Advanced Engineering Informatics 28 (2014) 28–36

Contents lists available at ScienceDirect

Advanced Engineering Informatics

journal homepage: www.elsevier.com/locate/aei

Freedom through constraints: User-oriented architectural design

R.A. Niemeijer, B. de Vries *, J. Beetz

Eindhoven University of Technology, P.O. Box 513, 5600 MB, The Netherlands

ARTICLE INFO

Article history:
Received 1 May 2012
Received in revised form 22 July 2013
Accepted 12 November 2013
Available online 5 December 2013

Keywords:
Natural Language Processing
Design constraints
Automated interpretation

ABSTRACT

In this article we report on validated research for the construction of design constraints by automated interpretation of natural language input. We show how our approach of dynamic reconfigurations of parsed syntax trees using a number of production rules is used to formalize and transform natural language constructs into computable constraints that are applied to concrete building information models. The calibration and validation of the proposed algorithms and production rules is based on two test data sets: The verbatim text of Dutch building code regulations for dormer extensions on existing roofs and constraints formulated ad hoc by design students based upon a series of example designs. We show how a prototypical implementation of our approach can be used to interpret 44% of the test data without human interference and how the remaining sentences can be interpreted with minimal additional effort or further development.

1. Introduction

Currently, customers buying a new house are typically limited to two options: either an architect's design is bought without modifications, or a discrete set of limited design alternatives is offered, such as two different kitchen types or the optional addition of a dormer. This customization, however, is very limited, as all design alternatives offered by the architect have to be completely designed up front. Consequently, customers do not get the exact house they want. In many cases this means that house owners immediately start remodelling after the house has been built to get the house they actually wanted. This is a very inefficient state of affairs, leading to unnecessary increases in cost and waste. It would be preferable for buyers to be able to make more extensive changes to the design of the building in the design phase already, so that they can get the house they want, eliminating the need for an additional remodelling step.

In many different industries, the ability to customize a product has become commonplace, with examples ranging from fast food to clothing to the car industry. In the building industry, however, adoption of this practice has lagged behind. At least two reasons for this lack of customization can be identified: (a) The tradition of the architect being the sole designer of the product and an implied artistic autonomy. (b) The large amount of regulations that apply to buildings. Over time, mass customization and participatory design have been applied to the building and construction sector [36,35]. In most cases, though, the amount of flexibility is limited, since the two traditional ways of offering mass customized housing—entirely customized design or choices from predesigned alternatives—result in a trade-off between the freedom of choice among design alternatives and the amount of time required to design them. When creating a design for a consumer product, many rules must be obeyed [18] which in the case of buildings stem from building codes, regulations and design requirements. In this paper, rules are referred to as constraints [26,38,15] composing a Constraint Satisfaction Problem (CSP), that [37] defined as "a problem composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can simultaneously take."

In current practice, most of these constraints are checked manually which results in labor-intensive and error-prone processes. Hence, the automation of formulating, processing and checking of constraints has been of great interest for researchers and practitioners from early years of computational support onwards. Initial contributions by Fenves [13], Fenves and Garret [14] have sparked a large body of research in this area. These include the developments of AI-based Expert Systems [34,20] as well as Knowledge Based Systems based on Frames [11,10] and Predicate, First Order and Description Logic [33,17]. With the advent of object-oriented Building Information Models (BIM) and particularly the Industry Foundation Classes (IFC), such systems have flourished considerably [19,24,40]. Recently, the incorporation of methods and tools from the Semantic Web initiative led to further advancements in this research field [3,39,32,41]. A concise overview of these developments as well as commercial implementations in the building industry can be found in [12].

* Corresponding author. Tel.: +31 402472388.
E-mail address: B.d.Vries@tue.nl (B. de Vries).

1474-0346/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.aei.2013.11.003

Most of these developments, however, are based on complex logic languages or conventional high-level programming, only allowing users with considerable ICT knowledge to specify, formalize and encode constraints. Secondly, most requirements engineering systems demand prior domain knowledge embedded as corpora in databases. In order to enable ad hoc, per-project formulations of computable constraints by design and engineering end-users, alternative methods are required to address these needs. The goal of this research is to find a method of constraint formulation and entry that is usable by end-user practitioners such as engineers and architects and that generates unambiguous computer interpretable constraints. The research builds upon existing Natural Language Processing (NLP) techniques and provides new technological insights and contributions in the context of architectural design constraints.

The outline of this paper is as follows. First we will discuss NLP in requirements engineering, NLP in building constraint entry and our approach: parsing without a corpus. Following that, we present our prototype system "ContraintSoup". The system's algorithms and components that constitute the prototype are explained in detail. The interface of the prototype is discussed only briefly. We then report the results of a series of experiments measuring the ability of automated interpretation of architectural constraints provided in the form of natural language as the input to the system. Finally, conclusions are drawn on the advantages of the proposed method and how the system can be developed further for improved performance.

2. Constraint and rule formalization

2.1. Natural Language Processing in requirements engineering

Natural Language Programming (NLP) originates from the desire in software development to use natural language instead of computer scientist specific programming language such as Java and C#. The advantages are obvious, namely a direct interaction between the domain expert and the computer that executes the expresses procedures. Despite the fact that many 'high level programming languages' have been developed automatic programming is still unresolved. [2] summarized four components of automatic programming: a means of acquiring a high-level specification (requirements), a mechanism for requirements validation, a means of translating the high-level specification into a low-level specification, and an automatic compiler for compilation of the low-level specification. Since then, so-called Very High Level Languages (VHLL) were developed, such as BIDL [27] that were in fact pseudo-natural languages with unambiguous syntax and semantics. At that time with the advent of Object Oriented Programming (OOP), many researchers (e.g. [31]) NLP in requirements engineering is seen as a part of the OO model generation process. This process can be viewed as a sequence of processing steps that starts from a raw text and proceeds to computer interpretable code. Berzins et al. [4] provide a model for NLP that recognizes four steps: (1) tokenization, (2) synthetic parsing, (3) semantic processing, and (4) pragmatics.

The first part of this process is called Part-of-Speech Tagging [5], and is performed by using a large sample of tagged text, which is referred to as a corpus [22,6,28,8]. In tagged text, every word has been assigned its proper part of speech. For every word in the input text, the algorithm searches the corpus to find the correct part of speech for that word.

The second step of NLP analyses larger chunks of a sentence than individual words. The POS information from the previous level is used but in combination with preceding and succeeding words. Using the semantic meaning of some of these words, hierarchical trees are constructed. Parsing methods apply often statistics and rely upon a corpus of training data (words) that are labelled with a specific meaning necessary for tree construction.

The third step is semantic processing. In this step ambiguity, which is common in natural language, is addressed. An example often used to illustrate this is the following sentence:

"Time flies like an arrow, but fruit flies like a banana."

The first occurrence of "flies" is a verb, but in the correct interpretation the second occurrence is a noun. This is an example of a case where one word has more than one possible part of speech, as hinted on in the paragraph on corpora. The deterministic approach stops working here, since the interpretation involving flying fruit would be grammatically correct, but incorrect from the perspective of common sense. The common solution for this problem is to use a statistical approach, where each production rule of the grammar (a rule that governs how parts of the sentence are combined, e.g. verb + adverb = verb phrase) is also attributed with a relative frequency with which it occurs. These types of grammars are referred to as Probabilistic Context-Free Grammars (PCFG) or Stochastic Context-Free Grammars (SCFG) [7,8]. Using these frequencies, the parser can check the neighboring words to determine the likelihood of a given interpretation scenario.

Finally, in the fourth step (pragmatics) is concerned with the more complex linguistics issues, such as resolving what a pronoun or noun refers to.

In [4], four challenges for using NLP in requirements engineering are listed:

(1) Ambiguity of word meaning and scope: words can have multiple meanings, and phrases and adjectives can refer to multiple words.
(2) Computational complexity: the possible need to check an exponential number of parse trees.
(3) Tacit knowledge and anaphora resolutions: the difficulty in resolving reference words such as 'they' when there are multiple possible targets without knowledge of the properties and behaviors of those targets.
(4) Non-linguistic context: information about the stakeholder, time of day, recent events, etc.

When it comes to word meaning, ambiguities can be partially resolved due to the fact that the system operates in a domain-specific context. The word column, for instance, could refer to a vertical list of figures in a spread sheet. In the context of building constraints, however, it is considerably more likely to refer to a pillar. Similarly, the variation in non-linguistic context is limited since all text processed with the system will be a constraint. The main contributions of the proposed algorithm lie in points 2 and 3. By using unit information and typical sentence structures of architectural constraints, it is possible to resolve references and other gaps in the parse tree without taking exponential time.

2.2. Natural Language Processing for building constraint entry

Currently, the majority of constraints in the building industry are specified using a natural language, such as Dutch or English. Examples of these include building codes and functional requirements in the client's brief. The inherent complexity and flexibility of natural languages, however, makes automated interpretation exceedingly difficult, as it requires not just an understanding of grammar, but also knowledge of a domain and a sense of context. Although NLP has been applied in building and construction in a large variety of areas, among which information extraction from documents is the most prominent (see [25] for an overview), only a limited amount of research has been dedicated to automate the

creation of rules with natural language input. Hjelseth and Nisbet [22] are working on a methodology to manually tag part-of-speech components from natural language building regulations to automate the generation of constraints as an input for code compliancy checking systems based on IFC models. The approach presented here, however, aims at a fully unguided process.

3. System prototype: ConstraintSoup

Given the lack of a suitable corpus of constraints, corpora-based algorithms (e.g. using statistical approaches to determine the most likely function of a word) are not suitable for this project. Therefore, an approach based on combining a deterministic grammar with supervised machine learning was chosen. The supervision consists of providing the algorithm with a dictionary of word functions and meanings and providing the desired outputs of a training corpus. This work has been documented in [30]. Inspiration for this solution was found in the domain of HTML parsing. HTML (HyperText Markup Language) is the language in which web pages are written. In theory, parsing an HTML document would be straightforward, provided the document was syntactically valid. However, a very large percentage of web pages found on the internet are not. Possible reasons include handwritten pages including unclosed tags, typos and HTML-generating applications that do not correctly follow the standards. Adopting a strict policy of only accepting syntactically valid pages will therefore mean that many pages cannot be rendered, which in many cases is not acceptable. Therefore, web browsers and other tools required to work with HTML as found in real-world cases have to be lenient, and make assumptions where the available information is insufficient. The HTML parser TagSoup [9] treats input as "tag soup", in which redundant, missing or incorrectly placed tags are expected and dealt with. Similarly, the algorithm developed in this research treats its input as "language soup". Provided all the words that are necessary for a correct interpretation occur somewhere in the sentence, the algorithm can deal with omitted repetitions, superfluous words and accepts different ways of phrasing constraints, as word order is largely irrelevant. This makes the algorithm very flexible in regards to its input, which is essential when dealing with the fluidity of natural language. We call this method "Architectural Language Parsing". The algorithm is an elaboration of the NLP model presented by Berzin et al. as follows: (1) Tokenization, (2) Synthetic parsing = Word loop-up + Preprocessing + Tree construction, (3) Semantic processing = Tree sanitization, while (4) Pragmatics is not included. The processing steps of our algorithm are shown in Fig. 1 and detailed in the following paragraphs.

3.1. Tokenization

The first step is to convert the input string into useable parts. This means converting the input to a list of separate chunks referred to as tokens such as words and numbers by splitting input strings where spaces and punctuation occur. This is a well-understood and established technique in language processing and is referred to a tokenization.

3.2. Word lookup

After tokenizing the input, the next step is to assign a meaning to the individual tokens. To do this, a database is kept that maps words to their meaning. For example, both the noun "height" and adjective "high" map to the "height" data type property of an element. In the prototype system we have implemented, the database was created manually. The word lookup results in one of three possibilities for every word: (a) the word has an associated

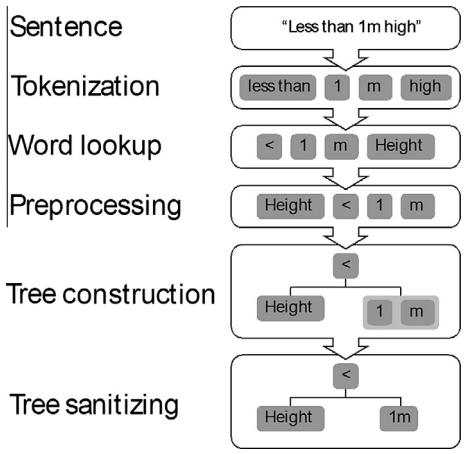

Fig. 1. Algorithm steps.

meaning in the database, (b) the word is marked as ignorable or (c) the word is not found in the database. Examples of ignorable words include articles, some prepositions, such as "of" in phrases such as "the height of the house", since no preposition is needed to establish the link between the noun and the adjective, and "because", since for the purposes of constraint checking reasons are of little to no value, Depending on whether or not the unknown word is required for a correct interpretation of the sentence, the user will have to add the word to the database and specify the meaning. The meaning of a word mainly consists of the type of the word, such as "Element" or "Property", with optional additional information depending on the type of the word. Values, for instance, also have an associated unit. The word types implemented in the prototype are shown in Table 1.

Comparative adjectives are interpreted as the associated adjective followed by a comparison. For example, "higher" is replaced with the property "height" and the comparator "greater than". Qualitative adjectives are either not sufficiently objective ("big", "expensive", etc.) or require element filters, which will be discussed in later sections. Other word types, such as verbs, are ignored; words like *shall*, *must* and *needs to* are not needed since constraints are positive by default. In negations (e.g. *must not*) the relevant word is not the verb (*must*) but the adverb (*not*). Contractions (e.g. *shall not*) must be expanded before this step.

3.3. Pre-processing

After identifying the relevant words, the sentence is transformed by reordering, adding or removing words to cover cases that are not yet processed correctly by the algorithm. These transformations are specified in a custom Domain-Specific Language (DSL) that operates on the token level. These expressions are then converted to proper regular expressions and applied to the sentence. As an example,

Table 1
Word types.

Word type	Examples
Element	Wall, door, window
Unit	Meter, millimeter, degrees
Value	2, 3.5, zinc
Relationship	Distance
Comparison	More than, equal to
Operator (Add/subtract)	Plus, minus
Operator (Multiply/divide)	Times, divided by
Operator (Boolean)	And, or

- (\i / \i) of

is replaced by

- `$1 times`

where \i means an integer and $1 is a back-reference, i.e. it is replaced with the contents of the first matched group in the pattern, indicated by the parentheses). As an example, "1/3 of the house's height" is replaced with "1/3 times the house's height". This rule is required because humans know that the two values are to be multiplied together, but this information is not stated explicitly in the sentence. Inspiration for this DSL was found in regular expressions, which is a commonly used DSL to search or replace text. The syntax is borrowed from regular expressions, with the following additions:

- \i matches an integer,
- \s matches a literal string, i.e. a piece of text between quotes,
- . matches any token,
- words in the pre-processing rule match Word tokens containing that string.

In order to facilitate these changes, these DSL rules are processed using regular expressions to produce proper regular expressions, which are applied to the string containing the tokens.

3.4. Tree construction

After identifying the relevant words, the list of words is converted to a tree by splitting the sentence based on priority. This works as follows: First, find the token in the sentence with the lowest priority (the table below shows the priorities of a few different operations, with 0 being the highest priority). The priorities of tokens are the same as in mathematics and many programming languages, which based on observation of examples also appear to apply to natural language parsing. Properties (function application) and elements (variables) have the highest priority, followed by arithmetical operators, comparisons and Boolean operators.

5	Boolean operator (e.g. and, or)
4	Comparison (e.g. more than)
3	Addition and subtraction
2	Multiplication and division
1	Relationship between two elements (e.g. distance)
0	Properties (e.g. height), Elements (e.g. wall), etc.

If multiple tokens have the same priority, the first one is chosen, based on the observation that natural language sentence typically use a left-to-right order. A tree node is created with the word in question as a value and a left and right branch with the words before and after the word, respectively. Then this algorithm is applied recursively to both branches. The algorithm stops when all the words in a branch have priority 0. The tree construction algorithm in pseudocode:

```
function makeTree(words):
ps = words sorted by ascending priority
if all words in ps have priority 0: return
  Leaf(words)
h = first element of ps
l = words to the left of h
r = words to the right of h
return Node(makeTree(l), h, makeTree(r))
```

Fig. 2 shows an example of these steps for the sentence "The width of the window must be more than 2 times the height of the window".

3.5. 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 system. The wide range of possible grammatical constructs means that words are likely to be in an undesirable position. 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 using production rules described in Section 3.4 results in the tree shown in Fig. 3.

Although correct in structure, there are two problems: leaves D and F lack the *Height* property (which is needed to compare their heights) and the leaf E is missing entirely. In order to fix this, the tree must be modified, in a process that in this paper is referred to as *sanitizing*. For sanitizing the tree, a top-down recursive search strategy [1] is used. The algorithm starts at the top of the tree looking for a constraint. The *And* node results in a constraint (since it is a combination of two constraints), so the algorithm moves onto the branches. *And* itself requires two constraints (i.e. two statements that evaluate to true or false), so the algorithm is recursively applied to the two branches. Now branch B is passed as a source for missing tokens to the checking of branch A and vice versa, since based on observations it is common that a required token can only be found in the other branch. Going to the branch A, a comparison is found, which is one type of constraint. A comparison requires two values. Branch D is added to the list of branches that are sources for missing tokens, so that when checking the branch C both (*Window*) and (<*Wall*) can be used. To see whether (*Height*, *Door*) can produce a value, a list of production rules [23] is checked. These production rules, which state the manner in which tokens can be combined to form new tokens, are also used to determine the result type and required branch types in the first two steps. To illustrate this concept, three examples of productions rules are provided here:

- $Property(which\ produces\ a\ value) + Element = Value$
- example: height + door = height of door
- $Relationship\ \ (which\ produces\ a\ boolean) + Element + Element = Constraint$
- example: above + roof + floor = roof must be above floor
- $Operator\ (with\ precedence\ of\ \ add/subtract) + \ Value + Value(of\ the\ same\ type\ as\ the\ first\ value) = Value$
- example: plus + 1000 mm + 2000 mm = 3000 mm

Note that in the second example the order of the words is important, as reversing the positions of *roof* and *floor* would produce the opposite outcome. While the algorithm is largely insensitive to local word order, the overall order of the branches still has to be correct. Based on observations this is rarely a problem in practice though, as constraints are typically grammatically and logically correct.

In the case of the current leaf (*Height*, *door*), searching the production rules reveals one that says "Property + Element = Value". All the required tokens are now present, so we can continue. For the branch D, a value is needed as well, but the only token available is an element. There is no production rule that allows a Value to be created from nothing but an Element, so additional tokens are needed. All production rules that produce the correct type and for which at least one of the required input tokens is available are considered. In this case, only the "Property + Element" rule

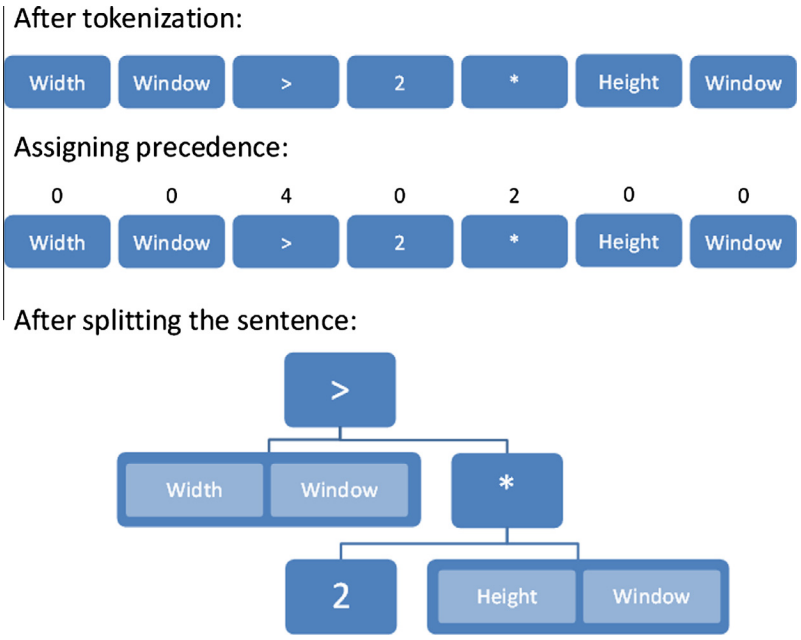

Fig. 2. Tree construction example.

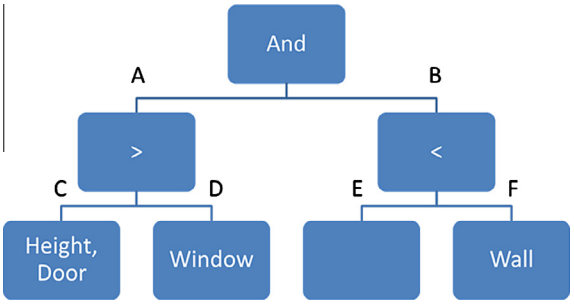

Fig. 3. Example syntax tree.

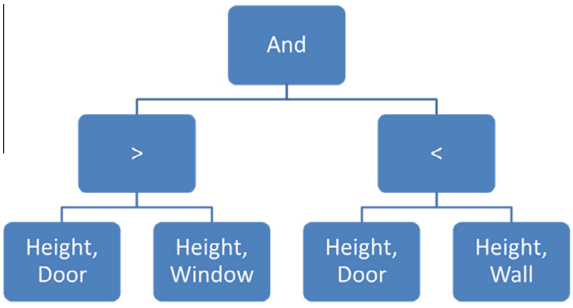

Fig. 4. Final syntax tree.

used earlier satisfies those two conditions, so a property token is needed.

Whenever a token is needed, the other branches are searched 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. Based on observations, 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.

Continuing with the example, *Height* is the only available Property node that can be combined with *Window* in branch D to produce a value, so it is added to the *Window* leaf. This branch is now completed, so the algorithm continues with the branch B. Here again a comparison is found, which is correct as in the left branch, so the algorithm can move onto the branches. For the missing branch E the aim is to find a completed leaf that can produce the appropriate type. Both branch C and D are suitable, but C is preferred since, like the missing branch, it is on the left. Finally, the branch F again requires a Property token. Now there is a choice of three *Height* properties, which all produce the same result. The final tree is shown in Fig. 4.

4. Interface

The algorithm described in the previous section has been implemented in a prototype. The interface, shown in Fig. 5, consists of a text box (A) to type the constraint into, followed by a list of results. The first part of the result is the list of recognized words (B) which is the result of the tokenization, word lookup and pre-processing steps. Clicking on one of these words allows the user to ascribe a meaning to it by choosing from the available token types, which is used to enter new words into the system (see Table 1). Below that is the initial tree that results from the tree construction step (C), followed by the end result after sanitizing the tree (D). In order to make the end result more easily readable, the tree is also displayed as an English sentence (E). For illustration purposes, a floor plan is presented at the right hand side (F) in which the walls in conflict with the constraint entered by the user are colored in red.

5. Evaluation

In order to get a representative sample of constraint input, a user test was conducted with architecture students. Each student was shown a random selection of five out of a total of ten scenarios that each depicted an undesirable building design. Fig. 6A–D shows four of the scenarios presented.

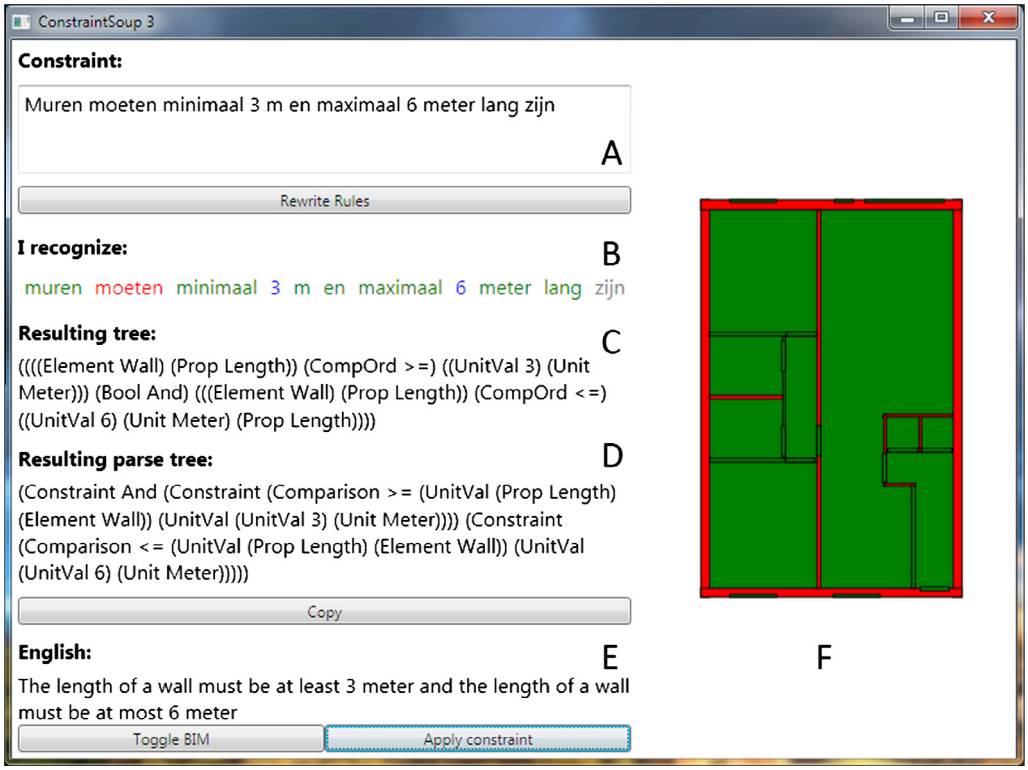

Fig. 5. Prototype interface.

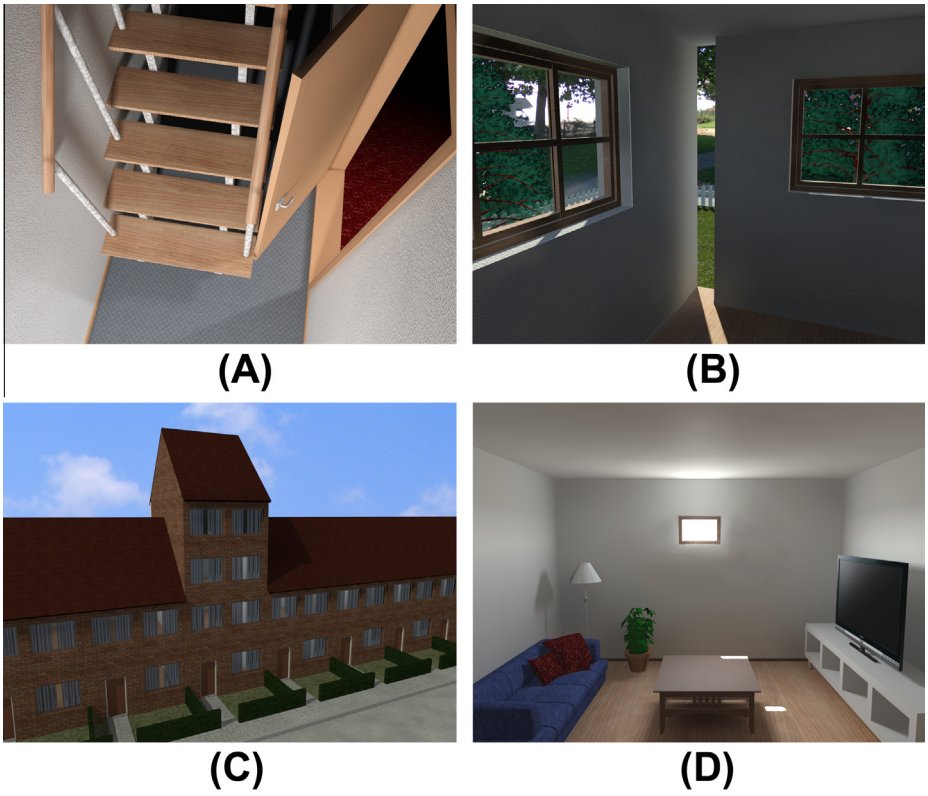

Fig. 6. (A) A door that cannot be opened. (B) Misaligned walls. (C) A house that is too high. (D) A window that is too small.

The students were asked to formulate one or more constraints for an automated constraints checking system to prevent these problems. No details of the system were provided and no limits were placed on the way the input should be formulated, save for the fact that the constraints were requested to be objectively decidable, as this is a prerequisite for any constraint checking system. 47 students participated in the test, yielding a total of 292 constraints. Of these 292, 89 were rejected for being insufficiently

Table 2
Constraint categories.

Constraint category	Amount	Percentage (%)
Viable	128	44
Not objective	89	31
Difficult/ambiguous	65	22
No constraint needed	10	3

Table 3
Valid constraint categories.

Valid constraint category	Amount	Percentage (%)
Correctly interpreted	53	41
Minor change required	23	18
Refers to neighboring houses	17	13
More grammatical constructs required	14	11
No desired encoding yet	14	11
Not yet handled correctly	7	6

objective. Examples include "window at eye level for better view" (the eye level varies from person to person) and "place higher fence", which fails to specify how much higher. An additional 65 were rejected for being either difficult to verify in current generation BIMs such as IFC or Revit (e.g. non-trivial spatial relationships) or due to ambiguous language, such as "it should be possible to open a door at least 90 degrees" and "the window should be as large as the least wide wall in the room" (which does not specify whether this applies to just the width or the entire surface area of the wall). In 10 cases, the participant did not consider the design to be problematic. This leaves 128 constraints out of the original 292, or 43.8%, that are can be interpreted by the system, meaning that even with no training or explanation of the system whatsoever, almost half of the constraints are suitable for the proposed constraint checking system. This indicates that, perhaps with a short training, architects should have little difficulty in formulating proper constraints. Table 2 shows the distribution of the different categories of constraints mentioned above.

The next step was to specify the desired syntax tree for the eligible constraints. This was done for 83 of the 128 constraints. The other 45 constraint fall in three categories: 14 require grammatical constructs that have not yet been implemented. Another 17 refer to the neighboring houses, for which a satisfactory encoding has yet to be found. The remaining 14 also lack a desired resulting tree, since in these cases the wording used was more metaphorical or context-sensitive. One example of this is the constraint "At least 6000 lux must enter the living room", in which no mention is made of the window through which this should occur. Of the 83 constraints, 53 were interpreted correctly, meaning they can be used for checking building designs. This was tested by evaluating the constraints on a simple building model. 23 more could be handled correctly with a minor change in wording, mostly because the algorithm in its current state has some trouble with constraints specifying equalities rather than inequalities. For example, "The walls of the dormer must be opaque" needs to be changed to "The walls of the dormer must be equal to opaque". This could be remedied by assuming equality by default unless otherwise specified. The remaining 7 constraints were not yet handled correctly, largely due to the occurrence of complicated spatial relationships, such as "at least 1 m of the roof should remain free". This means that of the constraints for which a desired parse tree was formed, 91.6% could be correctly interpreted with little or no changes. Table 3 shows this breakdown of the viable constraints. This means that of the 282 constraints formulated by the students (292 minus the 10 cases where no problem was found), 27% can already be automatically verified even with this early prototype.

6. Conclusions and discussion

Referring to the four components of automatic programming: (1) a means of acquiring a high-level specification (requirements), (2) a mechanism for requirements validation, (3) a means of translating the high-level specification into a low-level specification, and (4) an automatic compiler for compilation of the low-level specification, the following conclusion can be drawn. The presented architectural language parsing algorithm addresses component (1) and (3). Validation of the architectural requirements is devoted to application programs that can check whether the stated requirements are met, such as cost calculation and climate calculation. Compilation of the generated code by our algorithm is straightforward and can be handled by many existing programming language compilers.

The success rate of our algorithm is significantly lower than other NLP algorithms [8,33] which can be explained by the absence of a database with corpora. In the architectural design process elicitation of requirements by the client is regarded a crucial part of the whole building process. A fully automated process of requirements specification through text ignores the importance of face-to-face communication between the client and the architect. It is unlikely that all knowledge that is exchanged in a conversation can be captured by written constraints. However, a substantial part of these constraints can be captured without additional effort or knowledge, which narrows down the unwanted design solutions considerably.

The presented algorithm provides the following technological contributions to the field of NLP in requirements engineering:

- No full language grammar required

Unlike most NLP algorithms, ours' ignores word order, which means that no full grammar for the language being modelled needs to be developed. Therefore the algorithm can be easily implemented in a given application and language.

- Robustness

Aside from not being very sensitive to the precise word order, the algorithm is also able to work around some other potential problems, such as omitted repetitions and superfluous words. Most existing NLP approaches cannot handle such problems that occur often in natural language.

- No corpora needed

Whereas most NLP algorithms need a domain specific corpora to operate, ours' works without. In many application domains, like design constraints where corpora do not exist, the proposed algorithm can be applied.

The presented algorithm advances the field of architectural design constraints processing through the following results:

- Proof under realistic conditions

The experiments presented in this paper prove that NLP is a applicable method for real-time processing of typed natural language design constraints and converting these into a computer interpretable format.

- NLP interface embedded in a design system

The presented prototype system used in the experiments, shows how NLP can be embedded in architectural design systems. The interface allows for constraints specification and evaluation during the design process in a natural manner.

In addition, although no research has yet been performed to confirm these, the following strengths of the algorithm have been identified:

- Applicable to multiple industries

Since no domain-specific information is used, the algorithm can be used in any industry where it is useful to specify constraints using natural language, whether for the purposes of mass customization or as a support tool for a professional designer.

- Little training required

Because constraints are entered in natural language there is little to no required training on the part of the users. This represents a large improvement over current common practice, where constraints are often specified in a programming language or DSL.

This algorithm was developed based on the sentence structure of the Dutch language. It is expected that the algorithm will also work for English, since the internal representation of sentences is in English. As both are West Germanic languages, they possess a similar grammar. The main difference is the word order in subordinate clauses—Dutch uses subject-object-verb and English uses subject-verb-object. Compare for instance "When Bob ate the apple..." with "Toen Bob de appel at..." The impact of this, however, is limited since subordinate clauses are not very common in constraints. In a different test that used unmodified constraints take from the building codes for dormers in the municipality of Rotterdam, only two of the 31 constraints contained a subordinate clause. Aside from that, the algorithm is flexible in regards to the word order, which further reduces the problem. Although this has not been tested, it is likely that the algorithm will also work for other Western European languages, due to the insensitivity to the specific word order. Languages with a significantly different structure (such as right-to-left and top-down languages) will in all likelihood require changes to the algorithm, particularly to the order in which neighboring branches are traversed in search of missing tokens. While no tests have been conducted to determine how the algorithm scales with larger input, the importance of this is expected to be limited. The constraints used to test the system can be considered representative of those that will be used in practice, both in terms of content and in length.

Although the algorithm for interpreting natural language constraints developed in this research project has proven to be a promising start, there is still a considerable amount of work to be done before a constraint-based mass customization system can be effectively deployed, both in terms of the system itself as well as the building industry as a whole. The main limitation of the prototype that was developed as a proof of concept in the context of this research is the fact that the parser grammar has not yet been fully developed. There are two main omissions. The first is that several common grammatical structures, such as conditionals (e.g. "if the angle of the roof is more than 30°, no dormer is allowed") and existential constraints (e.g. "walls must contain at least one window") are not yet implemented. Adding these should be straight-forward. Secondly, the algorithm must become probabilistic in order to be able to resolve ambiguities. One way of doing this would be add a corpus to the algorithm, such as the corpus formed by the constraints that were used to test the performance of the algorithm (though a corpus used in practice will need to be many times larger). These constraints, which are annotated with the desired resulting syntax tree, can be used to determine the most likely interpretation of new constraints. Finally, although not essential, it would be beneficial to automatically derive word associations from existing taxonomies, ontologies or dictionaries such as the IFD library [16], though this will only help with building industry-specific terminology and not with general-purpose words.

References

[1] A. Aho, J. Ullman, The theory of parsing, translation, and compiling, Prentice-Hall, Inc. (1972).
[2] R. Balzer, A 15 year perspective on automatic programming, IEEE Transactions on Software Engineering, SE-11(11) (1985) 1257–1268.
[3] J. Beetz, J. Van Leeuwen, B. De Vries, IfcOWL: a case of transforming EXPRESS schemas into ontologies, AIEDAM 23 (2009) 89–101.
[4] V. Berzins, C. Martell, Luqi, P. Adams, Innovations in natural language document processing for requirements engineering, in: B. Paech, V. Martell (Eds.), Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs, Lecture Notes in Computer Science, Springer, Berlin Heidelberg, 2008, pp. 125–146.
[5] E. Brill, A simple rule-based part of speech tagger, in: ANLC '92 Proceedings of the Third Conference on Applied Natural Language Processing, 1992.
[6] P. Brown, J. Cocke, S. Pietra, V. Pietra, F. Jelinek, J. Lafferty, R. Mercer, P. Roossin, A statistical approach to machine translation, Comput. Linguist. 16 (2) (1990) 79–85.
[7] E. Charniak, Statistical techniques for natural language parsing, AI Mag. 18 (1997) 33–43.
[8] M. Collins, Head-driven statistical models for natural language parsing, Comput. Linguist. 29 (4) (2003) 589–637.
[9] Cowan, TagSoup home page, 2011. <http://ccil.org/~cowan/XML/tagsoup/>.
[10] C.L. Dym, R.P. Henchey, E.A. Delis, S. Gonick, A knowledge-based system for automated architectural code checking, Comput.-Aided Des. 20 (1988) 137–145.
[11] J.H. Garrett, S.J. Fenves, A knowledge-based standards processor for structural component design, Eng. Comput. 2 (1987) 219–238.
[12] C. Eastman, J. Lee, Y. Jeong, J. Lee, Automatic rule-based checking of building designs, Automat. Constr. 18 (2009) 1011–1033.
[13] Steven J. Fenves, Recent developments in the methodology for the formulation and organization of design specifications, Eng. Struct. 1 (5) (1979) 223–229, http://dx.doi.org/10.1016/0141-0296(79)90002-6.
[14] S.J. Fenves, J.H. Garrett Jr., Knowledge based standards processing, Artif. Intell. Eng. 1 (1) (1986) 3–14, http://dx.doi.org/10.1016/0954-1810(86)90029-4.
[15] E. Gelle, B. Faltings, Solving mixed and conditional constraint satisfaction problems, Constraints 8 (2) (2003) 107–141.
[16] R. Grant, IFD Library White Paper, Technical Report, CSI and IFD Library Group, 2008.
[17] M.M. Hakim, J.H. Garrett, A description logic approach for representing engineering design standards, Eng. Comput. 9 (1993) 108–124.
[18] J. Halman, J. Voordijk, I. Reymen, Modular approaches in Dutch house building: an exploratory survey, Housing Stud. 23 (5) (2008) 781–799.
[19] Han, J. Kunz, Law, Client/server framework for on-line building code checking, J. Comput. Civ. Eng. 12 (1998) 181–194.
[20] Eric J. Heikkila, Edwin J. Blewett, Using expert systems to check compliance with municipal building codes, J. Am. Plann. Assoc. 58 (1) (1992) 72–80, http://dx.doi.org/10.1080/01944369208975536.
[21] E. Hjelseth, N. Nisbet, Capturing normative constraints by use of the semantic mark-up RASE methodology. In: Proceedings of the 28th International Conference of CIB W78, Sophia Antipolis, France, 26-28 October, 2011.
[22] M. King, Parsing Natural Language, Academic Press Inc. Ltd., London, 1983.
[23] D. Knuth, Semantics of context-free languages, Theory of Computing Systems 2 (1968) 127–145.
[24] T. Liebich, J. Wix, J. Forester, Z. Qi, Speeding-up the building plan approval – the Singapore e-plan checking project offers automatic plan checking based on IFC, in: E-Work and E-Business in Architecture, Engineering and Construction, Proc. of 4th European Conference on Product and Process Modelling, Portoroz, Balkema, Rotterdam, 2002, pp. 467–471.
[25] K.Y. Lin, S.H. Hsieh, H.P. Tserng, K.W. Chou, H.T. Lin, C.P. Huang, K.F. Tzeng, Enabling the creation of domain-specific reference collections to support text-based information retrieval experiments in the architecture, engineering and construction industries, Adv. Eng. Inform. 22 (2008) 350–361.
[26] C. Lottaz, R. Stalker, I. Smith, Constraint solving and preference activation for interactive design, AI EDAM 12 (1) (2000) 13–27.
[27] R. Lu, Z. Jin, R. Wan, Requirement specification in pseudo-natural language in promis. In: Proceedings of the Nineteenth Annual InternationalComputer Software and Applications Conference, COMPSAC 1995, 1995, 96–101.
[28] C. Manning, H. Schütze, Foundations of Statistical Natural Language Processing, The MIT Press, Cambridge, MA, 1999.
[29] D. McClosky, E. Charniak, M. Johnson, Effective self-training for parsing, In: HLT-NAACL 06 Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, 2006.
[30] R.A. Niemeijer, Constraint Specification in Architecture: A User-Oriented Approach for Mass Customization, PhD Thesis, Eindhoven University of Technology, 2011.
[31] S. Nanduri, S. Rugaber, Requirements validation via automated natural language parsing. In: Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences, 3, 1995, 362–368.

[32] P. Pauwels, D. Van Deursen, R. Verstraeten, J. De Roo, R. De Meyer, R. Van de Walle, J. Van Campenhout, A semantic rule checking environment for building performance checking, Automat. Constr. 20 (2011) 506–518.
[33] W. Rasdorf, S. Lakmazaheri, Logic-based approach for modeling organization of design standards, J. Comput. Civ. Eng. 4 (1990) 102–123.
[30] Michael A. Rosenman, John S. Gero, Design codes as expert systems, Comput.-Aided Des. 17 (9) (1985) 399–409, http://dx.doi.org/10.1016/0010-4485(85)90287-8.
[35] H. Ridder, R. Vrijhoef, The "living building" concept: dynamic control of whole life value and costs of built services, in: Proceedings QUT Research Week, 2005, pp. 109–115.
[36] C. van den Thillart, Customised industrialisation in the residential sector: mass customisation modelling as a tool for benchmarking, variation and selection, Sun, Amsterdam, 2004.
[37] E. Tsang, Foundations of Constraint Satisfaction, Academic Press, 1993.
[38] B. de Vries, A. Jessurun, R. Kelleners, Using 3D geometric constraints in architectural design support systems, in: WSCG 2000 Conference, Bory, Czech Republic 'Proceedings of the 8-th International Conference in Central Europa on Computer Graphics, Visualization and Interactive Digital Media', 2000.
[39] A. Yurchyshyna, A. Zarli, An ontology-based approach for formalisation and semantic organisation of conformance requirements in construction, Automat. Constr. 18 (2009) 1084–1098.
[40] Q.Z. Yang, X. Xu, Design knowledge modeling and software implementation for building code compliance checking, Build. Environ. 39 (2004) 689–698.
[41] B.T. Zhong, L.Y. Ding, H.B. Luo, Y. Zhou, Y.Z. Hu, H.M. Hu, Ontology-based semantic modeling of regulation constraint for automated construction quality compliance checking, Automat. Constr. 28 (2012) 58–70.