nd modification x->x^-

the new dimensions:

$y_{1,0}$=$y_{2,0}$=10.555m

$x_{1,1}$=$x_{2,2}$=6.204m

This result shows that both constraints are met.

Step 8: Traversing further down the tree, no more constraints are found, as expected. Since the generated geometry is valid, the algorithm terminates.

In order not to confuse the reader with too many details we have not shown all intermediate calculations. In practise these are executed by tools, here we used a spreadsheet to do the calculations. The designer only needs a basic understanding of the process to interpret the results and propose modifications.

Matching of the two constraints requires an increase of total building floor area of originally 200m^2 by 8.75m^2. The designer can now decide, if he finds this result satisfying or he can either change the space program, the topology, or the penalty factors to improve the result. With tool support these operations can be performed rapidly and allow the designer to experiment in a large design space.

7 CONCLUSION

Mapping procedures such as the topology to geometry transformation algorithm introduced and illustrated above are aimed at improving the information flow in the building design. Although our algorithm is based on several restrictions and assumptions, it is already fairly complex. To further improve its generality, we anticipate work in three areas.

So far, only the 3-dimensional topology planning and the 2-dimensional topology-to-geometry transformations are supported by a prototype tool based on Matlab. Since all extensions are based either on analytical functions or on shape functions that are generated with the existing tool, the extension to full tool support is only a matter of programming effort. No basic problems have to be solved.

We also plan to experiment further with shaft constraints. The idea of the outlined algorithm seems to solve this problem, but we are not certain yet that it works under all conditions. For this purpose we first have to introduce empty spaces into the topology-to-geometry transformation.

Restricting the domain to orthogonal geometries is a significant limitation of the proposed algorithm. This is important particularly in the context of detailed design. We thus intend to incorporate geometry refinement operations, perhaps similar to those described in Suter and Mahdavi (2004). These operations extend the domain to certain classes of non-orthogonal geometries common in building design. Issues related to extending the domain, such as preserving constraints, or adapting shape functions, need to be addressed as well.

REFERENCES

Flemming, U. 1989. More on the representation and generation of loosely packed arrangements of rectangles. Environment and Planning B (16): 327 - 359.

Harada, M., Witkin, A., and Baraff, D. 1995. Interactive physically-based manipulation of discrete/continuous models. In Computer Graphics Proceedings, Annual Conference Series 199-208.

Kundu, S. 1988. The equivalence of the subregion representation and the wall representation for a certain class of rectangular dissections, *Communications of the ACM 31* (6): 752 - 763.

Liggett, R.S. 2000. Automated facilities layout: past, present and future. *Automation in Construction 9*: 197-215.

Bozer, Y., Meller, R. Erlebacher, S.1994. An improvement-type layout algorithm for single and multiple-floor facilities, *Management Science 40* (7): 918–932.

Otten, R. 1983. Efficient floorplan optimization. In Proc. Int. Conference on Computer Aided Design (ICCAD): 499 - 502.

Suter, G., Mahdavi A. 2004. Elements of a representation framework for performance-based design. *Building and Environment 39* (8): 969-988.

Zimmermann, G. 1988. A new area shape function estimation technique for VLSI layouts. *Proc. 25th Design Automation Conference*, Anaheim, USA: 60-65.

Zimmermann, G. & Suter, G. 2004. A model for hierarchical floor plan generation based on shape functions. Proc. ECPPM 2004, Istanbul, Turkey.