One size fits none – A user interface for constraint-based design
Niemeijer, R.A.; de Vries, B.; Beetz, J.

Published in:
ACADIA 09: reForm() – Association for Computer Aided Design

Published: 01/01/2009

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the author’s version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 18. Dec. 2018
One size fits none – A user interface for constraint-based design

Remco Niemeijer, Eindhoven University of Technology,
Bauke de Vries, Eindhoven University of Technology,
Jakob Beetz, Eindhoven University of Technology

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, due to the lack of an easy way to specify those freedoms. In this paper we present the prototype of an interface that allows architects to define constraints to which user-customized dwellings must conform.

1 Introduction and related research
People who buy a house typically have little, if any, input into the design. Because changing the technical drawings is a laborious and error-prone process, no more than a few alternatives are typically offered. By using constraints, however, changes can be made while guaranteeing the new design is still valid. This enables the client to make changes to the design without incurring significant additional costs.

The idea of automatically checking building, while common in the mechanical engineering industry (Gross 1996; Bettig and Shah 2003), has yet to be widely adopted in the building industry. The only major commercial CAD package to feature enforceable rules so far is Revit (Strömberg 2006). Projects that use such rules in practice, such as SMARTcodes (Wix, Nisbet and Liebich 2008) exist, but are few and far between. When such rules are used, they are frequently only applied to the geometry and not to non-physical aspects such as materials, costs or thermal isolation (Eggink, Gross and Do 2001; Nassara, Belblidia and Alby 2003; Thabeth and Beliveau 2003).

2 System description
We propose a system where the buyer of a house can change the architect’s design before the house is built, to achieve a better match with their needs. The traditional way of doing this, where each client would have to separately confer with the architect face-to-face, works only for small projects and does not scale up well to the more typical large housing projects. In our approach, the architect, after creating the initial design, defines a set of constraints (Kelleners 1999; Donath and Böhme 2007). After the initial design and the rules are completed, the buyer can modify the design within the rules of the architect.
3 Constraints

We define a constraint to be an assertion about building elements. It is effectively a function that takes as its argument a list of building elements and returns a Boolean, indicating whether all the elements satisfy the constraint. Unfortunately, constraints like this only work for requirements that can be objectively judged. Rules about aesthetical quality, for instance, will still have to be checked by humans. In our system we opt for constraint checking instead of constraint solving, since the solution space is far too large to effectively solve the constraints automatically and because this process does not give the user enough input.

At some point, the constraints will have to be entered into the system. This action will most commonly be performed by architects, as most of the building codes and law constraints will only have to be entered once. The goal, therefore, is to find a method of constraint entry that is easy for architects to work with. There are several alternatives, shown in Table 1:

Table 1

<table>
<thead>
<tr>
<th>Input type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming language, e.g. Java, Python, Haskell</td>
<td>Powerful</td>
<td>Difficult syntax</td>
</tr>
<tr>
<td></td>
<td>Easy implementation</td>
<td>Requires training</td>
</tr>
<tr>
<td>Domain-Specific Language</td>
<td>Easier syntax than programming language</td>
<td>Still requires training</td>
</tr>
<tr>
<td>Natural language</td>
<td>No training required</td>
<td>Very difficult to implement</td>
</tr>
<tr>
<td></td>
<td>Very easy to use</td>
<td></td>
</tr>
<tr>
<td>Tree-based input, e.g. Yahoo Pipes</td>
<td>Graphical User Interface</td>
<td>Difficult to read</td>
</tr>
<tr>
<td>Block input, e.g. Lego Mindstorms NXT</td>
<td>Fairly easy to read</td>
<td>Not as compact to display as a language-based solution</td>
</tr>
</tbody>
</table>

We have chosen the block input, or “puzzle piece”, option since we believe it offers the most advantages and the least disadvantages. One change we make, however, is that instead of representing every constraint as a single sequence of pieces, they are split up into four different sections. This is because initial paper-only tests revealed that all but the shortest sentences would quickly become very hard to understand. We have opted to split constraints into elements, definitions, conditions and rules. The elements section defines which elements the constraint applies to. The definitions section allows creating definitions similar to those found in legal documents. While most useful if a concept is referred to more than once, they can also help to split up long sentences. Conditions are a further filtering of the elements the constraint applies to. Finally, rules determine the rules the filtered elements must obey. Table 2 shows two example translations, taken from the Dutch building codes for dormers.

One size fits none – A user interface for constraint-based design
Table 2: Example translations

<table>
<thead>
<tr>
<th>Original constraint</th>
<th>Translated constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>The material of walls of dormers must be wood sheet, wood or zinc</td>
<td>Element: Walls in Dormers</td>
</tr>
<tr>
<td></td>
<td>Definition: m is its material</td>
</tr>
<tr>
<td></td>
<td>Condition: -</td>
</tr>
<tr>
<td></td>
<td>Rules: m must be wood or m must be wood sheet or m</td>
</tr>
<tr>
<td></td>
<td>must be zinc</td>
</tr>
<tr>
<td>Half-span roofs with an angle of less than 30 degrees cannot contain dormers</td>
<td>Element: Dormers</td>
</tr>
<tr>
<td></td>
<td>Definitions: -</td>
</tr>
<tr>
<td></td>
<td>Conditions: The type of its roof must be half-</td>
</tr>
<tr>
<td></td>
<td>span roof</td>
</tr>
<tr>
<td></td>
<td>Rules: The angle of its roof must be more than</td>
</tr>
<tr>
<td></td>
<td>30 degrees</td>
</tr>
</tbody>
</table>

4 Prototype descriptions

We developed a working prototype based on the approach described above, to test whether or not it works in practice. Figure 1 shows the main screen of the prototype, where the four sections of a constraint can be edited. In these four sections, rules can be added, edited and deleted. Creating or editing a rule brings up the puzzle sequence editor, shown in Figure 2. In the puzzle sequence editor, the left side of the screen contains the “library” of available pieces. These can be dragged over to the work area on the right, where they are assembled into constraints. When a puzzle piece is placed, the input is checked by a grammar to see which pieces can legally follow it. The puzzle piece library on the left is updated to show only those pieces. As a result, the user sees only useful pieces, which reduces information overload and makes
it easy to see if a mistake has been made: if the desired piece is not listed there must be an error in the sentence. When the constraints are checked, the sequences of puzzle pieces are converted to functions that are applied to every building element of the house (since the filtering takes place in the functions) and any violations are then reported to the user.

5 Discussion
Although the chosen puzzle piece approach seems to be reasonably successful in allowing people to define constraints, it is not a very fast method. Having to drag all the pieces to the correct location is a rather laborious process. Having said that, the task load should decrease in time, because many constraints will recur in different projects. The architect can build up a library of constraints for many different building elements that can be re-used after minor adaptation in other projects. As for the system as a whole, it will most likely not replace any existing methods. It will instead become an additional option for the architect. Some architects make one design, some offer several, and in the future some architects will allow clients to make decisions of their own.

References