Identifying technical obstacles for a constraint-based mass customization system

R.A. NIEMEIJER, B. DE VRIES, J. BEETZ
Eindhoven University of Technology
Den Dolech 2
5612 AZ Eindhoven
The Netherlands
R.A.Niemeijer@tue.nl

Key words: customization, user oriented, BIM, constraints, CAD, prototype

Abstract: Flexible mass customization of buildings is still in its infancy. Current systems for the automated support of owner-driven configuration management are limited with regard to the degree of freedom they offer to end-users. In this paper we present and discuss a constraint definition and verification system that allows the architect to specify boundary conditions within which the client can vary the design. The required constraint grammar is presented and applied to a real-world building code. A 2D floor plan prototype is tested to reveal the methodological and technical issues to be solved to make a step forward.

1. INTRODUCTION

As Computer-Aided Manufacturing (CAM) methods mature and the economy of consumer products becomes increasingly demand-based, consumers are starting to expect that products will meet their needs. Many industries now tailor their products to the consumer's wishes, either by offering several different versions (e.g. Apple's iPod line) or by making one product that can be adapted (e.g. cars). This approach is called mass customization: utilizing the economic benefits of mass productions without resulting in the uniformity that is typically associated with it. One notable exception to this trend so far has been the building industry, where all too often the client still has to simply accept the architect's design. One of the reasons for this is that mass production, let alone CAM, is still a rare occurrence in the building industry; some building elements are mass-produced, but the rest, as well as the dwelling itself, still require manual labour. The lack of a computerized manufacturing process means that changes in the design are costly and error-prone. Consequently, customers are usually offered little, if any, choice.

Although the need for more consumer input into the design has been established (Huang and Krawczyk 2007, van den Thillart 2004), most initiatives for providing this have been fairly limited. Usually the architect designs several alternatives, from which clients can choose. These alternatives, however, are generally not supported by market research, but rather by the intuition of the architect. As a result, the design will still fail to conform to the client's wishes if the architect failed to take those wishes into account. For example, a house with a choice of one or two bedrooms will be unsuitable for a family with four children. One possible solution for this problem is to have the architect design the house along with a set of constraints, within which consumers are granted the freedom to modify the design according to their wishes. The selection of constraints in this approach is a critical factor for the success of such an approach. Both too few and too many constraints result in an unworkable situation: designs would be impossible to build or so little choice would be offered that this approach would offer no advantage over the current situation.

The research that has been done so far into constraint-based CAD (Computer Aided Design) has mainly focused on providing a system for the architect (Gross 1996, Kelleners 1999, Eggink et al. 2001, Donath and Böhme 2007) and the technical aspects of using constraints in CAD (Martini 1995, Anderl, and Mendgen 1996, Hoffmann and Kim 2001, Bettig and Shah 2003). In this study however, we examine a constraint-based CAD system where the architect imposes the constraints on the design, after which

the user can modify the design within those constraints. Specifically, this paper focuses on what constraints are required, how they are imposed on the design and how they can be solved.

2. METHODOLOGY

First, an analysis is conducted to determine which constraints would be necessary to allow the architect to express his design wishes. The next step is to determine how these constraints should be handled. Finally, a prototype is developed to determine whether the chosen approach is feasible. The prototype discussed in this paper is limited to a 2D floor plan consisting of only walls, windows and doors. These can be moved and rotated and the architect can specify constraints, which are then automatically checked by the program. The choice for a 2D scene instead of a 3D scene means that the essential properties of the system can be tested without having to worry about problems that are not very important at this stage, such as 3D object intersection testing. If the prototype shows that it is possible to handle the constraints in real-time on a scene of decent complexity, it will be used as a basis for future prototypes, with additions such as 3D scenes and a more user-friendly way of entering constraints.

3. CONSTRAINT ANALYSIS

Initially, constraints seem to be fairly easy to represent, as at first only simple constraints come to mind, such as “The wall can be brick or concrete” and “The wall must be at least 3m long”. However, studying real-world examples, e.g. the building code for dormers in the Netherlands, reveals that constraints can become fairly complex. For instance, constraints such as “the width of the dormer must be no more than 1/3 the width of the building” and “gable roofs with an angle less than 30 degrees cannot contain dormers” reveal that constraints must be able to handle concepts such as arithmetic, conditionals and Boolean logic. In fact, the required flexibility of the constraints approaches that of a simple programming language.

We therefore decided to structure the constraints in the same way that many programming languages are structured: as syntax trees (Louden 1997). Just as English sentences can (according to some classifications) be divided into subjects and objects which can be further divided into nouns, adjectives, etc., so can computer programs be decomposed into smaller elements. This tree structure can sometimes be seen directly in the language’s syntax (e.g.

Lisp's parentheses, Python's indentation and C's curly braces), but this is not a requirement. For example,

```
Window01.Width > 500
```

might be converted to the following syntax tree:

```
MoreThan
+- Left
|  +-- Property
|      +- Name
|      |  +- Width
|      +- Object
|         +- Window01
+- Right
   +- Constant
      +- 500
```

Figure 1 shows our idea for the structure of a *constraint*. A *constraint* is something that, when evaluated, returns a Boolean. True if the constraint is satisfied, false if it is not. A constraint consists of a Boolean condition and two constraints, with the condition determining which one is evaluated. The condition and the constraints are all *predicates*. A *predicate* is a function such as == (equals), > (more than) or <= (less than or equal to); it takes two values and returns a Boolean. These values can be *constants*, *expressions*, *properties* or *relations*. A *constant* is a simple value, e.g. 3, "Hank & Co." or 150 cm. A *property* holds a property of a part of the building, such as width, material or manufacturer. A *relation* describes a relation between two parts of the building, such as intersection or distance. Finally, an *expression* applies a mathematical operation to a value. The bottom part of the diagram explains what we mean by "part of the building": an *ElementSelector* selects one or more elements of the building. It can either be a single element (*ElementName*) or a class of elements, e.g. windows or walls (*ElementClass*). An element class selection can be qualified by limiting the selection to a parent, e.g. "walls on the second floor" or "windows in the kitchen".

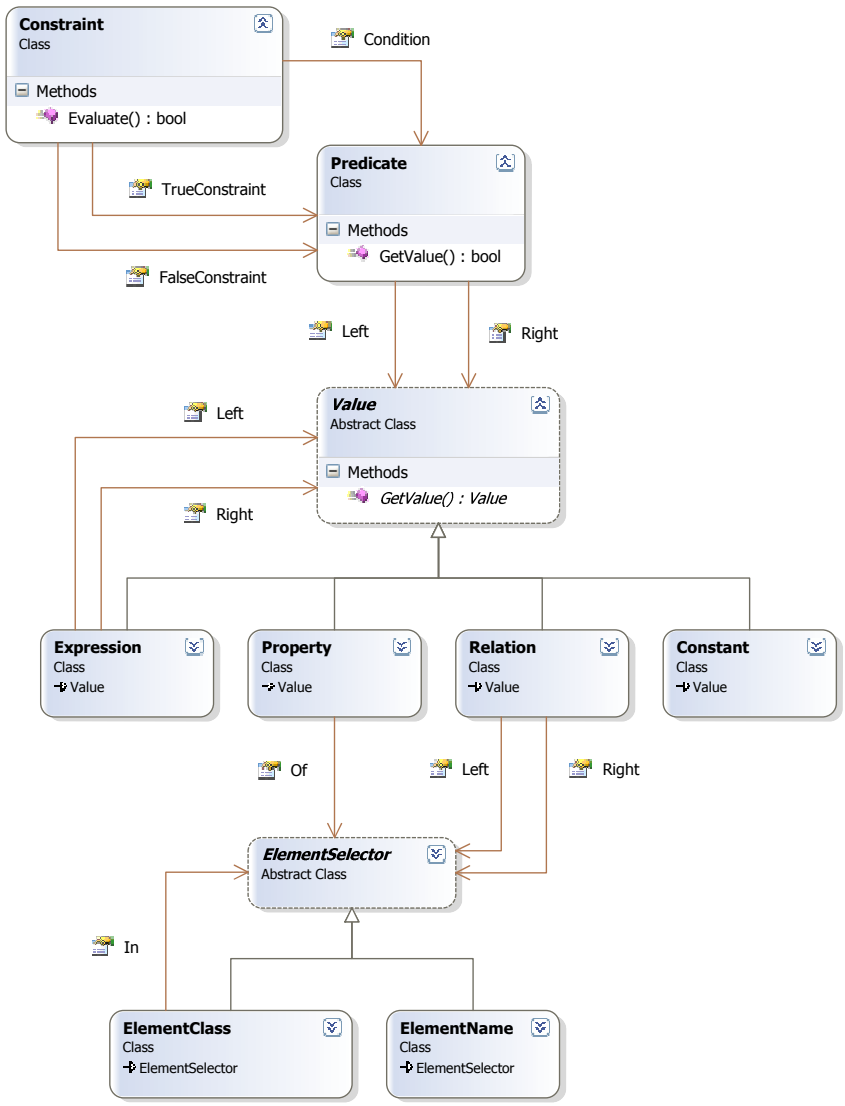

Figure 1: Constraint structure

As a test case, this constraint grammar was used to translate the building code for dormers in the Netherlands to constraints. Table 1 shows two successful examples and three types of clauses that cannot be expressed by the proposed system.

Building code text	**Resulting constraint**	**Problem explanation**
The distance between the dormer and the eaves of the roof must be between 0.5 and 1 m	The distance between dormers and eaves of roofs must be more than 0.5 m and the distance between dormers and eaves of roofs must be less than 1 m	None
The height of the dormer must be less than 1.5 m (including roof edge)	The height of dormers plus the height of roofs in dormers must be more than 1.5 m	None
There must be no visible connections or seams between the walls and roofs of the dormer (in the case of sheet material)	Inexpressible	Apart from the fact that the absence of seams is difficult to prove, expressing the constraint is difficult
The quality of the building style must be similar to that of the rest of the neighbourhood	Inexpressible	This formulation is too vague to be checkable by a computer.
Dormers on pitched roofs of tenement buildings must always be submitted to the committee	Inexpressible	This constraint defeats the purpose of mass customization

Table 1: Translating building codes into constraints

We discovered no grammar-related problems. The main reason behind the inexpressibility of some constraints is that the building code is designed for manual checking procedures: some constraints are so vague that they cannot be formalized and thus require explicit human intervention. This could be remedied by formulating clear, objective rules that can be checked by computers as well as humans. Even in the case that computer verification proves insufficient this will reduce costs, as architects can judge the design before submitting it to the committee, thus eliminating change-reject cycles.

To illustrate tree structures resulting from our translation, Figure 2 shows the constraint tree for the constraint “The distance between the dormer and the eaves of the roof must be between 0.5 and 1 m”.

```
Constraint
+- Condition = TruePredicate
+- TrueConstraint
|  +- Predicate:And
|     +- Left
|     |  +- Expression: MoreThan
|     |     +- Left
|     |     |  +- Relation:DistanceBetween
|     |     |     +- Left = ElementClass:Dormer
|     |     |     +- Right
|     |     |        +-- Property:Eaves
|     |     |            +-- Of = ElementClass:Roof
|     |     +- Right
|     |        +- Constant: MeterConstant
|     |           +- Value = 0.5
|     +-- Right
|        +- Expression: LessThan
|           +- Left
|           |  +- Relation:DistanceBetween
|           |     +- Left = ElementClass:Dormer
|           |     +- Right
|           |        +-- Property:Eaves
|           |            +-- Of = ElementClass:Roof
|           +- Right
|              +- Constant: MeterConstant
|                 +- Value = 1
+- FalseConstraint = TruePredicate
```

Figure 2: Example constraint tree

Since no grammar problems were found, we are confident that the structure presented in Figure 1 will be usable with little or no modification.

4. CONSTRAINT HANDLING

After a constraint is specified, there are two approaches for handling it: constraint solving and constraint checking. A constraint solver takes a list of constraints and returns a list of possible solutions.

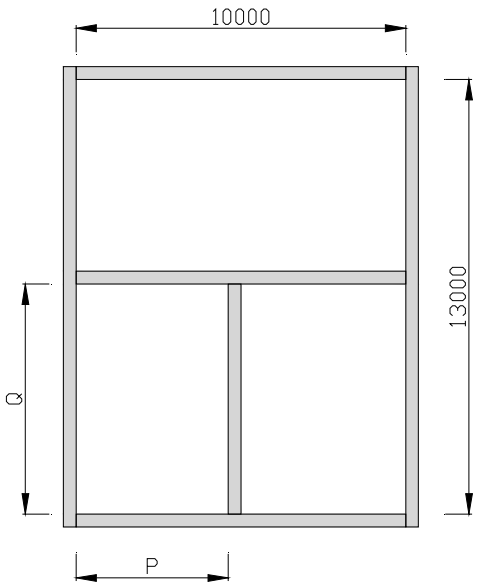

Figure 3: A basic floor plan

For example, the situation presented in Figure 3 might lead to the following constraints (assuming the external walls are fixed and the interior walls can be moved freely):

$$P \in \mathbb{N} \wedge Q \in \mathbb{N}$$
$$2500 \leq P \leq 7400$$
$$2500 \leq Q \leq 11400$$
$$P \times Q \geq 8 \times 10^6$$

I.e. the distances P and Q are whole numbers (measured in millimetres), P and Q are at least 2.5 meters and at most 7.4 or 11.4 meters and the area of the lower left room is at least 8 m^2. The constraint solver will then produce the list of solutions (P = 2500 and Q = 3200, P = 2500 and Q = 3201, etc.). This example shows a practical problem of this approach when choosing a fine granularity during the discretization of the problem: the resulting solution space will be very large. This simple example has a solution size of several million items. Adding just two more independent variables that can both vary by a few meters will result in trillions of solutions, which can no longer be contained in a contemporary computer's RAM (2 GB of memory

will hold around 500 million 32-bit integers). In order to use the solving method a lower resolution will have to be accepted, for instance by putting the elements on a 10 cm grid, which reduces the amount of solutions to several thousand.

By contrast, in constraint checking the "solving" of the constraints is done by the user. The system merely checks whether the given solutions conform to the constraints. Using the same three-room example, submitting a design where P = 2600 and Q = 3000 will not be accepted since the last equation does not hold ($P \times Q = 7.8 \times 10^6$). The user then modifies the design until it satisfies all the constraints. In contrast with constraint solving this method is very efficient, since all that is required is to check a series of Boolean predicates. Although reducing the resolution of the design is not an insurmountable problem, we have chosen to use constraint checking because it is simpler and better matches our goal of checking whether the customer's design is valid.

5. PROTOTYPE: CONSTRAINT CHECKING

In order to test the principle of constraint checking, a simplified prototype was created. As illustrated in Figure 4, the program consists of a standard viewport on the left hand side, a list of constraints in the top right corner and a list of violated constraints in the bottom left corner. The prototype was written in C# 3.0.

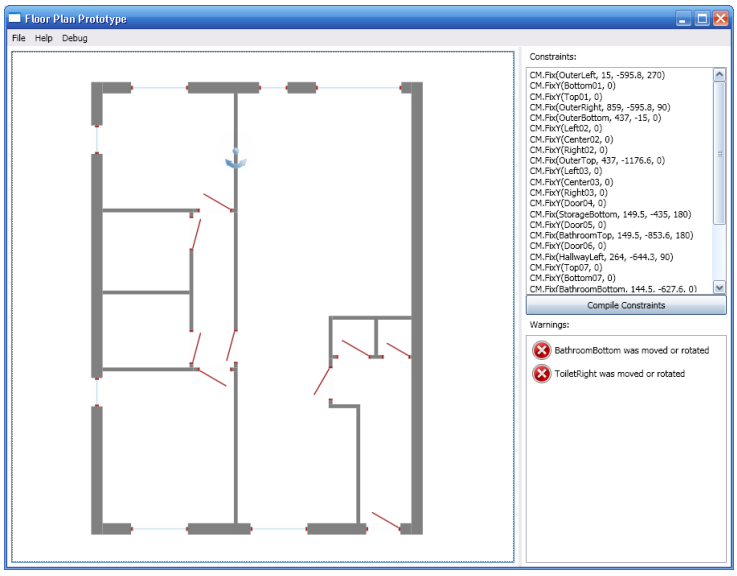

Figure 4: Prototype screenshot

Parse tree structures (Louden 1997) were used to formulate and capture the constraints in our example. The particular implementation of such parse tree structures (the Expression class in MS C# 3.0) we used in the prototype allows the runtime compilation of parse trees from C# code. We formulated the constraints illustrated in Figure 1 using C#, after which they are dynamically compiled into expression trees. Just as in the proposed constraint system these expression trees can be evaluated. If a constraint is not satisfied, the expression tree is used to create the error message that is presented to the user. For example, for the constraint expression

```
Wall01.Height > 2600
```

the underlying engine throws the error message “The height of Wall01 must be greater than 2600” when it is violated. The next section discusses the prototype in more detail.

6. RESULTS

To test the prototype, we used a floor plan from a commercial housing project (www.lupine.comwonen.nl). As boundary conditions, we fixated the exterior walls and the sanitary cores (see Figure 5). The other walls could be freely moved and rotated. Table 2 shows the constraints that were implemented for this prototype.

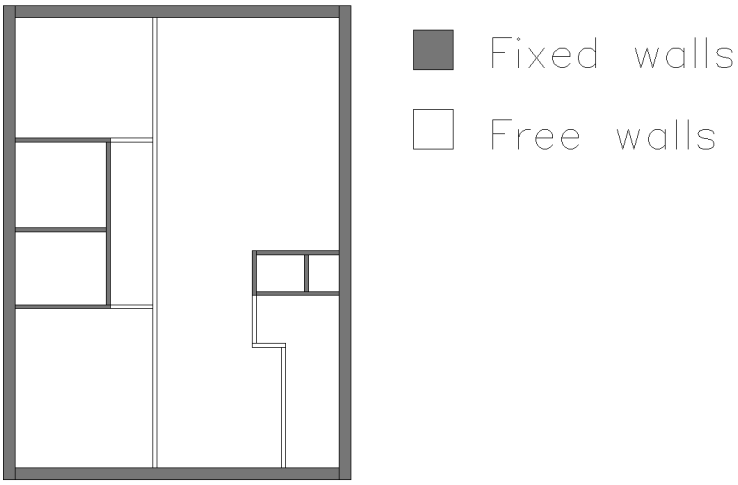

Figure 5: Prototype floor plan

Used	
Fix	Completely fix an object
FixY	Fix the Y position of an object
Implemented but not used yet	
FixX	Fix the X position of an object
FixRot	Fix the rotation of an object
Intersects	Specifies that two objects must intersect
DistanceTo	A relation between two objects. Can be compared using standard equals, more than, etc. predicates
Contains	Specifies that one object must be inside another object (e.g. that a window must be completely inside a wall)

Table 2: Concrete examples of constraints

Since the prototype did not support the deletion of objects, moving an object outside the floor plan was treated as deleting the item. The prototype was tested by several co-workers of the DDSS research group. Figures 6 through 9 show the designs they made (The reason that not all connections line up perfectly is that only grid-based snapping was implemented and changing the size of objects was not possible. Examples of this can be seen in the lower part of Figure 6 and the top right of Figure 9).

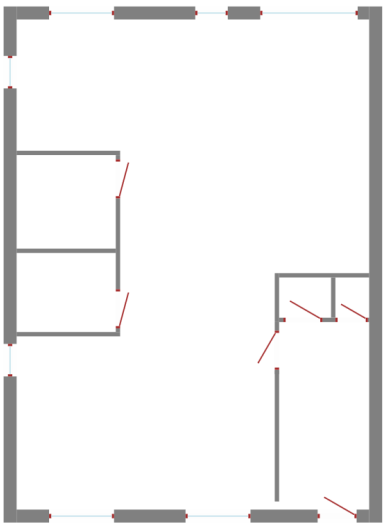

Figure 6: Design #1

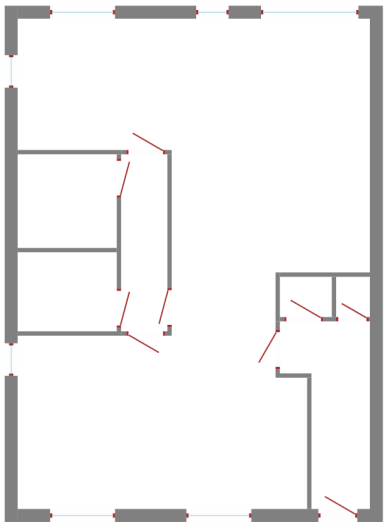

Figure 7: Design #2

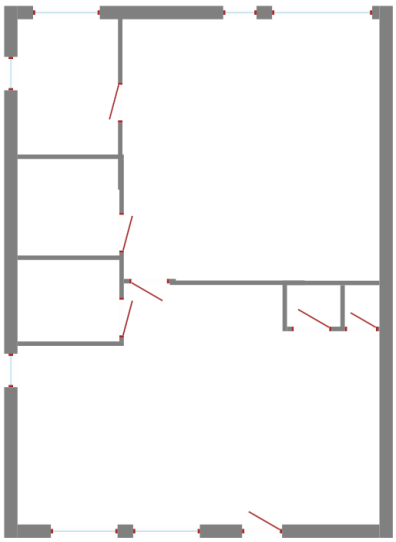

Figure 8: Design #3

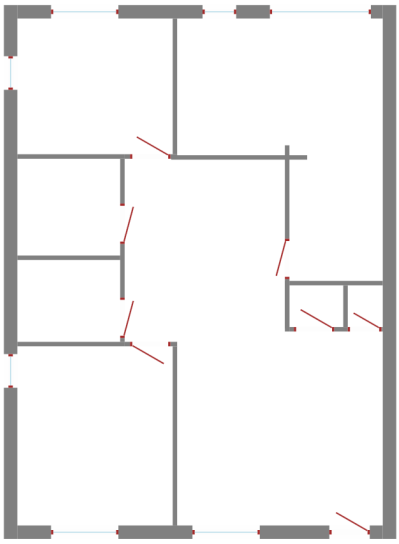

Figure 9: Design #4

Apart from features that were not yet implemented or not functioning correctly, the test subjects made the following observations:

- In the current version one has to pay close attention to see if changing an object is legal or not. A warning in the viewport would be appreciated. This supports the findings from an previous round of prototype testing focused on user-interface aspects where having both a list of errors and an icon over the object was the preferred method of error notification.
- It was unclear whether or not objects could be adjusted (there were two groups of sanitary cores with fixed walls). A possible solution for this is presented in the discussion.
- In the current version all constraint violations are treated as errors. One of the participants suggested also implement warnings, to signal designs that are not necessarily wrong, but that might possibly cause a problem or that are generally considered to be suboptimal. This comment was partially caused by the fact that it was possible to move fixed objects. The issue of allowing this is discussed in the discussion.
- Moving a completely fixed object currently produces the error message that the object was “moved or rotated”. This message was judged as being both ambiguous (as it does not discriminate between the two) and vague (the message only indicates that either event happened, not that it is a problem).
- Finally, one participant saw little use in being able to rotate objects. This might be caused by the fact that the outer walls form a rectangular shape (see Figure 5). A floor plan with non-orthogonal walls might make this feature more useful.

7. DISCUSSION

Although this prototype is a simplified situation, the chosen approach has not revealed any fundamental problems of constraint-based design. As desired, moving any of the fixed walls immediately produces an error. The previous section shows that a wide variety of user-created designs is possible, despite the inability to change the size of walls or to add or remove windows and doors. Therefore we feel confident that the chosen approach is a viable one, and will continue with the prototype development.

One consideration for future versions is how to handle objects with geometric constraints. Currently, it is possible to move or rotate an object regardless of any constraints placed upon them. Since this will immediately generate an error message, the usefulness of it is questionable. Not allowing the constraint to be violated will prevent inadvertent changes. Another consideration is the conceptual approach towards imposing constraints, for which there are two possible approaches: either starting with a completely

fixed design and allowing freedoms, or starting with a fully modifiable design and imposing restrictions. Both have their pros and cons. The advantage of starting with a fixed situation is that an oversight on the part of the architect will not allow invalid designs. The downside is that every additional degree of freedom will have to be carefully considered and designed by the architect, offering little benefit over the present day situation. Starting with no restrictions better embodies the spirit of this project, but the architect runs the risk of allowing unintended freedoms. We are looking into possible hybrid solutions which would offer the best of both worlds. For instance, making this a per-object choice means that important objects, such as the outer walls, can start fixed while less important objects (inner walls, windows, etc.) start out unrestricted. One possible user interface consequence of this hybrid approach might be to display a lock icon over highlighted objects, similar to the way Autodesk's Revit program displays constraints (Strömberg 2006). A closed lock would indicate a completely fixed object, a closed lock with a question mark means a fix object with some freedoms, an open lock means a completely free object and finally an open lock with a question mark indicates a free object with restrictions. Clicking on the lock icon would show the list of constraints that affect the object. This would both help the architect get an overview of which objects he has not handled yet and show the clients in what ways they can or cannot change the object.

A third consideration is the way the constraints are entered. So far we have identified three possible solutions: using a programming language (e.g. Python), using natural language (e.g. English), or using a Domain-Specific Language. The programming language is the simplest of the three (hence it is the system used in the current prototype), but we anticipate that architects, who are usually not familiar with programming, will not find this method user-friendly enough. Allowing constraints to be entered in a natural language most closely resembles their current methodology, but is also the most technically difficult of the three to implement. The use of Domain-Specific Languages in CAD, which has been explored before (Spinellis 1999, Gross 2001), lies between these two alternatives, in terms of both difficulty and expected familiarity, depending on the way it is implemented, which can range from a traditional text-based interface to a more graphical one. In order to choose between these options we intend to have all three options tested by architects or architecture students to see which one works better in practice.

8. BIBLIOGRAPHY

Anderl, R. and R. Mendgen, 1996, "Modelling with constraints: theoretical foundation and application", *Computer-Aided Design*, 28(3), p. 155-168

BBVH, 2008, *com·wonen – lupine*, http://www.lupine.comwonen.nl/pdf/Lupine_verkoopbrochure_fase2.pdf

Bettig, B. and J. Shah, 2003, "Solution selectors: a user-oriented answer to the multiple solution problem in constraint solving.", *Journal of Mechanical Design*, 125(3), p. 443-451

Donath, D. and L.F.G. Böhme, 2007, "Constraint-Based Design in Participatory Housing Planning", in: *eCAADe 2007 Conference, Frankfurt am Main, Germany*, p. 687-694

Eggink, D., M.D. Gross and E. Do, 2001, "Smart Objects: Constraints and Behaviors in a 3D Design Environment", in: *eCAADe 2001 Conference, Helsinki, Finland*, p. 460-465

Gross, M.D., 1996, "Elements That Follow Your Rules: Constraint Based CAD Layout", in: *ACADIA 1996 Conference, Tuscon, USA*, p. 115-122

Gross, M.D., 2001, "FormWriter: A Little Programming Language for Generating Three-Dimensional Form Algorithmically", in: *CAAD Futures 2001*, p. 577–588

Hoffmann, C.M. and K.J. Kim, 2001, "Towards valid parametric CAD models", *Computer-Aided Design*, 33, p. 81-90

Huang, C. and R. Krawczyk, 2007, "A Choice Model of Consumer Participatory Design for Modular Houses", in: *eCAADe 2007 Conference, Frankfurt am Main, Germany*, p. 679-686

Kelleners, R.H.M.C., 1999, "Constraints in object-oriented graphics", *PhD Thesis*, Eindhoven University of Technology

Louden, K.C., 1997, *Compiler construction : principles and practice*, PWS, Boston

Martini, K., 1995, "Hierarchical geometric constraints for building design", *Computer-Aided Design*, 27(3), p. 181-191

Spinellis, D., 1999, "Reliable software implementation using domain-specific languages", in: *ESREL, 10th european software conference on safety and reliability*

Strömberg, J., 2006, "Integrating Constraints with a Drawing CAD Application", *Masters Thesis*, Stockholm University

van den Thillart, C.C.A.M., 2004, *Customised Industrialisation in the Residential Sector: Mass customisation modelling as a tool for benchmarking, variation and selection*, Sun, Amsterdam