Constraint Soup
Interpreting Natural Language Architectural Constraints

Remco Niemeijer, Bauke de Vries, Jakob Beetz
Design and Decision Support Systems
Eindhoven University of Technology
Den Dolech 2, Eindhoven, The Netherlands

Abstract—To facilitate mass customization in the building industry, an automated method is needed to check the validity of user-created designs. This check requires that numerous complex building codes and regulations, as well as architects’ demands are formalized and captured by non-programming domain experts. This can be done via a natural language interface, as it reduces the required amount of training of users. In this paper we describe an algorithm for interpreting architectural constraints in such a system.

Computer aided architectural design; constraints; natural language processing; mass customization

I. INTRODUCTION

Currently, people buying a house in a newly built housing project are generally faced with one of two options: either only a single fixed design is available, or a few different discrete options can be chosen from a brochure. This lack of individual customization means that houses are generally not as well adjusted to the home owner’s needs as they could be. Allowing customers to modify the design to their needs would remedy this. The conventional way to do this is to have every buyer confer with the architect. However, this time-intensive approach scales poorly to larger projects. An alternative approach would be to use mass customization, i.e. to allow people to customize the design of their house by themselves without involving the architect directly. One of the biggest challenges in this approach is the question of how to support people, who have little to no architectural knowledge, in creating an acceptable building design. Naturally, it is neither desirable nor possible to exclude the architect completely. Instead, the goal is to automate checking for the more trivial mistakes, leaving the architect free to judge the designs on the more complicated and less objective criteria, such as aesthetics.

An automated system for the mass customization of buildings requires several components, among which a CAD interface for non-expert users, a way to show buyers different design alternatives, and a way to check whether the buyer-submitted designs comply with building codes, architects’ design intentions, etc. In this paper, we will focus on the latter of these three components, and more specifically on the way in which architects enter these rules into the system. An algorithm that allows architects to specify the building rules in natural language is presented and tested against a set of rules found in existing building legislation.

II. PROBLEM DEFINITION

There are many rules that apply to building a house; from building codes (e.g. houses in this area can be no taller than 3 stories) to building physics (e.g. rooms must have enough natural ventilation) to basic common sense (e.g. every door should have enough room in front of it to open). We will refer to such rules as constraints. In order to validate a design, all these constraints have to be checked. Since building projects involve large amounts of constraints, such a process has to be automated. This requires that both the building and the constraints are available in a in a machine readable model. There are several examples of such models in the building industry; CAD packages have an internal building model and open exchange standards such as the IFC (Industry Foundation Classes) enable interoperability among them. Automated constraint-checking systems, however, have not received much attention to date.

An automated constraint system has two main requirements: new constraints should be easy to enter for humans, since every design will have its own specific constraints (as well as constraints that are shared between projects, such as legislation) and all the constraints need an unambiguous interpretation.

III. RELATED RESEARCH

Mass customization is an increasingly popular concept in the building industry [4, 10, 24]. Predominantly, mass customization is offered in the form of multiple-choice options [12, 21]. Using constraints to enhance this process, while common in the mechanical engineering industry [1, 5, 11], has yet to receive widespread adoption in the building industry [19]. The only commercial CAD package that features constraints is Revit [23], and it only handles geometrical constraints. Insofar as constraints are adopted, they are typically focused on constraint solving – having the system generate a design that satisfies all the rules – [8, 15] as opposed to constraint checking. Although adopting constraints for checking designs made by professionals is slowly gaining traction [25], doing the same to designs created by non-expert users is still fairly rare. One of the few projects based on constraint checking in use is the Dutch Digital Dormer [17] initiative, which allows the digital submission and pre-approval of building applications for dormers by laymen.
Little research has been done in terms of expressing constraints in a user-friendly way. The SMARTcodes system by Wix et al. [25] represents constraints in XML, which is not particularly easy to interpret by human users. A Domain Specific Language-based approach [22] fares better, but still mentions training costs as a problem. There are few, if any, instances where natural language processing [16] is used to express constraints. More typical applications for natural language processing include content analysis, such as spam filters [2, 14], machine translation [7, 9] and search engines [6, 18].

IV. CONSTRAINTS

We define a constraint as an assertion about a building design, for instance “All concrete walls should be at least 20 cm thick” or “The distance between the kitchen and the living room should be no more than 5 m”. A constraint can therefore be thought of as a function that takes a building design and returns a Boolean value (true or false) that indicates whether or not the design complies with the constraint. A comparison can be made with quantifications from the field of mathematics. A quantification specifies how many elements of a set satisfy a given predicate. For instance,

\[ \forall x \in \mathbb{N} . \ x \geq 0 \]

means “For all x, where x is an element of the set of natural numbers, x is more than or equal to 0”. The similarity with constraints is easy to see. The “for all” is known as the universal quantifier. The other fundamental quantifier is the existential quantifier, as shown below:

\[ \exists x \in \mathbb{Q} . \ x^2 = \frac{1}{4} \]

This means “There exists an x in the set of rational numbers such that the square of x is one fourth”. This quantification is true since there are in fact two such numbers (\(\frac{1}{2}\) and \(-\frac{1}{2}\)). Although there exist many other quantifiers [3], these two are the most widely used, and they cover a large part, if not the majority, of the use cases in building design.

Representing constraints as functions makes it easy to verify a building design: apply all constraints to the design and show the ones that do not pass to the user. Naturally, this requires that the constraint is defined in a way that leaves no room for interpretation. Constraints such as “the architectural quality of the addition must match that of the surrounding buildings” cannot be checked, since “architectural quality” is ill-defined. This means that not all constraints that exist in the wild (legislation, mental models of architects, etc.) will be able to be handled. Fortunately, however, a large percentage can be. The proposed system will therefore not obsolete human judgement, but supplement to allow people to focus not on dull tasks that can be automated (wall widths, sound isolation levels, etc.), but frees them up to judge things like composition and aesthetic quality.

Important to note is that we do not use constraints for constraint solving, i.e. letting the computer generate a design that satisfies all of the constraints. This is done for practical reasons (the solution space for a building is simple too large to be able to generate designs in a reasonable time frame), but also for philosophical reasons: the buyer should be able to modify the design himself, and not merely hope that the computer generates something acceptable.

V. CONSTRAINT PARSING

As mentioned, constraints should be easy for architects to implement. In a previous phase in the project, we designed a system [20] where constraints would be assembled from a series of jigsaw pieces (see Fig. 1), each containing one or multiple words. This system had the advantage of limiting the grammar that was used, making it easy to interpret the constraints, while still producing sentences that were valid English, which in turn made them easy for people to understand. A field test with architects, however showed that this input method was too slow to be used in practice.

Instead, we tried to see how close we could get to the method that is currently used in legislation to define building constraints: natural text. The problem with natural text is that it
is a much richer grammar and vocabulary than computer languages or the jigsaw-piece approach we had been using, not to mention problems like spelling errors and grammatical mistakes which make correct interpretation even harder.

In the field of natural language processing, two main approaches exist for interpreting language: symbolic and stochastic processing [13]. In symbolic processing, a fixed system of rules (e.g. a parser) is used to interpret the input. Stochastic processing, on the other hand, is a statistical approach that requires a pre-assembled body of knowledge. Depending on the specific aspect of natural language processing in question, this can be a corpus (word tagging), a treebank (parsing), or an ontology (semantics), among others. While in theory more powerful than symbolic processing, particularly for resolving ambiguities, stochastic processing’s main requirement, the availability of a body of knowledge, presents a problem in this case. The constraints are entered in Dutch, which has far fewer corpora available than English. Additionally, since the system operates in a niche (the building industry), a large percentage of the terminology used (e.g. mansard roof, dormer) cannot be found in the general-purpose corpora. This means that no adequate corpus is available to us, forcing us down the symbolic path, at least for the core of the algorithm. At a later stage it will likely be necessary to add a certain degree of stochastic processing to resolve ambiguities.

Inspiration for a solution that does not depend on an existing database was found in the domain of HTML parsing. HTML (HyperText Markup Language) is the language in which web pages are written. An HTML document is a tree where each element is either a leaf or a node containing multiple trees. A node consists of an opening and a closing tag, with the contained trees between them.

```html
<html>
  <head>
    <title>Document title</title>
  </head>
  <body>
    <p>A paragraph of text</p>
  </body>
</html>
```

Parsing an HTML document is fairly simple, provided the document is syntactically valid. Unfortunately, a very large percentage of the web pages on the internet are not. Perhaps they are handwritten and the author forgot to close a tag, or made a typo, or perhaps the application generating the HTML pages does not follow the standards correctly. For whichever reason, sticking with a strict policy of “only syntactically valid pages will be accepted” is not an option. Therefore, browsers and other tools that are required to work with HTML as it is found in real-world cases, have to be far more lenient, and make assumptions where the available information is insufficient.

In our approach, we assume we are working with “language soup” (a term inspired by the HTML parser TagSoup); correct grammar is not expected and some words are expected to be missing. The algorithm consists of four steps, which will be described in the following paragraphs.

A. Tokenization

The first step is to convert the input string into useable parts. Basically, this means converting the input to a list of words and numbers by splitting the string on spaces and getting rid of punctuation. This is a simple and basic procedure.

B. Word Lookup

Next, we need to know what the individual words mean. To do this, a database is kept that maps words to their meaning. For example, both “height” and “high” map to the height property of an element. This step results in one of three possibilities for every word: either the word has an associated meaning in the database, the word is marked as ignorable (for words that contain no relevant information, such as prepositions and articles), or the word is not found in the database. This information is shown to the user, so that relevant words that do not have a meaning yet can be given one. Ignorable and unknown words are discarded for the subsequent steps. As an example, in the sentence “the height of the door must be less than 3 meters”, the relevant words are “height”, “door”, “less”, “3” and “meters”. Currently, a word can only have a single meaning. In a future version, this will have to be amended to deal with ambiguous words.

C. Tree Construction

After identifying the relevant words, they are turned into a tree by splitting the sentence based on priority. This works as follows: First, find the token in the sentence with the highest priority (the table below shows the priorities of a few different operations). The table is largely based on the fixity of operators in existing programming languages, such as Haskell.

```
<table>
<thead>
<tr>
<th>Priority</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Boolean operators (e.g. and, or)</td>
</tr>
<tr>
<td>4</td>
<td>Comparison (e.g. more than)</td>
</tr>
<tr>
<td>3</td>
<td>Addition and subtraction</td>
</tr>
<tr>
<td>2</td>
<td>Multiplication and division</td>
</tr>
<tr>
<td>1</td>
<td>Relationship between two elements (e.g. distance)</td>
</tr>
<tr>
<td>0</td>
<td>Properties (e.g. height), Elements (e.g. wall), etc.</td>
</tr>
</tbody>
</table>
```

If multiple tokens have the same priority, choose the first one. Create a tree node with the word in question as a value and a left and right branch with the words before and after the word, respectively. Then apply this algorithm recursively to both branches. The algorithm stops when all the words in a branch have priority 0. As an example, the sentence “The width of the window must be more than 2 times the height of the window” results in the tree shown in Fig. 2.
D. Tree Sanitizing

Since the tree created in the previous step is based directly on the user’s input, it will likely need to be sanitized before it can be used by the program, since the wide range of possible grammars means that words are likely not in the place where we want them. The sentence “the height of the door must be more than that of the window and less than that of the wall”, for instance, is tokenized as: “Height”, “Door”, “>”, “Window”, “And”, “<”, “Wall”. Converting these tokens to a tree results in the tree shown in Fig. 3.

Although correct in structure, there are two problems: the “Window” and “Wall” leaves lack the height property (since we want to compare their heights) and the left leaf of the right branch is missing entirely. In order to fix this, we need to sanitize the tree. Sanitization is a top-down search strategy [13] that uses a recursive algorithm. We start at the top of the tree and say that we want a constraint. The And node results in a constraint, so we move on to the branches. And itself requires two constraints, so we recursively apply the algorithm to the two branches, but now we pass the right branch as a source for missing tokens to the checking of the left branch and vice versa, since it is fairly common that a required token can only be found in the other branch. Going to the left branch, we find a comparison, which results in a constraint. This is what we want, so again we move on the branches. A comparison requires two values. Again, we add the opposite branch to the list of missing token source branches, so that when checking the (Height, Door) branch we can draw from both (Window) and (< Height Wall). To see whether (Height, Door) can produce a value, we check a list of “merge rules”: a list of rules that define what tokens are required to produce a node of a certain type. A merge rule is essentially the same as a production from a context-free grammar [13]. These merge rules are also used to determine the result type and required branch types in the first two steps. For instance, one of the merge rules is Comparison + Value + Value = Constraint.

In the case of the current leaf, we go through the rules and find one that says Property + Element = Value. We have all the required tokens, so we can continue. For the right branch, we also need a value, but all we have is an element. There is no merge rule that allows us to create a Value from nothing but an Element, so we need additional tokens. We take all the rules that produce the correct type and for which we have at least one of the required input tokens. In this case, only the Property + Element rule used earlier satisfies those two conditions, so we need a property token. This is where the sources for missing tokens we mentioned earlier come in.

Whenever we need a token we search the other branches for a token of the corresponding type. When multiple options are available, priority is given to tokens that are in a branch on the same side as the current one, as word omissions in repeated structures usually retain the structure of the original; e.g. in the sentence “The dog chases the cat and chases the mouse”, the dog is the one chasing the mouse rather than the cat, since they both appear on the left. If no tokens on the same side can be found or multiple exist, the closest one (measuring the distance using the tree rather than the word order) is chosen, since it has a higher chance of referring to the correct scope.

In this case, Height, being the only Property node available, is added to the Window leaf. We are now done with this branch, so we move on to the right branch of the And node. Again, we find a comparison, which, as before, is correct, so we will move on to the branches. For the missing left branch we try to find a completed leaf that can produce the appropriate type. Both (Height, Door) and (Window, Height) are suitable, but (Height, Door) is preferred since, like the missing branch, it is on the left. Finally, the right branch again requires a Property token. Now there is a choice of three Height properties, which all produce the same result. The final tree now looks as shown in Fig. 4.

In the implementation, one more transformation is applied to this tree by converting the nodes into objects for easier use later on (the main result of this is that the tokens in the leaves are ordered consistently and that extraneous tokens are eliminated), but this is a trivial step.
VI. PROTOTYPE

The prototype is a direct implementation of the algorithm described in the previous section. The interface, shown in Fig. 5, consists of a text box to type the constraint into, followed by the results of parsing the sentence. The first part of the result is the list of recognized words. Clicking on one of these words allows the user to ascribe a meaning to it, which is mainly used to enter new words into the system. Below that is the initial tree, followed by the end result. Aside from the interface, it is also possible to check the result of parsing a constraint programmatically to facilitate easy testing of multiple constraints. This prototype was used to parse a series of 42 constraints taken from the Dutch legislation regarding dormers from both the national regulations and the building codes of the municipality of Rotterdam.

VII. EVALUATION

Out of the initial 42 constraints, five are too vague to be formalized (e.g. referring to the architectural quality of a dormer), three preclude the use of the system (if a certain condition holds, the design must be submitted to a committee), one is susceptible to multiple interpretation and two are outside the current scope of the system (e.g. referring to neighbouring buildings while we currently only focus on one building). This leaves 31 constraints for our tests. For each of these 31 constraints, we specified the desired resulting tree. We than ran the algorithm to see if the correct result was produced. The results of this test were as follows:

3 constraints were interpreted correctly as they were written. They are (translated to English): “The height of the dormer must be less than 1.50 m”, “The distance of the dormer to the front façade must be more than 1 m” and “Profile width frames dormer max. 0.07 m.”

6 constraints required minor modifications to be interpreted correctly (usually adding one word). (Translated) examples include “the width of the dormer must be less than 1/3 the width of the dwelling” (there is no explicit token to indicate that 1/3 and the width of the dwelling must be multiplied. Can be fixed by saying “1/3 times the width of the dwelling”), “Height dormer: <= 1.5 m (incl. roof edge)” (since the height of the roof edge is on the right, it gets added to the 1.5 m rather than the height of the dormer. Can be fixed by saying “Height dormer (incl. roof edge): <= 1.5 m”) and “Walls: dark grey” (there is no explicit token to indicate that this is an equality. Can be fixed by saying “Walls: equal to dark grey.”)

3 constraints required heavy modifications, to the point of no longer being grammatically correct. This shows up in relationships between elements, such as “the distance between a and b”. Here, both a and b get placed in the right branch of the distance node, resulting either in “the distance between a and a” or “the distance between c and a”, depending on whether or not another element c exists in the tree. This problem will be fixes in a next version of the system.

13 constraints were not yet interpreted correctly. These constraints all require additional constructs in the tree representation that have yet to be implemented. Most of them should become interpretable by adding just a few constructs. By implementing conditionals (if-then-else), element filtering (e.g. walls that are made of concrete) and existentials (e.g. no dormers are allowed on roofs with a low angle), 10 out of those 13 should become interpretable.

For the final 6 constraints, we have yet to determine what the desired interpretation should be. One example of such a constraint is “The limited size of the half hip on a gable roof is unsuit for additions. The sides of the roof are more suited for this and are to be treated as a saddle roof.”

This results in a 29% being interpretable with little to no modification, with an additional 42% percent scheduled for implementation in the near future. The remaining 29% will require additional algorithm tweaks, constructs, and/or additional systems.

VIII. DISCUSSION

Results obtained from the dormer constraints test case study described here showed that using natural language to express constraints is feasible to a certain extent. While no full evaluation can be given at this point since the research is still ongoing, the initial results are promising. At present, 29% of our test suite of constraints are interpretable with little or no modification, which is expected to rise to 71% in the near future.

The test case that was used so far tests the performance of the system on constraints as defined by municipality employees. In a future experiment, we will test its performance on constraints provided by architects. In order to do this, we will provide a group of architects with a flawed building design. Their task will be to define constraints to prevent these errors. These constraints will form the second constraint test suite. As with the first suite, we will check whether the system’s interpretation matches the desired one.
REFERENCES