Constraint specification in architecture: a user-oriented approach for mass customization
Niemeijer, R.A.

DOI:
10.6100/IR715226

Published: 01/01/2011

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.

Link to publication

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: 17. Mar. 2018
Constraint specification in architecture: A user-oriented approach for mass customization

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus, prof. dr. ir. C.J. van Duijn, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op dinsdag 28 juni 2011 om 16.00 uur

door

Remco Arnout Niemeijer

geboren te Dordrecht
Dit proefschrift is goedgekeurd door de promotor:

prof.dr.ir. B. de Vries

Copromotor:
dr.Dipl.-Ing. J. Beetz
Preface

This thesis marks the end of a four-year research project at the Design Systems group at the Eindhoven University of Technology (TU/e), which examines the use of constraints in architectural design, particularly with the goal of facilitating mass customization. This PhD project came as somewhat of a surprise, since when I submitted the proposal for my Master thesis I had expected to get a job afterwards. On the subsequent meeting, however, my supervisor, Bauke de Vries, presented me with a proposal for a PhD project that had the same topic as my planned Master thesis (though naturally with a larger scope). I had chosen the topic because the concept of mass customization had interested me for a while, and had featured in two earlier projects I did during my Master studies. In the first, I created a website for configuring a multiple-choice house. The second, which was done as an internship at the architecture firm BBVH, was conceptually the same, but, instead of having to decide based on 2D pictures on a website, clients were able to walk through a full 3D model of the house while making changes along the way. The idea of mass customization interests me because the housing industry is one of only a few remaining consumer industries that fails to offer much in the way of choice. When buying most any other consumer product, be it a carton of milk or a stereo system, competition and product diversification have resulted in a wide range of products to choose from. In the housing industry, however, the lack of competition and a fairly traditional way of working mean that there is often little, if any, choice for a given location, the only alternative being to find a different housing project, assuming there is one. It is my belief that even in the housing industry it would be possible to give customers the freedom of choice they are used to from other industries and I hope this research will be a small step towards that goal.
Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Bauke de Vries. His input was invaluable, both in terms of planning the research and in writing articles and of course for this thesis. I also want to thank Jakob Beetz for his feedback on my writing. I want to extend my thanks to the other members of my committee — Loe Feijs, Imre Horvath and Erik Jansen — for their feedback on my work.

My thanks go out to all of my colleagues at the Design Systems group, but three of them I want to highlight in particular: Joran Jessurun, for his insight during the times when I was stuck on a coding problem and for helping me become a better programmer, but also for him introducing me to the world of board games; Sjoerd Buma, for his technical support, and Marlyn Aretz for helping me navigate all of the administrative hurdles that occur in a PhD project.

Lastly, but perhaps most importantly, I want to thank my parents for their love and support.
CHAPTER 1

Introduction

Even a journey of a thousand miles starts with a single step

Lao Tzu
1. Introduction

The industrialized method of building, which has been in use since World War II for the construction of residential areas, does not leave much room for individual choice. Economies of scale dictate that all houses are identical, resulting in urban monotony and houses that do not fit their occupants as well as they might. This lack of choice is common when first introducing mass production in an industry (as shown by Henry Ford’s famous quote: “Any customer can have a car painted any colour that he wants — so long as it is black.”). However, other industries have shown that mass production can be combined with individual choice. This approach is called mass customization (van den Thillart 2004, Huang and Krawczyk 2007). Examples include the car industry and computer manufacturer Dell. By making the production process more flexible, designs that are tailored to individuals can still be produced industrially. When trying to introduce mass customization to the building industry, a difference with other industries becomes apparent: unlike a car or a computer, a house does not have a standardized design. The approach of choosing the components for the predefined design (what type of seats? which colour paint?) is therefore insufficient. This approach is already being used in practice in some projects, but it only allows limited flexibility. To create houses that are truly adapted to their buyers, a different approach is needed.

1.1 Motivation

Currently, people buying a new house typically have one of two options: either the design is made by the architect and no modifications can be made, or they can choose from a brochure with a limited number of alternatives, such as two different kitchen types or the optional addition of a dormer. This customization is only very limited though, as all design alternatives offered by the architect have to be fully designed up front. This means that people are not able to get the exact house they want, meaning that it is not uncommon for people to immediately start remodelling after the house has been built to get the house they actually wanted.
This of course 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 house in the design phase, so that they can get the house they want, eliminating the need for an additional remodelling step. This same basic philosophy is advocated by Hennes de Ridder in his “Living Building” concept (de Ridder and Vrijhoef 2005). This, however, leads back to the problem of the architect having to make a lot of different designs, which is costly and time-consuming, as it requires sitting down with every family and making a customized house for each of them. In order for this philosophy to be feasible in large-scale projects, a different method must be used.

In this thesis, an approach is presented where buyers are free to make modifications to the design by themselves. This way, all design modifications can be done concurrently rather than sequentially. This greatly speeds up the process. The fact that the modifications can be made independently frees up the architect’s time as well. Naturally, this approach introduces a number of challenges. Since buyers normally have no architectural experience the program will have to support them, offering suggestions and preventing them from making design mistakes. Additionally, the architect will naturally want to retain a certain amount of control over the resulting designs to ensure they still match his vision.

In order to achieve this, the program will have to reason about the design. Several types of reasoning exist. One of the more straightforward types of reasoning is rules-based reasoning or constraint-based reasoning. Here, designs are evaluated for compliance with a series of rules. An alternative method is to determine design validity by comparing it against known previous designs, in a process known as case-based reasoning. Both of these approaches are deterministic — a design will be either valid or invalid. It is also possible to use non-deterministic reasoning. Probabilistic reasoning indicates the chance that a design is valid rather than trying to give a single verdict. Fuzzy reasoning or vague reasoning is another non-deterministic type of reasoning. At first glance similar to probabilistic reasoning, fuzzy reasoning indicates to what extent an element is a member of a set rather than a possibility of something happening.

Each of these types of reasoning has its application domains. Case-based reasoning is, among others, used heavily in the practice of law. It is somewhat less suited for use in the building industry since each project has a unique location, different requirements and often a greatly differing look, which makes finding similar cases difficult. Probabilistic reasoning is used extensively in the financial domain, such as in stock trading algorithms where shares are bought and sold based on the chance that certain events will happen. Constraints in the building industry,
however — at least those found in legislation and building codes — are rarely expressed in terms of probabilities. Wood, for instance, is expected to meet a certain fire safety classification; the probability of the fire reaching the element is not taken into account. The same holds true for fuzzy reasoning. There is typically little, if any, doubt as to whether, for example, a wall is a load-bearing wall or not. Though certain aspects of building regulation would qualify for the use of fuzzy reasoning, such as building climate-related demands, the actual rules are generally expressed in more well-defined terms. This leaves constraint-based reasoning as the most suitable candidate for testing for problems and conformity with design intent.

In this new method of housing design, the architect would create a base design, just as he does currently (note: in this thesis, people will be referred to as he, regardless of gender). In addition to that, however, he also specifies a series of constraints indicating what he will and will not allow people to change. The base design and the associated constraints are then presented to the buyers through a simplified CAD interface, in which they can make the modifications they want. At every step, the system will check whether the new design still meets all of the constraints (which are not limited to the ones defined by the architect, but also include constraints such as building codes, energy performance guidelines, constructive engineering rules and ‘common sense’ constraints).

The success of such a system is contingent on the ability of buyers to make valid designs and on whether they feel the system affords them enough flexibility. Important to note is that the constraints are not used for constraint solving, which is the automated search for a design that satisfies all of the given constraints. Instead, the constraints are only used to verify human-created designs, a process which is called constraint checking. The goal is not to create an optimal design since it is arguable whether an optimal house even exists given the amount of (often conflicting) criteria, which is illustrated by the fact that centuries of architecture have yet to produce an optimal design.

Creating a constraint-based architectural design system introduces many technical challenges, including the creation of a user interface that is usable by non-experts, the development of a Building Information Model that is detailed enough to contain all the building elements referred to by the constraints and the changes that must be made to the production chain to support mass customization. This research focuses on the entry of the constraints into the system. Since architects typically have little to no experience with computer programming, the challenge is to make the constraint entry process as simple and familiar to them as possible while ensuring the computer can still understand the constraints that are entered.
**Methodology**

The study of the proposed approach of allowing buyers to modify their house through *constraint-based design* is conducted as design inclusive research (Horváth 2007). This methodology is the one most suited to this project, since constraints are currently used only sporadically in architecture, making practice-based design research difficult. A case could be made for research in a design context, but the nature of the research is more constructive than analytical, making design inclusive research the better choice. The one modification that was made to the typical process for this methodology is that the phase of creative design actions is split up into three rounds. Rather than the waterfall method of software development (Royce 1970) in which an initial phase of requirements gathering is followed by a long period of development resulting in a final product, three separate programs are developed in a process known as *rapid prototyping* (Somerville 2001). Rapid prototyping consists of multiple shorter cycles of analysis and development. Because of this, user feedback is available earlier in the development cycle, meaning that it is easier to change direction than in the traditional way of working. The three prototypes created in this project are outlined in figure 1.1.

![Prototype overview](image)

The goal of the first prototype is to prove the hypothesis that using *constraint checking* to practice constraint-based design is technically viable. Constraints are placed on a small building design to see whether it is possible to modify the design while satisfying the constraints. The second prototype explores the use of *visual programming* for constraint entry. It tests the hypothesis that a constraint entry method based on assembling sentences via blocks containing sentence fragments
can retain the familiarity of natural language while avoiding the difficulties that arise from natural language parsing. The prototype is tested on architects to determine whether this method of constraint entry is suitable for use in practice. The third prototype, finally, tests the hypothesis that constraints specified in a subset of natural language can be interpreted using a relative simple algorithm. To test this algorithm, prototype based on natural language processing was developed and evaluated by assembling a test suite of natural language constraints from both building codes and architecture students to determine the success rate of the algorithm.

1.3 Research questions

The ultimate goal of this research is to explore and aggregate knowledge for a system that can intelligently support non-designers in the process of creating and modifying architectural designs. This goal provokes a large variety of research questions, ranging from ‘what is the best way of presenting an architectural design to a non-designer?’ and ‘to what extent do current construction methods have to be adapted to support mass customization?’ to ‘what are the current limitations of building information models in regards to automated design validation?’.

This research project focuses on the validation of client designs against the constraints that apply to buildings, such as zoning laws, building codes and the vision of the architect. Specifically, the goal is to define a specification entry method for building components that can be applied by architects and suppliers in customizable building designs. This advances the current state of the art of constraint use in architecture by providing a far more flexible method of specifying constraints than the current methods, which are typically limited to simple geometrical constraints. The research questions addressed by this PhD project are:

▷ Is it possible to develop a method of specifying all geometrical and material specifications for a building component within a set of constraints that are determined by its functional or technical properties?

▷ Can the constraints be specified in such a way that little or no training is required on the part of the architect to start using it?

Important to note is that this thesis does not claim to address the problem of the infiniteness (de Boer et al. 1999), vagueness (Fine 1975, Kyburg and Morreau 2000) or ill-structured nature (Simon 1973) of some constraints. The algorithm that is developed deals only with constraints that are objective and well-structured. The problem of infiniteness is avoided by performing constraint checking rather than constraint solving, which means that instead of trying to search the entire solution
space in order to generate a valid design, an existing design is evaluated for compliance with the constraints. An algorithm is presented that determines semantic dependency of natural language constraints based on word order. The algorithm combines robustness with a relatively simple implementation.

1.4 Outcomes
The focus on constraint entry in this thesis presumes that constraint checking in the building industry is feasible. This was expected to be the case, since similar techniques have been used in other industries for many years. The constraint checking prototype demonstrated that the same holds true in the building industry, freeing the way to look at the constraint checking process in more depth. The results of the second prototype revealed that visual programming, although it initially appeared to be a good idea for constraint entry, is too laborious to be used in practice. It was therefore decided to change direction and switch to a constraint method based on natural language, which despite it being an early prototype produces interesting results, with 59% of the constraints already being interpreted correctly by the system with little or no modification. Using natural language therefore appears to be a promising way of handling constraint entry for constraint-based design.

1.5 Thesis overview
Chapter 2 discusses the use of mass customization in both the building industry and other industries and shows that there are still many opportunities for increased adoption of mass customization in the building industry. Chapter 3 investigates how constraints are used in various industries, and explains the way constraints will be used in this research. Chapter 4 describes two of the three developed prototypes, testing the viability of constraint-driven design in the building industry and one method of constraint entry. Chapter 5 covers the final constraint entry method based on natural language input, as well as the theory behind natural language processing. Chapter 6 describes the test setup used to evaluate the natural language prototype from the previous chapter and presents the results. Chapter 7, finally, summarises the project and discusses future research.
You can have the Model T in any color you want—as long as it’s black.

Henry Ford
Before the invention of computer-aided design, most manufacturing processes were limited to one of two options: mass production or customization. Mass production is the production of large amounts of identical parts. The idea has existed for hundreds of years (the Venetian Arsenal, a shipyard in Venice, employed mass production to produce one ship per day as far back as the 14th or 15th century), but it did not achieve widespread popularity until its adoption by Henry Ford’s Ford Motor Company. Due to economies of scale, mass production reduces costs significantly, but it prohibits individual choice. This is illustrated by a well-known quote from Henry Ford about the Model T Ford, of which an abbreviated version can be found on the title page of this chapter. The full version of the quote is “Any customer can have a car painted any colour that he wants, so long as it is black” (Ford and Crowther 1922). On the other end of the spectrum is customization, where each product is unique and built according to a different specification. A textbook example is the art of portrait painting, where each portrait is necessarily unique. Since the lack of repetition reduces the possibilities for standardization, costs are typically higher than for mass production. Important to note is that mass production and customization can exist within the same industry and even within the same project. In the housing industry, for instance, housing projects are usually executed as mass production while one-of-a-kind buildings such as museums are customized, although they are often constructed out of mass-produced components.

Mass customization (van den Thillart 2004, Shin et al. 2008, Benros and Duarte 2009) is a mix between the two systems that attempts to combine the low cost of mass production with the flexibility of custom work. Although not inherently required, mass customization is often achieved through computer aided design, which allows for more flexible output with little or no additional variable costs (though at the cost of a higher up-front investment). The car industry is a good
example of this approach. Instead of merely producing one car model as in the
days of the Model T Ford, customers can now choose from a wide range of models
in different colours and with different extras, which are all assembled on the same
assembly line. In this chapter, some of the ways that mass customization is used in
various industries are discussed. Building Information Models are also covered,
due to their use in mass customization in the building industry. Finally, a method
for adopting mass customization in the housing industry is proposed, which is
explored in more detail in the following chapter.

2.1 The different types of mass customization
There are many ways to subdivide mass customization into different types. This
section contains several of the more commonly cited categorizations in mass cus-
tomization research (Duray et al. 2000, Nambiar 2009)

2.1.1 Standardization and marketing
In the seminal book “Mass Customization: The New Frontier in Business Com-
petition” (Pine II 1993), four different types of mass customization are identified:
collaborative, adaptive, transparent and cosmetic customization. These four types
can be put in a two-dimensional matrix based on the amount of standardization
and the extent to which the product is marketed as customized; see table 2.1.

<table>
<thead>
<tr>
<th></th>
<th>Not marketed</th>
<th>Marketed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customized</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standardized</td>
<td>Transparent</td>
<td>Collaborative</td>
</tr>
<tr>
<td></td>
<td>Adaptive</td>
<td>Cosmetic</td>
</tr>
</tbody>
</table>

Table 2.1 Customization types according to Pine II

In the case of adaptive customization, the manufacturer’s product is standardized,
but can be modified by the customers. An example would be a typical office chair,
which can be adjusted to accommodate people of varying sizes by adjusting a few
levers. There is a high degree of standardization and the product is not marketed
as being customizable. If a standardized product is marketed as being customized
to a certain target audience, this is called cosmetic customization. For example,
SUVs (Sport Utility Vehicles) are marketed to people in more rural areas for their
off-road capabilities and to people in the suburbs for their safety. The opposite of
cosmetic customization is transparent customization. Here, a customized prod-
uct is marketed as standard. This is prevalent in the food industry. Fanta Orange,
for example, has a different flavour and colour in different countries, but has the
same name everywhere. In collaborative customization, there is a dialog between
the producer and the consumer to determine the consumer’s exact needs. This
information is then used to manufacture a product specifically for that customer. The automobile industry operates on this basis; the customer chooses a model, a colour, etc. and this information is sent to the factory, where the chosen car is produced. Here, the product is customized and also marketed as such; In Germany, Volkswagen offers a service where you can order your car and drive to the manufacturing plant to see it assembled (Volkswagen 2011). It is interesting to consider the adoption of a similar methodology in the building industry, as there appear to be no real reasons why the same could not be achieved with buildings. Projects where buildings can be customized are not uncommon and the construction of a 15-storey hotel in just 6 days in China in 2010 (Yahoo! News 2010) demonstrates that rapid construction is possible as well. What remains is to combine the two. The result would be rapidly constructed, customizable housing, which is a significant improvement over current common practice.

2.1.2 Involvement of the client in the production process

Another subdivision is given in “Customizing Customization” (Lampel and Mintzberg, 1996), where five types of mass customization are identified: pure standardization, segmented standardization, customized standardization, tailored customization and pure customization. These five different types are discriminated based on the moment at which the customer enters the production chain.

In pure standardization, the client does not enter into the production process. The only flexibility provided is that which is inherent in the product, such as the adjustability of car seats. In segmented standardization, the client enters the production process in the distribution phase, giving them control over the delivery schedule, for instance. In customized standardization, the client is brought in one step earlier, in the assembly phase. In this approach, the customer can assemble the product out of standardized components. This is known as modular customization (Starr 1965, Duray 2000, Halman 2008).

Modular customization can be applied to a single product, but it is also possible to develop an entire product family in which all products share the same components. This is known as platform-based customization: (Du et al. 2001, Simpson 2004). Well-known examples of modular customization are IKEA’s book cases and other storage solutions and Subway, where customers can assemble sandwiches from a fixed list of ingredients. Tailored customization involves the client from the fabrication phase onwards. An obvious example is a tailor that adapts the article of clothing to fit the customer. In pure customization, finally, the client is involved starting from the earliest phase, in which the product is designed, a good example being custom software development, where the application is written according to the client’s specification.
2.1.3 **Type of modularity**

If modular customization is used, six types of modularity can be identified (Duray et al. 2000). In component-sharing modularity, different products share the same component, such as different cars sharing the same chassis. Component-swapping modularity means that parts of a product can be replaced with other ones, such as in customized computers where elements such as the CPU or hard drive can be replaced with other models. In cut-to-fit modularity the dimensions of a component can be changed to a certain extent, provided the dimensions of the interface to the rest of the product remain unchanged. Eyeglasses are a well-known example. Mix modularity is similar to component swapping, save for the fact that once combined the components lose their unique identity, as is the case in house paints, for example. Bus modularity is the ability to add a module to an existing series, when one or more modules are added to an existing base. An example of this type of modularity is track lighting. In section modularity, finally, products are created by arranging standard components in unique configurations, such as in LEGO models.

2.1.4 **Consumer influence on the design**

Alford et al. distinguish three types of mass customization based on the extent to which customers can influence the design, namely core, optional and form customization (Alford et al. 2000). Their examples are all drawn from the automotive industry. Core customization occurs when customers have a large amount of influence over the design, mainly occurring in the low-volume specialist vehicle market. Optional customization is achieved by being able to choose from a wide range of options. The sedan market segment, with its large number of brands and several models per brand, is a good example of this. Form customization covers the small modifications that are offered by dealers and retailers, such as satellite navigation systems or hub caps.

2.1.5 **Influence, production flexibility and repeatability**

In the paper “Fundamental modes of operation for mass customization” (MacCarthy et al., 2003), three dimensions are identified for disambiguating different mass customization types: influence, flexibility of the production process and repeatability. Not all of these are viable combinations, however, which leaves the following five: catalogue mass customization, fixed and flexible resource design-per-order mass customization, and fixed and flexible resource call-off mass customization. In catalogue customization, products are manufactured regardless of demand and customers choose from a predefined range of products. Many consumer products, ranging from cars to MP3 players to food, fall in this category. Both fixed and flexible resource design-per-order mass customization (DPOMC) deal with one-time purchases, the difference being that in fixed resource DPOMC
the customer needs to take the limitations of the manufacturing process into account, as this is standardized. In flexible resource DPOMC, this requirement is removed as the manufacturing process can be adapted as needed. Computer manufacturer Dell provides an example of fixed resource DPOMC; the assembly process does not change and orders are not expected to be repeated. Fixed and flexible resource call-off mass customization are largely the same as their DPOMC equivalents, save for the fact that orders are expected to be repeated. Corporate stationery is an example of fixed resource call-off mass customization.

2.1.6 Conclusions

In this section, a number of different types of mass customization are presented. The question, however, is to what extent these types can be used to provide more flexibility in the housing industry. In the categorization by Pine II, only collaborative customization provides the required flexibility. Transparent and cosmetic customization can be seen as the current method of working: the architect tries to tailor the design to the average member of the expected target audience or the standardized houses are marketed to one or more target audiences. Adaptive customization is more problematic in architecture, since the scale of buildings means that they are usually difficult to modify in any significant way. Pure and segmented standardization, as identified by Lampel and Mintzberg do not offer the required amount of flexibility and can therefore be dismissed. The remaining three are all theoretically possible, but from a practical standpoint customized standardization is likely to be the better choice, since it matches the current practice of composing buildings out of standardized components well. Tailored and pure customization will likely prove to be too cost-ineffective. They also pose a problem in terms of uniformity, since allowing buyers to influence the sizes, materials, etc. of all the building elements will make it very difficult to maintain any semblance of coherence in the design.

Of the different types of modular customization identified by Duray et al., component-sharing modularity is to be expected, if not particularly important from the viewpoint of the buyer. A row of houses, for example, will in all likelihood share the same floor elements. Component-swapping and bus modularity are of limited use, since the position of a door, for instance, will often be more important than the wood used for the doorframe. Cut-to-fit modularity is less relevant in the housing industry due to the more or less standardized dimensions of many components, such as doors and kitchen cabinets. The two most suitable types, then, are section and mix modularity. The similarity between building a LEGO model and a house is easy to see, and mix modularity covers finishes such as stucco and paint. The appropriate level of consumer influence is more difficult to determine. It probably lies somewhere along the spectrum between core and optional customization, but
further study is required to determine the best trade-off between flexibility and the required effort. In the categorization of MacCarthy et al., finally, catalogue mass customization represents the current state of mass customization in the housing industry, which does not offer a sufficient amount of choice. Both types of call-off customization are irrelevant since a house is a one-time purchase, leaving the two design-per-order types, with fixed resource DPOMC being the more likely of the two options. In summary, the preferred method of using mass customization in the housing industry is to give the buyer the ability to assemble his house out of pre-built components in the design phase of the project. This should occur in collaboration with the architect.

2.2 **Mass customization in the housing industry**

Up until the 19th century, houses were typically built individually, and were customized to the owner. This was of course not cheap, and hence proper houses were reserved for richer citizens. Less affluent people often lived in shacks they built themselves. In the Netherlands this started to change at the start of the 20th century when increasing urbanization, industrialization and the Woningwet (Housing law) of 1901 started a trend towards social housing. Social housing, like the industrialization that led to it, is based on mass production. A large amount of identical – and thus cheap – houses were constructed for the lower classes. After the Second World War, the required rebuilding effort and the baby boom resulted in an even greater push towards social housing. The late 20th and early 21st century, however, saw the beginning of a movement away from mass-produced identical houses towards individually customized homes. This is largely due to increasing levels of wealth, which provide both the means and desire for customized housing.
2.2.1 Dom-ino house
Between 1914 and 1915, architect Le Corbusier designed the “Dom-ino” house (von Moos 1982), shown in figure 2.2. Although it was never realized, it was one of the first dwelling designs that could be mass-produced while retaining a large degree of flexibility. Rather than a completed design, the Dom-ino house is a framework upon which designs can be based. The Dom-ino house concept consists of concrete floor slabs supported by pillars, with concrete stairs connecting the floors. Wall materials, floor plan layouts and duct and pipe locations can be modified on a per-house basis, making the Dom-ino house a good basis for mass customization.

2.2.2 Rietveld-Schröderhuis
Designed in 1924 by Dutch architect Gerrit Rietveld in cooperation with client Truus Schröder, the Rietveld-Schröder house (shown in figure 2.3) is noteworthy because it has no fixed interior walls. Instead, both floors are big open spaces that contain flexible dividers that can be used to delineate spaces. Because of this, occupants are completely free to define their own floor plan. This can be seen as an example of adaptive customization, where the product itself is standardized, and flexibility is provided through user-adjustable controls.

2.2.3 Habraken’s alternative to mass housing
In 1964, the BNA (Bond van Nederlandse Architecten, i.e. the Union of Dutch Architects) presented a young architect by the name of John Habraken with the opportunity to start a research foundation. Two years prior, Habraken had written a book called “Supports, an alternative to mass housing” (Habraken et al. 1999). In this book, he claimed that the elimination of the buyer that occurred in mass-produced housing was incorrect. Instead, Habraken argued for dividing a house in two different spheres, which he called “support” and “infill”. The infill sphere refers to parts of the house that are likely to change between different owners, such as kitchens, the floor plan and floor material. The support sphere contains the more permanent features of the house (load-bearing walls, ducts, etc.). A project would have only a small number of different support spheres, but the infill sphere was unique to each dwelling. The support sphere therefore has to be flexible enough in its layout to accommodate many different infills. Buyers could view the different infills in showrooms, and make a selection with guidance from a specialist.

The research foundation that John Habraken founded was named SAR (Stichting Architecten Research, i.e. Foundation for Architect’s Research). This foundation further developed and promoted Habraken’s “Open bouwen” (open building) method, and this marked the first step towards mass customization in the Netherlands. Although Habraken’s method is a significant increase in freedom of choice for the buyer, there are a few limitations. The potential for more involved
changes, such as adding an extra floor, is limited, since the construction method for the supports is based on mass production. Additionally, the need for buyers to confer with a specialist when choosing the infill means that this method becomes labour-intensive on larger projects.

2.2.4 IFD building

In the early 1990s, a design philosophy called “Industrieel, Flexibel en Demontabel bouwen” (Industrial, Flexible and Demountable building) was introduced (van den Boogaard 1990, van Gassel 2003). It has the goal of making buildings more sustainable, which it accomplishes in three different ways. The “Industrial” part of the name refers to the fact that buildings should be produced in such a way that they can be mass produced, resulting in more efficient production that saves resources and economies of scale that lower building costs. Flexibility is achieved in two ways. Allowing the initial client to modify the design removes the need for them to remodel. Additionally, it should be easy to adapt the building to the needs of subsequent clients. If a building cannot be retrofitted in this way, it should be possible to disassemble the building in its constituent components without generating a lot of waste, reducing the impact on the environment. Although IFD was not developed with mass customization in mind, it does share many of the same goals and the adoption of IFD facilitates mass customization.
### 2.2.5 Multiple choice housing

Participatory design (Sanoff 1990, Schuler and Namioka 1993), i.e. the act of involving the user in the design process, can be conducted in one of two ways: co-located or dislocated. The traditional approach to customizing houses requires that the architect and the buyer meet in person to discuss the desired changes, which falls under co-located participatory design. A method of dislocated participatory design that has been growing in popularity in the past decade or two is to present buyers with a limited number of variants for different parts of the house (Hofman et al. 2006), resulting in a sort of “multiple choice” style of participatory design. This increased flexibility usually takes the form of brochures in which people can choose between different alternatives, e.g. an optional extra storey. The brochure can take the form of a traditional paper brochure, or of a computer application that allows you to interactively modify the design on a computer. Examples of the latter approach include the “Woonwijzer” project by architecture firm BBVH (BBVH 2011) and TNO’s iBuild (TNO 2010). This approach has the advantage of eliminating the discussion, making it much less time-intensive on the part of the architect at the cost of limiting the amount of input the client can have on the design.

This “multiple choice” approach to architecture requires that the architect designs all the alternatives up front. Due to the combinatorial growth of these alternatives — five choices of three alternatives each already result in $3^5 = 243$ total variations — the amount of options is usually kept fairly limited. This means that while a number of different alternatives are offered, there is still a good chance that a customer’s desired design variation is not offered. Additionally, there is the risk that some of the designed alternatives will not be used at all, since they are created by the architect instead of by the users.

### 2.2.6 User-driven design

A different approach to dislocated participatory design that eliminates the problem of having to create many design alternatives is to have the clients make the design themselves by offering modular mass customization. A well-known adopter of this strategy is IKEA, who offer tools for designing kitchens, storage solutions and offices, among others (IKEA 2011). An example of using this philosophy to design an entire house is the “Wenswonen” project by Heijmans (Heijmans 2011). The advantage of this method is that clients are given even more freedom than in multiple choice housing. The main drawback stems from the fact that clients are not professional designers. As such, they are prone to make suboptimal — or outright illegal — designs. This means that the client designs first have to be checked by an expert for viability. An alternate method to employ user-driven design is to use it to gauge consumer preference, by seeing which modifications clients frequently make to the design (Orzechowski 2004).
Building information models in mass customization

As covered in the previous section, having people choose from a set of alternatives for different parts of the house quickly becomes impractical due to the combinatorial explosion of the amount of total variants that have to be designed. The traditional alternative — a one-on-one meeting with the architect to design the house — suffers from the problem of not scaling well to larger projects, as the amount of time required to have a design session with all of the clients becomes prohibitive. A third alternative, which will be explored in this thesis, is to allow people to modify the design by themselves. This way, architects don’t have to create variants themselves, saving a lot of time, and buyers are able to design their house exactly the way they want. Allowing non-experts to design a house, however, introduces a new problem; being unaware of building regulations, they are likely to create illegal designs. Therefore, all the buyers’ designs will have to be checked to see if they comply with both building regulations and the architect’s vision. For this reason blueprints — the medium typically used to express designs — are not well suited to this task, as they are very labour-intensive to check for errors. The fact that the clients have little to no experience creating blueprints only exacerbates this. Ideally, a large part of the design verification could be done by a computer.

Although a computer cannot assess the aesthetical quality or the practicality of a design, many of the building regulations prescribe criteria that can be easily and objectively measured, such as the maximum height of a building element. In order to be able to achieve this, the design needs to be represented in a way that the computer can understand, meaning that the elements and concepts referred to in the constraints can be easily determined from the design representation. For example, it would be difficult for a computer to check the criterion “Windows should be at least 1 m high” for a blueprint, since it would first have to determine which lines form the windows, which is far from trivial.

One way of making the required information easily available to the computer is to represent the design as a collection of building elements, with information stored in the properties of the elements. This methodology is called parametric design (Roller 1991, Matcha 2007). It is a key part of Building Information Modeling (BIM), a term first coined by architect and Autodesk employee Jerry Laiserin in 2002 (Laiserin 2002), although the same concept was introduced under the name “Building Product Model” by Charles M. Eastman in the late 1970s (Eastman 1999). Aside from the properties of individual elements, BIMs may also store information that is not restricted to a single element, such as relationships between elements, and non-geometrical information, such as climate information. In this thesis, the term BIM is used to refer to all types of parametric models.
In a BIM, architectural models are stored as a collection of building elements rather than a series of lines, as was customary previously, a tradition inherited from the days of the drafting table (Ibrahim et al. 2004). This means that, for example, a window is not merely a group of rectangles, but a single element with properties such as width and height, making it significantly easier for the computer to reason about the design. BIMs have many advantages over the classical line-based method of representation. They offer increased convenience in creation and modification, since architects can work on the element level rather than having to manipulate individual lines and because there is only one model, instead of every plan and section being a completely unrelated drawing. BIMs also improve communication between tools (Kouider et al. 2007), because it decouples the model from its representation. Consider as an example to CAD packages, one 2D and one 3D. Without the use of a BIM, collaboration would be very difficult, since once the design is saved in the 2D packages, all information would be saved as a 2D collection of lines, destroying a lot of the information in the design. Using a BIM, the 2D package can display the model in 2D while still maintaining the building elements when saving. Because of these — and more — advantages, the majority of CAD applications now store designs in BIMs. Examples include Revit, Microstation and ArchiCAD, which was the first CAD package to introduce the concept of a BIM.

A final advantage is that BIMs open up the possibility of automated design verification. Because the design contains actual building elements rather than a series of lines that can only be understood by humans, it becomes possible to automatically check whether those elements meet certain criteria. This automated design verification is the focus of this thesis. In subsequent chapters, methods for performing the verification will be discussed before focusing on how these criteria can be formalized in a way that architects will be able to define their own criteria for building designs. The aim of this is to support mass customization, since the architect will be able to ensure that designs produced by the buyers comply with his architectural vision, as well as the other requirements that are imposed on houses, such as building codes.

2.3.1 IFC
One of the advantages of BIMs mentioned earlier was the improved communication between tools. This is true, provided that both tools use the same BIM. Unfortunately, tools from different software firms tend to use their own proprietary BIMs, each with its own pros and cons. Intercommunication between different products of the same vendor tends to work well, but communication across vendors is frequently problematic, since the BIMs are not directly compatible. Although no worse than the situation before the introduction of BIMs, when different packages could also not open one another’s file formats, it hurts productivity due to
the resulting communication barriers. In order to solve this problem, the International Alliance for Interoperability (IAI 2010) developed a vendor-independent specification for a building model called IFC (Industry Foundation Classes). The intent is to provide an open standard for BIMs that all CAD vendors can read and write to facilitate interoperability between the various CAD programs. IFC is an ISO-certified standard to describe a BIM. This is one of the few data exchange standards that the building industry has that does not only describe geometry, and a good number of CAD packages are compatible with it. Thus, a system that is compatible with IFC should automatically work with all major CAD applications, without having to account for all their different internal BIMs. Another advantage of the IFC standard is that it is an open standard, as opposed to the mostly closed standards of the CAD software manufacturers, making it easier to implement. In the prototypes discussed later in this thesis, IFC is used to import CAD models.

2.4 Conclusions

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. Several explanations can be given for this, such as the tradition of the architect being the sole designer to the fact that buildings are subject to far more regulations than many consumer products. These obstacles are not insurmountable; at various points in the past century there have been experiments in which clients were given more design freedom. The past few decades in particular have seen an ever-increasing adoption of mass customization. In most cases, though, the amount of flexibility is still limited, since the two traditional ways of offering mass customized housing — a fully custom design or choosing from predesigned alternatives — result in a trade-off between the freedom offered and the amount of time required. This could be partially solved by a better use of Building Information Models, but a full solution will require a different approach entirely.

When making a design for a consumer product, there are many rules that must be obeyed (Halman et al. 2008). Some derive from human morphology; a phone must be small enough to fit in your hand. Some are marketing-based, such as a maximum cost requirement. Yet another source of design rules are laws and regulations, for example the safety requirements on cars. All these rules are constraints that the final design must satisfy. Currently, checking whether all of these rules have been satisfied is, in most cases, done manually. Due to the large amount of rules this is very labour-intensive. Finding a way to automate this checking process would greatly benefit this phase. This requires that building regulations can be formalized in an objective manner so that they can be verified by a computer. Although a large subclass of all building regulations can indeed be formalized,
there are exceptions. Regulations such as “the architectural quality of the addition must match that of the surrounding buildings” have no objective interpretation, as “architectural quality” is an ill-defined term: does it refer to technical quality? Aesthetics? There is no single accepted definition and thus the rule cannot be formalized. As such, a system that performs automated regulation checking will not be able to handle every regulation that is currently found in practice. It will, however, be able to check a sizeable amount, if not the majority, of the regulations, removing the need for people to worry about the trivially checked rules and giving them more time to focus on questions of aesthetics and such. From here on out, design rules that can be formalized will be referred to as constraints (de Vries et al. 2000). The word constraint has many different definitions in various fields. In this thesis, the definition given in “Foundations of Constraint Satisfaction” (Tsang 1993) is used:

“...a CSP [Constraint Satisfaction Problem] is 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.”

Constraint satisfaction (Dohmen 1995) is the process of arriving at a design solution that satisfies all of the constraints. There are two ways of finding such a solution; the first is to keep them in mind while designing and afterwards checking to see if you satisfied them all. This method is the simplest one from a technological standpoint, but the risk of overlooking one or more rules and thus having to create another design iteration — or worse, forgetting a constraint — is significant. In addition, checking the design for constraints compliance is a laborious process, as each constraint has to be checked manually and individually, and the process has to be repeated for every design iteration. A large majority of all building design processes take this approach. The alternative is to explicitly represent the constraints in the design and to check them continually while designing. This ensures that any design will satisfy the constraints placed upon it. This is considerably more complicated to implement technically, but it is the option that will be explored in subsequent chapters. The issue of decomposition and generalization of constraints is not addressed in this thesis, although the need to support the decomposition of constraints to express certain types of constraints is addressed in chapter 6.

Three types of constraints can be identified: quantitative constraints (e.g. the height of the wall must be less than 3 m), qualitative constraints (e.g. windows cannot overlap) and hybrid constraints that combine elements of both. In this thesis all three types of constraints are used, though quantitative constraints are the main focus. As mentioned, not every building regulation can be formalized as a
constraint. This does not mean that these regulations need to be ignored. Instead of formalizing these rules, they could be represented as plain text, which is ignored by the system but shown to the person evaluating the design during the checking of the model. The architect or building committee can then decide whether or not the constraint is satisfied. While this might not appear to be useful at first glance, these non-formalized rules still serve a purpose. They act as a checklist of issues that might otherwise be forgotten. Even the worst-case scenario, where there are no constraints but only subjective regulations, would still be an improvement on the current situation, as it prevents overlooking any of them.

Achieving true mass customization in the building industry will require involving the client in the production process, as it is impossible to supply a tailored product without knowing the demand. Doing so provides several key benefits: architects no longer have to make multiple design alternatives, as this work is transferred to the clients. Customers achieve a much greater level of flexibility without the corresponding price increase that used to be associated with it in the traditional design process. The fact that clients now get a customized product from the start removes the need for post-delivery remodelling, saving money and resources and, consequently, the environment. It also reduces the total required construction time. Adopting mass customization does imply a certain amount of standardization in house design. Some might argue that every house should be unique since the context of every house is different, be it because of a different environment, different inhabitants or any of a long list of possible dissimilarities. This argument certainly has merit, but given that a majority of housing projects are mass-produced, with little in the way of customization, it can be argued that mass customization is a step towards this ideal rather than away from it. This new approach to the building production process does require a few significant changes to the design process, however, one of which will be discussed in the next chapter.
5 M.P.H.
It’s not wise to violate rules until you know how to observe them.

_Thomas Stearns Eliot_
3. **Designing with constraints**

The previous chapter discussed mass customization, which gives people the ability to modify the products they buy. Naturally, there are limitations to the adjustments customers can make, both from a technical perspective and because it is important not to overwhelm buyers with options. This chapter examines the process of automatically enforcing these limitations. This is done by first looking at the way constraints are used in other fields and contrasting this with the adoption of constraints in the building industry. Finally, it describes the way in which constraints will be used in this project.

3.1 **Constraints in other industries**

The idea of using constraints to automatically verify designs is not a new one. Constraints are used in many industries to automate design verification, though they may not be referred to as such. This paragraph looks at a few examples of the use of constraints in other industries.

3.1.1 **Electrical engineering**

Due to the increasing complexity of printed circuit boards and integrated circuits, Electronic Design Automation (a form of CAD), has become an indispensable tool in their design. Many steps of the design process are partly or fully automated, including placement, routing and power optimization (Rubin 1974, Rabaey et al. 2003, Scheffer et al. 2006). Placement refers to the circuit’s components, for which constraints include the total wire length, congestion and timing. Following the placement of the components comes the routing step, in which the components are connected through wires. In power optimization, circuits are modified to minimize power consumption without affecting the operation of the circuit. After the design of a circuit is finished, it is validated both through simulations and real-world benchmarks to confirm that it performs correctly.
3.1.2 Software engineering

In the software engineering industry there are several ways of applying constraints to a unit of code, among which static typing, unit testing and code contracts. They mostly have the same goal, but use a different methodology. In static typing, the type of each variable is known at compile time. Any attempt to treat it as a variable of a different type will result in a compile-time error. This in contrast to dynamic programming, where the type of a variable is mutable and thus not checked until runtime. Static typing can thus be seen as a constraint on the possible values that can be assigned to a variable. As an example, the following code is valid in JavaScript (a dynamically typed language), but will produce an error in C# (a statically typed language), because after the variable is initialized as a number, it is not allowed to replace the value with a string.

```javascript
var number = 2;
number = "not a number";
```

Unit testing is the practice of writing checks to see if individual units of code behave as expected (typically functions, though it is also common to test larger groups of code. In this case, it is referred to as integration testing). In particular, this makes it easy to see whether any behaviour has been broken after modifying part of the program (regression testing). Code contracts are assumptions about sections of code, typically the input or output of a function. These can be statically checked to catch potential errors at compile-time rather than runtime, though using only these techniques this is not possible in all cases. It is possible to go further by proving the correctness of a program mathematically (Backhouse 1986). This requires a clear specification of the inputs and outputs of the program. The resulting proofs, however, are long and cumbersome (Dijkstra 1976), which is why they are not often used in practice. All three mentioned methods can be seen as constraints. They formalize the criteria that the code should satisfy and can be checked automatically, thus preventing the programmer from making some types of mistakes.

3.1.3 Mechanical engineering

In a lot of respects, mechanical engineering is similar to building design. In both disciplines, three-dimensional objects are designed that have to obey a series of constraints. Despite the similarities, there are also clear differences between the two. Mechanical engineering has a much stronger tradition of storing design semantically rather than only as the resulting geometry. There are several — often complementary — avenues of research in this field. Parameterized solid modeling (Barr 1984, Requicha and Voelcker 1985, Sederberg and Parry 1986, Bettig and Shah 2003) aims to store designs as parameterized objects, optionally deformed.
by parametric spatial operations. The ability to change any of the parameters, even if subsequent transformations have been applied, makes parametric designs significantly easier to modify than non-parametric designs. In feature based modelling (Bronsvoort and Jansen 1993, Dohmen 1998, Bidarra and Bronsvoort 1999, van Leeuwen 1999) designs are decomposed into semantic features, such as through-holes, slots and grooves. Feature based modelling can overlap with parametric solid modelling in that the features are often parametric objects. The success of feature-based modelling can be seen in the fact that many commercialized mechanical engineering CAD packages offer one or multiple forms of feature-based modelling. The same approach of decomposing a design into semantic parts can also be applied on a larger scale, in which case it is referred to as component-based or modular design (Huang and Kusiak 1998). A fourth research field is that of constraint-based design (Light and Gossard 1982, Bouma et al. 1995, Rao 1996), in which designs are checked for, or generated based on, compliance with a series of constraints. Although constraints can be applied to non-parametric geometry, they are more commonly applied to parametric designs, due to the greater ease with which variables can be referred to — Constraint-based design is therefore frequently combined with feature-based design (Shah 1995, Anderl and Mendgen 1996, Gross 1996).

3.2 Constraints in the building industry

Compared to other industries, the building industry — and more specifically, the architecture domain — has seen little adoption of constraints, at least not in the sense that they are automatically checked. Naturally, building designs have to comply with a multitude of constraints, such as building codes and functional and technical requirements that follow from a client’s brief, but verifying these is still a manual process in most cases. Only in the past decade have constraints started to get some traction.

3.2.1 Digital Dormer

One project in which constraints are used is De Digitale Dakkapel (The Digital Dormer). The goal in this project is to shorten the delay between submitting a proposal for adding a dormer to a house and getting the permit for it (van Leeuwen et al. 2004). This is achieved by replacing the process of submitting blueprints to the municipality with a step-by-step guide on a website. After entering some basic information about the house in question, people can design their dormer by specifying the width, height, types of panels, material, etc. When this is done, the web service automatically checks the design against the building codes to determine whether or not a permit is needed. If this is not the case, no additional interaction with the municipality is needed, which speeds up the process considerably. In this project, constraints are represented in XML syntax, with the condition being Perl
code. As an example, the constraint that the distance between the left side of the dormer and the side of the house must be at least 50 cm is represented as follows:

```xml
<check id="103" expression="$left>=0.5" type="vrom">
  <fail>De afstand tussen de dakkapel en de linkerkant van het huis is kleiner dan 50 centimeter.</fail>
</check>
```

Expressing the constraints using a programming language requires that they are entered by someone with sufficient technical knowledge. In this thesis, an attempt is made to remove this barrier.

### 3.2.2 SMARTcodes

Another project in which the aim is to check building models for building code compliance is SMARTcodes (Wix et al. 2008, Borrmann et al. 2009, Eastman et al. 2009). SMARTcodes is an initiative by the buildingSMART alliance, whose goal is to promote BIM adoption in the building industry. The aim of SMARTcodes is to validate IFC models using IFC’s constraint framework. This is achieved by taking a natural language constraint (e.g. a building regulation) and mapping words and phrases to concepts defined in a dictionary created by the ICC (International Code Council). This is done in a process similar to highlighting (see figure 3.1). By using established terms, the computer can understand the constraints and apply them to building models, after which the results can be displayed textually or graphically (see figure 3.2).
3.2.3 **Revit**

The first high-profile application in architecture to offer support for constraints is Autodesk’s Revit (Strömberg 2006). Currently, it is limited to geometrical constraints only, such as specifying boundaries on lengths and distances between elements. Other types of constraints, such as those on materials or costs, are not yet supported. Additionally, constraints are mostly specified graphically or through entering values in predefined constraints, which somewhat limits the flexibility of the system. Despite these limitations, it is a milestone on the way to bringing constraints into the mainstream in the building industry since it marks the first time that automated constraint checking has been available to a large group of architects.
3.2.4 **Civil Engineering**

A field in the building industry in which constraints are more commonplace is civil engineering. Because the scale of projects is typically much larger than in architecture and because of the correlating increase in the amount of parties involved, there is a greater need for clear agreements between the different players. It is therefore not surprising that constraints have found a more widespread use in this field than in architecture. One discipline that makes heavy use of the constraint philosophy is systems engineering, which places great importance on the requirements gathering phase as a driver for the design process (Blanchard and Fabrycky 1998). Systems engineering is employed in many large organizations, ranging from Bell Labs to NASA (Hall 1962). Systems engineering is frequently used in civil engineering, with examples including ProRail (the company responsible for the Dutch railway system) and Rijkswaterstaat (the ministry that maintains the public waterworks in the Netherlands).

3.2.5 **Reasons for the Limited Adoption of Constraints in Architecture**

Although the building industry is not entirely devoid of examples of constraint usage, adoption of constraints lags significantly behind that of other industries. Several reasons can be given for this discrepancy. The first is the slow adoption of CAD in architectural design in general. Although not strictly speaking a necessity for using constraints, CAD packages, particularly those using BIMs, do make automated constraint checking considerably simpler. In recent years the traditional drafting tables have started being phased out at an increasing rate, but in many cases they have only been replaced by traditional CAD packages rather than BIM-based ones. The limited acceptance of constraints is therefore not entirely surprising. Another reason is that there is a greater tendency in architecture than in other domains, such as mechanical engineering, to treat the design holistically, rather than as distinct components that are designed by separate teams. The fact that the architect has few, if any, other parties to cooperate with means that there is much less need to account for the designs of others, which would advocate the use of constraints. This has begun to change in recent years, however, due to the increasing shift towards working in design teams.

Another issue might be the relative prevalence of non-objective constraints in architecture compared to other industries due to the stronger focus on issues such as aesthetics. Since these issues are deemed to be very important by architects but the associated constraints cannot be checked by the computer (insofar as they are explicitly formulated at all) architects might be tempted the dismiss the constraints that can be checked objectively.
3.3 Types of architectural constraints
Architectural constraints can be subdivided into many different types based on the topic of the constraint. Some of the more common types are:

▷ Geometrical constraints
Constraints on dimensions, such as the length or area of an element

▷ Structural constraints
Constraints regarding the strength of elements, such as the minimal supportable load of a floor

▷ Building physics constraints
Constraints about the climate of a building, such as the maximum allowable temperature in a room

▷ Material science constraints
Constraints in regard to the properties of materials, such as the durability of wood

▷ Financial constraints
Constraints on the cost of parts of a design or the design as a whole.

▷ Aesthetic constraints
Constraints intended to achieve a certain look, such as a required façade material

Another way of subdividing constraints, which is the one that will be used in the rest of this thesis, is based on the source of the constraint. This categorization includes legislation (laws, building codes, etc.), design constraints (constraints defined by the architect) and practical constraints (e.g. doors should be placed such that they can be fully opened). This subdivision is orthogonal to the categorization based on topic listed above — a geometrical constraint can be a legal, design or practical constraint and architectural constraints include geometrical, financial and aesthetic constraints.

3.4 Methods of using constraints
There are two ways of dealing with constraints, depending on who creates the design — the user or the computer. Although both approaches result in a design that satisfies all of the constraints, they have different characteristics and different application domains.
3.4.1 Constraint solving
The first way to use constraints is constraint solving, i.e. taking the constraints as input and trying to find a design that satisfies them (Kelleners 1999, Eggink et al. 2001, Belbidia and Alby 2003, Böhme and Cárdenas 2006, Donath and Böhme 2007). The precise manner in which this is done depends on the method that is used to generate the design and the complexity of the constraint. For simple constraints such as “the height of the wall must be between 2 and 2.6 m”, it is easy to generate all possible valid solutions. Computing all combinations of all the possibilities for each element results in a list of all valid design alternatives, from which an alternative can then be picked using any criterion, such as minimal cost. Depending on the size of the solution space, this can take a very long time (Ian et al. 2006).

As an example, consider a scenario in which a 1000x1500mm window is placed in a 5000x3000mm wall, the constraint being that the window is no closer than 500mm to the edges of the wall. Since position is a continuous variable, there is technically an infinite amount of solutions. However, since the accuracy with which buildings can be constructed is limited, the position variable can be turned into a discrete variable with a 1mm resolution with little effect on the end result. This results in a solution space of 1500x4000 = 6 million possible positions if the assumption is made that the window should fall entirely within the bounds of the wall and the only allowed movement is in the plane of the wall. Of these 6 million possible solutions, 500x3000 = 1.5 million are valid. This solution space can be significantly reduced by applying constraint propagation (Kumar 1992, Sannella 1994, Jussien and Lhomme 2002), which reduces the allowed domain of a variable once one of the variables that it is dependent on changes. For instance, once the position of the floor is determined, the domain for the vertical position of the walls is reduced to a single value, vastly reducing the overall solution space. The requirement for constraint propagation to be effective is the existence of dependent variables. Although a building design contains a sizeable number of dependent variables, there are still many independent variables, which means that there is a limit on how far the solution space can be reduced.

Instead of a solution space that is too large, it is also possible for a set of constraints not to have any valid solutions. In this case the problem is overconstrained. To resolve this, one or more constraints need to weakened, which is known as constraint relaxation (Fox 1983, Freuder and Wallace 1992). Assuming that the constraints that apply to every building design (such as building codes) do not over-constrain the problem, this will likely not pose a problem for the architect. The typical way of working will be to create a design first and specify the constraints afterwards. The constraints are added one by one, and thus the architect is immediately notified when the solution space is reduced to zero.
3.4.2 Constraint checking

The second way of using constraints is to make a design, check to see if it meets all the constraints, and adjust it if necessary. This method is called constraint checking. Because the designer has the advantage of common sense, it will take less time to converge on a good (though not necessarily optimal) design than by using the constraint solving method. In this scenario, the constraints serve more as a reminder than as the defining factor in the design. An example of this method of constraint handling can be found in the CAD package Navisworks and the Solibri Model checker (AECbytes 2009), which offers, among others, automatic checking for the unwanted intersection of elements in the 3D model. This method shifts the task of navigating the solution space to the user, which makes it easier to avoid suboptimal solutions but it runs the risk that part of the solution space is overlooked.

3.5 Decidability and computability

An automated constraint checking system will only be able to check constraints that can be computed. This requires that constraints are both decidable and computable (Davis 1985, Sipser 1996). Decidability means that the function can be evaluated in finite time. A well-known example of an undecidable problem is the halting problem, which is the problem of determining whether a program halts for a given input. Constraints that are not decidable should not be able to be handled by the system. The fact that a constraint is decidable does not automatically mean it can be evaluated in a short amount of time. A constraint that takes several hours to compute would make the system unpractical to work with.

Constraints in the building industry typically fall in one of two categories in terms of computability: either they are simple guidelines or rules of thumb than can be quickly calculated, or they require a computationally intensive numerical simulation, as is the case in for example constructive engineering or building physics constraints. The former is suitable for inclusion in a real-time constraint checking system, the latter is not. In some cases, it may be possible to replace a non-realtime constraint with a less computationally intensive approximation for the purposes of real-time constraint checking, with the full calculation only being run at a certain interval. This thesis focuses on real-time constraints. No distinction will be made between realtime and non-realtime constraints, since this distinction depends mainly on the computational power of the computer. As the speed of computers improves, more and more constraints will become eligible for real-time checking. Since constraints can in most cases be evaluated independently, constraint checking can be performed either sequentially or in parallel. This is an implementation detail that has no consequences for the suitability of constraints for realtime checking.
Conclusions

Constraints are required to make mass customization truly feasible. Achieving mass customization, however, is not the only argument for using constraints. Other industries have a long history of using constraints for verifying designs, with applications ranging from automated circuit board design to ensuring software correctness and streamlining large-scale projects. The building industry, on the other hand, has so far not seen a significant adoption of constraints, for varying reasons. There are no fundamental reasons why constraints couldn’t be used in the building industry, though. In this thesis the potential is explored for a system for specifying architectural constraints on building designs, specifically for use in mass customization. Since the goal of this project is to have the computer support the designer (both architect and buyer), the system will use constraint checking rather than constraint solving, as constraint solving allows no user input beyond setting the constraints. Constraint checking gives clients the freedom to make the design themselves and only indicates problems to them, matching the design intent. Additionally, it prevents performance problems that might arise from the combinatorial explosion of possible design solutions. In the next chapter, two of the three prototypes used to explore the use of constraint checking in architecture are discussed.
CHAPTER 4
Constraint checking & entry

The more constraints one imposes, the more one frees one’s self.

Igor Stravinsky
In the previous chapter the use of constraints in various industries was examined as a prelude to the development of a constraint-based design system. In order to explore such a system, a series of prototypes were created in a process known as rapid prototyping (Tripp and Bichelmeyer 1990, Chua et al. 2003). The goal of rapid prototyping is to facilitate gathering user feedback during development, making it easier to see when a change of direction is required. Since this project represents a significant change to the current way of working for architects, it is important to take their opinion into account. Rapid prototyping is a methodology that is advocated by the agile software development philosophy (Somerville 2001). It is interesting to note that the agile philosophy is almost diametrically opposed to systems engineering (Dept. of Defense 1974) in terms of when and how often requirements are gathered, even though both have the same end goal of delivering an end product that matches the client’s wishes. In this research project, three prototypes were developed (see figure 4.1 for an overview). The first prototype is used to test the hypothesis that using constraint checking in architecture is technically and conceptually viable. The second prototype was created to evaluate a visual method of constraint entry by asking architects to specify a series of constraints using the prototype. In the third prototype, which is discussed in the next chapter, an algorithm is developed that can interpret constraints written in natural language.

**Figure 4.1** Process overview
**4.1 Constraint checking**

Before progressing to more in-depth questions regarding constraint-based design, it is important to determine whether or not it is viable to use this design methodology in the building industry. Three aspects are chosen to determine this: the first is technical viability, i.e. is it technologically possible to apply constraints to a building design and get a result? The other two aspects are more conceptual: constraints should prevent undesirable designs while allowing valid designs. In order to test these aspects of viability, a prototype of a constraint checking system was developed. For this prototype, constraints were represented as assertions about building elements, similar to universal quantifiers from the field of mathematics. For instance, the quantification

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

means “for all natural numbers \( x \), \( x \) must be more than or equal to zero”. An example of a constraint might be “for all windows, the height of that window must be at least 1200 mm”. A constraint is therefore effectively a function that takes as its argument a list of building elements and returns a Boolean, i.e. true (all elements satisfy the constraint) or false (at least one element does not). Verifying the validity of a design is therefore a matter of applying all constraints to the design, and checking whether they all return true.

**4.1.1 Prototype implementation**

As illustrated in figure 4.2, 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 right corner. In this prototype, constraints are formulated in C# code, which is also the language the prototype was written in. This was done to simplify the implementation. During program execution, the constraints are converted to syntax trees (Aho and Ullman 1997) by compiling the constraints at run-time using the C# compiler present in the .NET framework (Calvert 2008). Syntax trees are a method of representing a section of code that is independent of the syntax. For example, both expressions below represent the same calculation, but one is written in prefix notation (as used in the Lisp programming language, for example) and one in the more commonly used infix notation.

\[
\begin{align*}
(\ + \ (* \ 2 \ 3) \ 4) \\
2 \ * \ 3 \ + \ 4
\end{align*}
\]

Abstract syntax trees (ASTs) remove the issue of syntax and present programs in a uniform way. For example, the AST for the calculation in the examples would in both cases be the one shown in figure 4.3. These syntax trees are converted to lambda functions, which can be applied to the building elements. Lambda func-
tions are identical to regular functions, save for the fact that lambda functions do not have an associated name. As an example, compare the following two functions (using the syntax of the Haskell programming language for the sake of brevity):

```
plus a b = a + b
\a b -> a + b
```

Both methods add two numbers together, and indeed the following two expressions yield the same result.

```
plus 2 3
(\a b -> a + b) 2 3
```

In the case of constraints, there is no particular reason to assign a name to the function, as all of the functions are generated dynamically. Using lambda functions means they can be passed directly to the function that performs the evaluation,
rather than having to use run-time compilation to create a new class to hold all of
the generated functions. Whenever a constraint is not satisfied, the syntax tree is
used again to create the error message that is presented to the user by converting
the AST back to natural language. The reason for this is that in this prototype the
constraints are specified using programming language, which can be difficult to
understand for non-programmers. For example, the AST for the constraint expres-

\[
\text{Window.Width > 2600}
\]

is

\[
((>) \ (\text{width \ (window)}) \ 2600)
\]

By traversing this tree in-order and producing the associated natural language
equivalent for each token (e.g. properties become “the <property> of”, compari-

\[\text{IfcBuildingElement} \]

\[\text{Properties} \]

\[\text{Angle} \]

\[\text{Bottom} \]

\[\text{Height} \]

\[\text{Left} \]

\[\text{Length} \]

\[\text{Location} \]

\[\text{Name} \]

\[\text{Rectangle} \]

\[\text{Right} \]

\[\text{Top} \]

\[\text{Width} \]

\[\text{X} \]

\[\text{Y} \]

\[\text{IfcWindow} \]

\[\text{Class} \]

\[\rightarrow \text{IfcBuildingElement} \]

\[\text{Properties} \]

\[\text{FrameWidth} \]

\[\text{GlassThickness} \]

\[\text{IfcDoor} \]

\[\text{Class} \]

\[\rightarrow \text{IfcBuildingElement} \]

\[\text{Properties} \]

\[\text{DoorThickness} \]

\[\text{FrameWidth} \]

\[\text{OpenAngle} \]

\[\text{IfcWall} \]

\[\text{Class} \]

\[\rightarrow \text{IfcBuildingElement} \]

\[\text{Properties} \]

\[\text{IfcFireDoor} \]

\[\text{Class} \]

\[\rightarrow \text{IfcBuildingElement} \]

\[\text{Properties} \]

\[\text{FireRating} \]

\[\text{DoorThickness} \]

\[\text{FrameWidth} \]

\[\text{OpenAngle} \]

\[\text{IfcFireWall} \]

\[\text{Class} \]

\[\rightarrow \text{IfcBuildingElement} \]

\[\text{Properties} \]

\[\text{FireRating} \]

\[\text{DoorThickness} \]

\[\text{FrameWidth} \]

\[\text{OpenAngle} \]

\[\text{Figure 4.4} \]

Prototype BIM class diagram
sons become “must be $<comparison>$”), the resulting error message becomes “the width of the window must be more than 2600.” This error message is then presented to the user. The building model used in this prototype is very simple and contains only walls, windows and doors. A class diagram is shown in figure 4.4. The next section discusses the results of the prototype in more detail.

4.1.2 User testing

To test the prototype, a floor plan from an existing housing project (Comwonen 2011) was used. As boundary conditions, the position of the exterior walls and the sanitary cores are fixed (see figure 4.6), which means that all six degrees of freedom (three degrees of position and three degrees of rotation) are constrained. The other walls have no constraints placed upon them, and can be freely moved and rotated. Table 4.5 shows the constraints that were implemented for this prototype. Aside from fixing the position in the X and Y directions or just the Y direction, more constraint types were implemented, but they did not end up being needed in the chosen design and their exclusion does not impact the conclusions.

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

**Table 4.5** Constraint types implemented in the prototype

Since the prototype did not support the deletion of objects, objects that were moved outside of the floor plan were considered to be deleted. The prototype was tested by several co-workers of the DDSS research group. Figures 4.7 through 4.10 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 4.7 and the top right of figure 4.10). No problems regarding the technical viability were found — none of the designs presented problems for the constraint checking algorithm. The resulting designs did not violate any of the given constraints and no valid designs were rejected, satisfying the other two aspects of viability.
4.1.3 Evaluation

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

▷ Since the warnings of constraints violations are presented in a different location it can be easy to overlook them. It would be preferred to have a warning in the viewport in addition to the warning in the constraint violation list.

▷ Since walls with constraint were not differentiated visually from walls without constraints, participants had to rely on trial and error to discover which walls they were allowed to move. A visual indication would be preferred.

▷ 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.

▷ Moving a completely fixed object 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. A floor plan with non-orthogonal walls might make this feature more useful.
4.1.4 Conclusions and discussion

In the introduction, three criteria were identified to determine the viability of constraint checking for architectural design — technical viability, the rejection of undesirable designs and acceptance of valid designs. Although neither the design nor the constraints used in this prototype are particularly complicated, the chosen approach has not revealed any fundamental technical problems. As desired, moving any of the fixed walls immediately produces an error and a user test showed
that a wide variety of valid designs are possible, despite the inability to change the size of walls or to add or remove windows and doors. Thus all three aspects are satisfied and there is little doubt that constraint checking in the building industry is possible, clearing the way to focus on the entry of the constraints.

Before proceeding to the constraint entry, there are a few considerations regarding the way constraints are treated in constraint checking. 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, one can question whether this is necessary; not allowing the constraint to be violated will prevent inadvertent changes. On the other hand, disallowing constraint violations makes it impossible to get to a new valid design via an intermediate state that is itself not valid. As an example, it would not be possible to change a 3x2 room to a 2x3 room by moving two of the walls independently if there was a constraint that the area must be exactly 6m². Another consideration is the conceptual approach towards imposing constraints, for which there are two possible philosophies: either starting with a completely fixed design and granting 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.

It might be possible to develop a hybrid solution that would offer the best of both worlds. For instance, making this a per-object choice means that objects that cannot be changed independently of neighbouring plots, such as the outer walls, can start fixed, while objects that do not influence other designs (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). 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 in which the constraints are entered. In this prototype, a programming language was chosen for the sake of simplifying the implementation. When putting a system like this into practice, however, there are other considerations to be made. Since architects will be the users that will most frequently be entering constraints, the constraint entry method should be tailored to them. The next section describes the various possibilities in which constraints can be entered.
4.2 **Constraint entry**

Since architects are the ones who will be entering the majority of constraints on a day-to-day basis, the interface should be designed with their needs in mind. The goal, therefore, is to find a method of constraint entry that is easy for architects to work with. There are several alternatives (Myers et al. 2006), which are described below.

4.2.1 **Synthetic language-based constraint entry**

The first option is to use a programming language. This is a natural choice because the amount of expressive power required of the constraint system is similar to that of a (simple) programming language and because programming languages are commonly used to express rules in many different domains. One advantage of this option is that the implementation is fairly straightforward. Additionally, though hard to prove, it is likely that at least a majority of all constraints can be formalized using a programming language, based on the use of programming languages to encode constraints in other industries. The main disadvantage is that programming languages are very formal and require a great attention to detail in order to correctly express oneself, which a lot of architects will likely not be used to. Aside from the precision required, there is the additional minor issue that many programming languages have a syntax that will not be familiar to non-programmers. For instance, in Java the translation of the constraint “The height of windows in brick walls must be between 1 and 2 m” might result in the following code:

```java
if (wall.material == materials.Brick) {
  for(window : wall.windows)
    assert(window.height >= 1 &&
            window.height <= 2);
}
```

This code sample shows a few examples of syntax that differs from natural languages, such as curly braces to define scope and the use of `&&` instead of `and`. Some of these issues could be remedied by using an Application Programming Interface (API) or a Domain-Specific Language (DSL) targeted at defining architectural constraints (Spinellis 1999). This reduces the amount of unfamiliar syntax the architect has to deal with. The same example constraint might then be expressed as something along the lines of:

```
window.height between 1 and 2 for window in windows of wall
if wall made of brick.
```
An example of a DSL that tries to remove unfamiliar syntax to the point of looking a lot like English is the Inform 7 programming language. It is a programming language specifically designed for creating text adventure games. A short extract of some sample code (Short 2011):

The Law Library is north of the Great Dining Hall. “Many [books of precedent] line these walls, containing every kind of contract that can be made to bind every kind of soul. A hole in the floor descends to the other, less savory portion of this place.” Some books are scenery in the Law Library. Understand “shelves” and “books” and “contracts” as the books. The description is “It is not as though you would understand the language in which they are written.” The great contract book is a thing in the Law Library. Understand “contracts” as the contract book.

This code sample defines a room, two objects in that room, and gives those objects descriptions and synonyms, so that for example the command “look at shelves” will produce the description of the books, rather than giving an error message that the meaning of the word shelves is unknown.

4.2.2 Natural language-based constraint entry

Although very different from programming languages and DSLs from a technical standpoint, natural language processing takes the concept of removing unfamiliar syntax even further, since it allows the architect to enter the constraints in a natural language, such as English. This removes the requirement for training on the part of the architect, since he can use the language he is familiar with. On the other hand, it greatly increases the difficulty of the implementation, as natural languages are far harder to interpret than programming languages, since natural languages have not been designed with automated interpretation in mind. Using the same constraint again, we could express it in any of the following (and a multitude of other) ways:

▷ The height of windows in brick walls must be between 1 and 2 m
▷ Windows in walls made of brick must be between 1 and 2 m high
▷ The height of any window in a brick wall must be higher than or equal to 1 m and lower than or equal to 2 m
The main difficulty in interpreting natural language is the presence of ambiguity — unlike programming languages, the precise meaning of words can depend on the context. There are different types of ambiguity (Hutchins 1992):

▷ **Category ambiguity**
   Ambiguity regarding the grammatical category (noun, verb, etc.) of a word. Compare for instance the use of the word *set* in the following sentences: “I set the box on the table”, “They are part of a matching set”, “Are you set?”

▷ **Homography**
   Two words with the same spelling having a different meaning. Compare for instance “His ear was infected” and “He ate an ear of corn.”

▷ **Transfer ambiguity**
   The same word having different meanings in different languages. Compare for instance “I had a chat with him” and “le chat est sur la table.”

▷ **Structural ambiguity**
   One sentence having multiple different interpretations. For example, “Flying planes can be dangerous” can mean both “It can be dangerous to fly planes” and “Planes which are flying can be dangerous.”
Ambiguities can be resolved through different means, such as context and real-world knowledge. These are difficult to simulate though. Supporting natural language input can be made more feasible by restricting certain language constructs, such as metaphors or similes. In general, the more formal and specific the language used, the easier it is for a computer to interpret.

4.2.3 **Visual constraint entry**

The three categories (programming language, DSL, natural language) mentioned in the previous paragraphs cover different types of text-based constraint entry. However, this is not the only possible method. It is also possible to use a graphical interface. One way of doing so is to represent the constraints as trees, mirroring their internal structure (Myers 1990). Examples include ConMan (Haeberli 1988), Microsoft’s Visual Programming Language (a programming language for a virtual robotics environment) (Microsoft 2011) and Yahoo Pipes (Yahoo! 2011), which is a way to customize RSS feeds. Figure 4.11 shows a feed that searches Yahoo News and Google News for news items from the World News category.

![Figure 4.11](image)

**Figure 4.11** Lego Mindstorms NXT

![Figure 4.12](image)

**Figure 4.12** Block-structured sentence
This approach has the advantage over text-based constraint entry that the full capability of the system is exposed to the user, since all the blocks that are available for use are listed. In a text-based system it is more difficult to predict whether a certain expression will be supported or not. The downside is readability. Especially with more complex trees the function of the constraint will not be immediately obvious. Another method that can be used to solve this problem is a hybrid between the tree structure and natural language solutions. The idea is to construct natural-language sentences from blocks. An approach similar to this is used in Lego Mindstorms NXT (National Instruments 2011), an environment for programming Lego robots (see figure 4.12). Figure 4.13 shows one example of what using this technique with constraints might look like. This approach combines the readability of natural language with the discoverability of the tree.

A variation on this method is to use a 3D visual programming language, as presented in “The CUBE language” (Najork and Kaplan 1991; see figure 4.14). However, the practical use of this seems limited, as it is not easy to quickly see the meaning of a rule. Additionally, it complicates interaction with the constraint, since a 3D environment requires orbiting as well as panning and occlusion may prevent the entire constraint from being visible at once.
4.2.4 **Prototype constraint representation**

For the first constraint entry prototype, the block-based *visual constraint entry* method was chosen, as this seemed to offer the most advantages and the fewest disadvantages and appeared to be a good trade-off between ease of implementation and ease of use. Since constraints can become difficult to understand as their length increases, constraints in the prototype were split up into four different sections. The four sections that constraints were split into are *elements, definitions, conditions* and *rules*. These four sections were derived from the possible expressions in list comprehensions in programming languages, since constraints are effectively a mapping of a list of input elements to a list of results. The next four paragraphs will explain each section in more detail, comparing them to the different parts of a list comprehension in a programming language. Haskell is chosen as an example, though the same holds true for other programming languages as well.

4.2.5 **Elements**

Elements determine the building objects the constraint will be working on. This section is the equivalent of a *generator* in a list comprehension. For example, in the code

\[ n \mid n \leftarrow [1,2,3,4,5] \]

The expression

\[ n \leftarrow [1,2,3,4,5] \]

is the generator. It means that the list comprehension will operate on the numbers one through five. Similarly, the elements section of a constraint might indicate that it operates on all doors and windows.

4.2.6 **Definitions**

Definitions allow local variables to be declared for the sake of brevity. They are equivalent to *local declarations* in Haskell. For example, in the code sample

\[ \text{square} + \text{square} \mid n \leftarrow [1,2,3,4,5], \text{let square} = n \times n \]

the generator is followed by a local declaration. This can be useful for referring to the same value more than once without having to repeat it more than once or for improved documentation. The given code sample produces \([2,8,18,32,50]\) as output.
4.2.7 **Conditions**

Conditions can filter the values produced by the elements section according to some predicate. Guards are the Haskell equivalent. In the following example, the input is reduced to the numbers three through five by the guard.

\[ \{ n \mid n \leftarrow [1,2,3,4,5], \ n > 2 \} \]

An example of a condition would be restricting a constraint regarding the quality of the wood of doors to only doors that are made of wood.

4.2.8 **Rules**

Rules, finally, are the actual constraints that are placed on the elements. There is no specific equivalent term in Haskell, since the list comprehension can produce any expression, not just a rule. This is the first part of the list comprehension. As an example,

\[ \{ \text{odd } n \mid n \leftarrow [1,2,3,4,5] \} \]

Would produce \{True, False, True, False, True\}. A possible rule for a constraint would be the minimum height of an element. A valid constraint must have at least an elements section and a rule. Definitions and conditions are optional.

4.2.9 **Example constraint representations**

In the following example, we will look at the constraint “Windows with metal frames must have a profile thickness of at least 30 millimetres”. The elements in the constraint are frames in windows. Not all frames in windows are considered, however, only the ones made of metal, so the condition is added that the material of the frame must be metal. Definitions could be used in theory, but since there are no expressions that are used more than once there is little practical value to do so in this case. The constraint finally, is that the thickness must be at least 30 millimetres. Table 4.15 show some additional example translations, taken from the Dutch building codes for dormers.
<table>
<thead>
<tr>
<th>Original constraint</th>
<th>Representation</th>
</tr>
</thead>
</table>
| The distance between the dormer and the eaves of the roof must be between 0.5 and 1m. | Element: Dormer  
Definition: d is its distance to the eaves of its containing roof  
Condition: -  
Rules: d must be more than 500 mm  
          d must be less than 1000 mm                                                                                                                                 |
| The width of the dormer must be not more than 1/3 of the width of the house         | Element: Dormer  
Definition: -  
Condition: -  
Rules: Its height must not be more than the width of its containing house divided by 3                                                                                          |
| The material of walls of dormers must be wood sheet, wood or zinc                    | Element: Walls in Dormers  
Definition: m is its material  
Condition: -  
Rules: m must be wood or m must be wood sheet or m must be zinc                                                                                                              |
| The depth of profiles of plastic frames must be at least 0.03m                       | Element: Frames in Dormers  
Definition: -  
Condition: Its material must be plastic  
Rules: The width of its profile must be at least 30 mm                                                                                                                        |
| Half-span roofs with an angle of less than 30 degrees cannot contain dormers         | Element: Dormers  
Definitions: -  
Conditions: The type of its roof must be half-span roof  
Rules: The angle of its containing roof must be more than 30 degrees                                                                                                          |

**Table 4.15**  Example translation of building code constraints
Figure 4.16 Prototype main screen

For each of these elements:
- Wait

Where we define that:
- its area is its Length times its Height

If it meets these criteria:
- its area must be more than 20 SquareMeter

It must satisfy these rules:
- its Length must be less than 12000 Millimeter

Figure 4.17 Puzzle piece editor

Figure 4.18 Autocompletion example
4.2.10 **Prototype implementation**

To test the approach described above in practice, a working prototype was created. The left side of Figure 4.16 shows the four sections of an example constraint. Constraints can be added (with the green ‘+’ buttons), removed (using the red ‘x’ buttons) and edited (with the pencil and paper icon). The floor plan on the right shows which elements satisfy the constraint (green) and the ones that do not (green). Constructing the various parts of the constraint is done by combining blocks, as shown in figure 4.17. To indicate visually that the blocks are supposed to be linked together, puzzle pieces were chosen as a visual metaphor. The left side of the screen holds a “library” of available pieces. Whenever a piece is placed, the library is updated to only show the pieces that can grammatically follow the sentence constructed so far. This speeds up the process of creating the constraints. By dragging these to the right, they are added to the sentence. Depending on the type of puzzle piece, a dialog may be shown prompting for further input. For example, placing a property piece will show a dialog asking the user to choose between length, height, colour, etc.

In order for the computer to be able to understand the constraints that are entered by the user, they must first be converted to computer-executable code. This is done as follows: first, the series of puzzle pieces is converted to a string consisting of a series of identifiers, which indicate the type of puzzle piece, followed by the user input for that piece (if any). For example, the constraint “its height must be more than 50 mm” is converted to

```
;ITS;PROPHeight;COMPINEQgt;INT50;UNITmm
```

i.e. an ITS piece (no input), a PROPerty piece with a Numeric value (Height), a COMParison piece specifying an INEQuality (greater than), an INTeger piece (50) and finally a UNIT piece (mm). This string is then parsed by the constraint parser, in which the allowed grammar is defined. This grammar is also used to filter the list of available puzzle pieces to show only the ones that are grammatically allowed at that point in the sentence. This makes it easier for the user to find the puzzle piece he wants. Figure 4.18 shows an example of the pieces that are shown while constructing our example sentence. Puzzle piece subtypes, such as the fact that property is numeric and that the comparison is an inequality are determined from the word used and do not require separate puzzle pieces. The parser converts each (group of) puzzle piece(s) to a lambda function, which are ultimately combined into a single lambda function using the grammar rules. The example above would be converted as follows (the syntax used here is that of C#, since that is the language the prototype was written in):
The lambda function for the comparison uses a concept called **lifting** (Partee 1987) to pass the element parameter to the two parameters of the comparison. Using these parts, the whole constraint becomes:

```csharp
element =>
   ((a, b) => a(e).MoreThan(b(e)))(element)
```

or, simplified:

```csharp
element =>
   element.GetProperty("Height").MoreThan(new MillimeterValue(50));
```

which is a function that takes a building element and returns a Boolean, i.e. a function that tells whether or not an element meets a constraint. This function is then applied to all the elements selected by the first and third section of the constraint (which are also converted to lambda functions) to see which elements violate the constraint.

### 4.2.11 Compatibility with IFC

The initial version of the puzzle piece prototype uses a custom Building Information Model. Naturally, it would be preferable to be able to specify constraints for models saved in a standardized BIM. Since IFC is the most widely adopted BIM in architecture, an attempt was made to make the prototype compatible with IFC. Since a good library for importing IFC’s native file format (the ISO 10303 Part 21 Step Physical File Format, which is based on the EXPRESS language defined by STEP) was not yet available for Microsoft’s .NET framework, the ifcXML format was used instead. Like the Part 21 format, this format is widely supported and it has the advantage that it can be handled by XML parsers, which are available as a library in many programming languages. Using ifcXML, it was possible to import a building design that was made in Revit into the prototype and perform
constraint checking on it. The process was not entirely hassle-free, however. As is the case in the standard EXPRESS-based file format, many references are used in the ifcXML format; if an object is used more than once, it is defined once and then referenced whenever it is needed. Although this results in smaller file sizes, it is not particularly practical to work with; a reference is merely an ID, with no indication of where to find the referenced element. Because of this, all references were replaced with the object they point to. Although it would be possible to use the resulting data structure directly, it is more convenient to extend it with additional information. The reason for this is that some common properties of objects are not available in a straightforward manner in IFC. For instance, walls do not have a length, width or height property. To get this information, the associated shape representation has to be examined. To give an example, most walls will be represented by an extrusion. Getting the dimensions for an ifcWallStandardCase involves the following steps:

▷ Take the Representation property
▷ Choose the correct item from the Representations property
▷ Choose the first item from the Items property
▷ The height of the wall is the Depth property
▷ Take the SweptArea property
▷ Take the Outercurve property
▷ Take the points in the Points property
▷ The length and width of the wall are the differences between the maximum and minimum x and y coordinates of those points, respectively.

Adding the additional properties eliminated the need to perform these steps every time.

4.2.12 Evaluation
To test this prototype, three academic colleagues and three practicing architects were asked to convert building codes written in natural language to the proposed grammar. The reason for choosing practicing architects is that they are the ones that will most commonly be entering constraints, since the architect's constraints vary with each project. They are therefore the main target audience of the system.
The colleagues were used to determine the presence of any major bugs before visiting the architects. Each test lasted an hour, starting with an explanation of the idea and an example. After this introduction, they were asked to try for themselves. Two colleagues were able to correctly transform constraints at the end of the test, though it must be noted that both of them are programmers. One of the two indicated that he first mentally converted the natural language text to programming code and then to the prototype’s grammar. Still, results indicate that the proposed system will probably not be difficult to learn for people with a programming background. The third colleague still had difficulties after the one-hour session, but he cited the language barrier (the grammar is based on English) as a possible cause.

Testing the prototype on the architects revealed an interesting spectrum of different opinions. The first architect had his doubts about the approach, but indicated at the end of the hour that with some more training he would probably be able to use the language to enter constraints. The second architect did not believe in using rules in the design process at all, saying “I first design the house and then argue about the rules”. He indicated that he would never use a program like this to design houses. The third architect was interested in the approach and did not have too much trouble learning to use the system, though some hints were still needed. Part of the reason for this might be that she was already familiar with the constraints from the CAD package Revit. She said that with one day of training she would probably be able to work with the system.

Although two out of three architects indicated that they would probably be able to work with the system, all three of them indicated that the prototype was too laborious, which was one of the reasons the second architect indicated he would not use the system. Having to drag the puzzle pieces from the library to the workspace for every word was deemed to be too time-consuming.

**4.3 Conclusions**

An initial prototype in which the use of constraints in architectural designs was tested revealed no technological obstacles, freeing the way to focus on the constraint entry method. The reason for doing so is that the architect has a key role in the proposed process, making the ease with which he can enter these constraints one of the more important aspects of this research. If assigning constraints to the design is too cumbersome, architects will not be inclined to adopt constraint-based design. The initial approach was to have users assemble constraints with blocks containing single words or short sentence fragments, since this method was judged to be fairly easy to implement and not too difficult to learn. Both of these assumptions proved to be true, and user tests demonstrated that the prototype was reasonably successful in allowing people to define constraints. However, a system
that is easy to use is not necessarily efficient. Several of the test candidates mentioned that the block method, at least as it was implemented in the prototype, is not a very fast method, as having to drag all the pieces to the correct location is a rather laborious process. There are some steps that could be taken to improve this problem. One solution, which was also suggested by some of the candidates, would be to automatically add a piece to the end of the sequence by double-clicking on it in the library. However, this would likely lead to only a small improvement, leaving the core problem intact. If a system like this is to be used in practice, a different, more efficient, approach will have to be used. Even if such a method is found, the responses of the first and second architect showed that this way of working will 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. This is expected, as the constraint-based design philosophy is not intended to completely replace the traditional approach to architecture. The next chapter explores an alternative method of constraint entry based on natural language.
CHAPTER 5
Natural language constraints

When ideas fail, words come in very handy

Johann Wolfgang von Goethe
5. Natural language constraints

In the previous chapter, a prototype for a visual method of constraint entry was tested. Although it performed reasonably well in terms of usability, the efficiency of the interface left a lot to be desired. It was therefore decided to try a different approach. Currently, the majority of constraints in the building industry are specified using a natural language, such as Dutch or English. Examples of this include building codes and functional requirements in the client’s brief. The flexibility of natural language means that it is able to express any constraint, making it a good medium for this. The downside to the flexibility of natural language is that automatically interpreting natural text is exceedingly difficult, as it requires not just an understanding of grammar, but also knowledge of the world and a sense of context. The process of mechanically interpreting natural language is known as natural language processing. In this chapter, the basics of parsing in general and parsing natural languages are reviewed before discussing a constraint entry prototype based on natural language processing.

5.1 Parsing and semantic analysis theory

Parsing is the process of using a formal grammar to convert a text to a data structure – typically a hierarchical data structure such as a tree. Parsing is used in several domains, the more common ones being interpreting and compiling programming languages, reading data files and analyzing natural language. Parsing typically consists of two steps; lexical and syntactic analysis. This is normally followed by semantic analysis, in which the resulting data structure is used in an algorithm. These steps will be explained in the following paragraphs.
5.1.1 **Lexical analysis**

The first step in parsing is to split up the single input string into individual units referred to as *tokens*. The algorithm that is typically used to do this is to loop over the input string one character at a time and either adding it to the current token or starting a new one, based on the possible tokens that have been defined. The end result is a list of tokens. As an example, a simple calculator might define three kinds of tokens: integers, which consist of one or more digits, an addition operator consisting of a single plus sign, and a subtraction operator (a single minus sign). Lexical analysis would then split the input string “12+3-7” into the tokens 12, PLUS, 3, MINUS and 7.

5.1.2 **Syntactic analysis**

After producing a list of tokens, the next step is to convert the list of tokens into a (usually hierarchical) data structure. This is done through a formal grammar, i.e. a series of *production rules* that map combinations of input tokens to the resulting output tokens, most commonly a *Context-Free Grammar* (CFG). The way these rules are expressed varies, but most resemble the Backus-Naur Form (BNF) (Knuth 1964), shown in the example below, which is one possible grammar for a simple calculator:

\[
\begin{align*}
<\text{calc}> &::= \text{INT} <\text{operator}> \text{INT}* \\
<\text{operator}> &::= \text{PLUS} | \text{MINUS}
\end{align*}
\]

The top-level production rule of this grammar is a calculation, which consists of an integer followed by zero or more occurrences of an operator followed by an integer. Formal grammars are often specified in a BNF-like syntax and subsequently converted to the host programming language by use of a parser generator such as ANTLR (Parr and Quong 1995) or Lex/Yacc (Brown et al. 1995). Alternatively, the grammar can be written in the host language directly through a technique such as *parser combinators* (Leijen and Meijer 2001). The idiomatic data structure for representing the resulting data structure is a tree. Again using the simple calculator parser example, a basic binary tree data structure is created to store expressions (using Haskell for the syntax):

```haskell
data Operator = Plus | Minus

data CalcTree = Node Operator CalcTree CalcTree | Leaf Int
```

which means that a tree is either a node containing an operator (plus or minus) and two subtrees, or a single integer. The parse rule could then become (using the Parsec library (Leijen 2001) as an example)
op = (Plus <$ char '+') <|> (Minus <$ char '-')

int = Leaf . read <$> many1 digit

calc = chainl1 int (Node <$> op)

or, in pseudocode:

operator parser:
  return Plus if the character ‘+’ is matched or
  Minus if the character ‘-’ is matched

integer parser:
  match one or more digits
  convert the digits to a number
  return a leaf with the number

calculation parser:
  match an integer and store the leaf in a variable
  whenever an operator followed by an integer is matched:
    replace the variable with a node with the old value
    as the left and the new integer as the right branch.
  return the variable

Interesting to note is that in Parsec there is no separation between lexical and syntactic analysis — parsers that match literal characters or strings are not differentiated from token- or expression-level parsers. This in contrast to more traditional parsers such as ANTLR, in which the separation is enforced more strongly. Applying the calc parser to the input “12+3-7” would produce

    Node Minus (Node Plus (Leaf 12) (Leaf 3)) (Leaf 7)

as output. In order to keep the example simple, spaces are not allowed in the expressions; trying to parse “1 + 2”, for example, will result in a syntax error.

5.1.3 SEMANTIC ANALYSIS

The final step is to evaluate the resulting parse tree. The algorithm used in this step depends entirely on the specific problem being solved. In the case of the simple calculator example, the desired value is the result of the calculation. Since the calculation is now stored in a binary tree, calculating the result can be done with a simple recursive algorithm:
eval (Node Plus l r) = eval l + eval r
eval (Node Minus l r) = eval l - eval r
eval (Leaf i) = i

which means that the result of a leaf is equal to the value it holds, while the value of a node is procured by adding or subtracting the values of its left and right children based on the operator.

5.2 Natural language parsing

Writing a parser for a natural language is significantly more complicated than writing one for a programming language, since natural languages have not been designed with automated parsing in mind. Problems include ambiguity (one word may have multiple meanings) and the flexibility of the vocabulary (since new words are frequently created, defining the entire vocabulary in the parser becomes impractical, if not impossible).

5.2.1 Corpora

The most widely used method of parsing natural languages is to use the grammar of the language in question, i.e. each word is assigned a grammatical function (noun, verb, adjective, etc.). Words are combined into groups such as noun parts and predicates and finally into a sentence. The first part of this process is called Part-of-Speech Tagging (Brill 1992), and is performed by using a large sample of tagged text, which is referred to as a corpus (King 1983, Brown et al. 1990, Manning and Schütze 1999, Collins 2003). 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. If the word either does not exist in the corpus, or if the word occurs in different roles in the corpus (compare for example “he saw the polished well” and “he polished the saw well”), the algorithm will have to determine the correct part of speech another way, typically employing probability.

5.2.2 Probabilistic grammars

In the previous section, an example was given of a deterministic parser, i.e. at no point in the process was there any ambiguity as to what type of token was encountered. This is usually the case in domains such as programming languages or data files. However, this is not true in all domains, with natural language processing being the most prominent counter-example. A well-known example 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 is also given 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) (Charniak 1997, Collins 2003). Using these frequencies, the parser can then check the neighbouring words to determine the more likely scenario. In addition to the grammar, the corpus can also be used in this process. Since “fruit flies” will occur more often as two nouns than as a noun followed by a verb, this will be chosen as the correct interpretation.

5.2.3 Machine learning

The size of the vocabulary of natural languages, combined with the speed with which new words get added to the language means that it is likely that words in the input text will not appear in the database of known words. Depending on the task for which it is used, the majority of natural language processing systems will therefore need a way to automatically assign or ask the user for the function or meaning of a word, based on the current model. This process, in which a model is built or expanded from annotated training data, is known as machine learning (Thompson et al. 1999). Machine learning is not limited to the meaning of words, or in fact to natural language processing. Many industries use machine learning to improve their models. The percentage of the training data that is annotated can vary, from zero (unsupervised learning) (Thompson et al. 1999, Banko and Brill 2001) to partial (semisupervised learning) to 100% (supervised learning) (Manning and Schütze 1999, Jurafsky and Martin 2000). The quality of the newly synthesized information depends on the quality of the current model — a poor initial model will result in a poor model after machine learning. A large domain of possible inputs will therefore require an extensive model before unsupervised learning can be used.

5.2.4 State of the art

The following overview of the state of the art in natural language parsing is by no means comprehensive, but is intended to give an impression of the current state of the field. The paper “Immediate-Head Parsing for Language Models” (Charniak 2001) describes two competing parsing models: immediate-head parsers and strict left-to-right parsers. In strict left-to-right parsers, the function of each word is determined for each word in turn, usually based on the trigram of the current word and the two preceding words. Backtracking may be required if the choice of
a word makes the subsequent words very unlikely. Immediate-head parsers try to recursively identify the tag, constituents and head (the most important word, e.g. the head of “the red ball” is “ball”) of a sentence fragment. This approach is based in the experience of the statistical parsing community that the properties of the words in a constituent are largely based on the head of that constituent. The paper shows two types of immediate-head parsers to have a superior performance to two strict left-to-right parsers. Most parsers that are used in natural language processing are lexicalized parsers, i.e. both the function of a word as well as the word itself are taken into account. The extra information compared to unlexicalized parsers, in which only the function of the word is preserved, typically results in a significantly improved performance. Klein and Manning, however, demonstrate that unlexicalized parsing can approach (though not match) that of modern lexicalized parsers, even beating older lexicalized parsers (Klein and Manning 2003). The significance of this is that unlexicalized parser are easier to work with and more time- and space-efficient.

As mentioned earlier in this section, probabilistic parsers produce a list of possible parses of a sentence along with the respective probabilities of those parses. Traditional probabilistic parsers simply choose the parse with the highest probability. Re-ranking parsers, on the other hand, first refine the probabilities of these alternatives by taking into account additional information, referred to as features. Collins and Koo identify 12 different feature types, including rules, bigrams (adjacent pairs of words) and head modifiers (Collins and Koo 2005). They show that using re-ranking can reduce errors by 13%. The same error reduction is achieved by a different parser built on the same principles (Charniak and Johnson 2005).

Conclusions
The most common method of interpreting natural language is to use an annotated corpus. The system being developed, however, has two properties that limit the availability of a suitable corpus. Since the application is intended to be used in the Netherlands, Dutch is the preferred language for entering the constraints. Unfor-
Fortunately, the availability of Dutch corpora is extremely limited compared to the much more widely used English corpora. As an added complication, constraints will contain a lot of building industry-specific terminology that is not found in general-purpose corpora, rendering this approach somewhat problematic. Since no building industry-specific Dutch corpora exist — existing legislation is not annotated and therefore of no use in determining the meaning of constraints — interpreting natural language will require either creating such a corpus or finding a parsing algorithm that does not require a corpus. The approach that was chosen for the prototype is a combination of these two: a database containing the meaning of words is created, but no corpus of existing constraints is used. Lacking the training data to perform unsupervised learning, the system will only use supervised learning.

The algorithm that is developed in this thesis is based on tree transduction due to the requirement that the system being developed not only needs to generate a parse tree, but that it also has to normalize the tree to a sufficient degree for conversion to programming code. This involves, among others, the need to fill in the gaps caused by omission of repetitions (e.g. “the wall must be more than 2 m and less than 3 m high” instead of “the wall must be more than 2 m high and the wall must be less than 3 m high”). Tree transduction is the best candidate to solve this problem, as it allows copying parts of the parse tree to fill in gaps.

5.3 System prototype: ConstraintSoup

Given the lack of a suitable corpus, corpora-based algorithms are not suitable for this project. Therefore, a grammar-based tree transduction approach was chosen. 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. 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, save for self-closing tags such <img>. An example of an HTML document is shown below.

```xml
<html>
<head>
  <title>Document title</title>
</head>
<body>
  <p>A paragraph of text</p>
  <p><img src="image.jpg"></p>
</body>
</html>
```
Parsing an HTML document is fairly simple, provided the document is syntactically valid. A very large percentage of the web pages on the internet, however, are not. Possible reasons include the author of handwritten pages forgetting to close a tag, 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 opened, which in many cases is not acceptable, since users expect the product to work with any page. 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. The HTML parser TagSoup (Cowan 2011) 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 necessary words occur somewhere in the sentence, the algorithm can deal with omitted repetitions and superfluous words and accepts many 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. I call this method architectural language parsing. The algorithm consists of five steps, which are shown in figure 5.1 and detailed in the following paragraphs.

FIGURE 5.1 Algorithm steps
5.3.1 **Tokenization**
The first step is to convert the input string into useable parts. This means converting the input to a list of separate tokens such as words and numbers by splitting the string on spaces and removing punctuation. This is a simple and basic procedure.

5.3.2 **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 “height” and “high” map to the height property of an element. In this prototype, the database was created manually. In theory, some of the associations could be derived from the IFD library (Grant 2008), though this will only help with building industry-specific terminology and not with general-purpose words. The word lookup 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 articles and some prepositions), or the word is not found in the database. The meaning of a word mainly consists of the type of the word, such as Element or Property, with additional information depending on the type of the word, such as the unit of a value. The word types implemented in the prototype are shown in table 5.2.

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

**Table 5.2**  Word types

There are a number of word types that do not yet appear in this table. In the constraints used to test the system, there were no verbs that were required to interpret the sentence. Comparative adjectives are interpreted as the associated adjective followed by a comparison. For example, higher is replaced with height and greater than. Qualitative adjectives are either not sufficiently objective (big, expensive, etc.) or require element filters, which will be discussed later on. Other word types, like verbs, have not yet been required so far.
This information is shown to the user, so that relevant words that do not yet have a meaning 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.

5.3.3 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 Domain-Specific Language similar to regular expressions that operate on the token level. These expressions are then converted to proper regular expressions and applied to the sentence. As an example, one rule replaces

\[= (\backref{i}) - (\backref{i}) \text{ mm}\]

(where \(i\) means an integer) with

\[\geq \backref{1} \text{ mm and } \leq \backref{2} \text{ mm}\]

(\(\backref{1}\) and \(\backref{2}\) are backreferences, i.e. they are replaced with the first and second integer matched in the pattern, respectively). This rule is added because without modification the dash would be interpreted as a subtraction rather than an allowed range. The sentence

\[2 \times \text{run} + 1 \times \text{rise stair} = 570 - 630 \text{ mm}\]

will therefore be changed to

\[2 \times \text{run} + 1 \times \text{rise stair} \geq 570 \text{ mm and } \leq 630 \text{ mm}\]

Without this rule, the resulting constraint would instead be

\[2 \times \text{run} + 1 \times \text{rise stair} = -60 \text{ mm}\]

In the prototype, 28 preprocessing rules are defined, which are listed in table 5.3. As the algorithm is developed further, most of these rules should become obsolete since they are specific instances of more general cases. Ideally, no preprocessing rules would be required at all.
<table>
<thead>
<tr>
<th>Replace</th>
<th>With</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{i / \i} ) van ( \text{i} ) maal ( \text{i} ) % van ( \text{i} ) maal</td>
<td>( \text{m}^2 )</td>
</tr>
<tr>
<td>vierkante meter</td>
<td></td>
</tr>
<tr>
<td>vrije hoogte</td>
<td>vrijehoogte</td>
</tr>
<tr>
<td>vrije doorgang</td>
<td>vrijedoorgang</td>
</tr>
<tr>
<td>dient van toepassing te zijn</td>
<td></td>
</tr>
<tr>
<td>dient (tenminste) (,)</td>
<td>( \text{1} ) ( \text{2} )</td>
</tr>
<tr>
<td>is gelijk aan ( \text{1} )</td>
<td></td>
</tr>
<tr>
<td>( \text{1} ) - ( \text{1} ) mm</td>
<td>( \leq \text{1} ) ( \text{mm} ) en ( \geq \text{2} ) ( \text{mm} )</td>
</tr>
<tr>
<td>minimum (( . )+)is</td>
<td>( \text{1} ) minimaal</td>
</tr>
<tr>
<td>oppervlak van de ramen</td>
<td>raamoppervlak</td>
</tr>
<tr>
<td>onder een hoek staan</td>
<td>hoek</td>
</tr>
<tr>
<td>uit komen</td>
<td>uitkomen</td>
</tr>
<tr>
<td>niet boven (( . )+)uitkomen</td>
<td>niet hoger zijn dan ( \text{1} )</td>
</tr>
<tr>
<td>houten plaatmateriaal</td>
<td>houtplaat</td>
</tr>
<tr>
<td>houten delen</td>
<td>houdtelen</td>
</tr>
<tr>
<td>afstand tussen (( . )) en</td>
<td>afstand ( \text{1} ) tot</td>
</tr>
<tr>
<td>((CompOrd _)(UnitVal _ _)(Unit ___))(( \text{Rel Distance} ))( .+ )</td>
<td>( \text{2} )( \text{5} )( \text{3} )( \text{1} )</td>
</tr>
<tr>
<td>((CompOrd _)( \wedge )(UnitVal _ _)(Unit ___))(( \text{van} ))( .+ )</td>
<td>(Rel Distance Distance)( \text{2} )( \text{1} )</td>
</tr>
<tr>
<td>^((Element _))(( \text{UnitVal } __ ))(( \text{Unit } __ ))(( \text{Prop } __ ))s</td>
<td>( \text{4} )( \text{1} )(CompOrd ==)( \text{2} )( \text{3} )</td>
</tr>
<tr>
<td>^( \text{w}^<em>(\text{Prop } __ ))( \text{w}^</em>(\text{Element } __ ))</td>
<td>( \text{1} )( \text{2} )(CompOrd ==)( \text{3} )( \text{4} )</td>
</tr>
<tr>
<td>( \text{w}^<em>(\text{UnitVal } __ ))( \text{w}^</em>(\text{Unit } __ )) ( \text{w}^* )</td>
<td>( \text{1} )( \text{2} )(CompOrd ==)( \text{3} )( \text{4} )</td>
</tr>
<tr>
<td>^( \text{w}^<em>(\text{UnitVal } __ ))( \text{w}^</em>(\text{Unit } __ )) ( \text{w}^* )</td>
<td>( \text{1} )( \text{2} )(CompOrd ==)( \text{3} )( \text{4} )</td>
</tr>
<tr>
<td>( _ \text{moet} )( _ \text{zijn} ) (CompOrd ==)( _ \text{zijn} )</td>
<td>( \text{1} )</td>
</tr>
<tr>
<td>( _ \text{moet} ).( _1,2 ).( _ \text{zijn} ) (CompOrd ==)( \text{1} )</td>
<td></td>
</tr>
<tr>
<td>((CompOrd _))( \text{w}^*(\text{UnitVal } __ ))(( \text{Unit } __)( \text{Rel } __)( .+ )</td>
<td>( \text{3} )( \text{4} )( \text{1} )( \text{2} )</td>
</tr>
<tr>
<td>(Rel _ In)(( _ \text{het} ))(( \text{Prop } __ )) (Rel Distance Distance)</td>
<td>( \text{1} )</td>
</tr>
<tr>
<td>( _ \text{tussen} )( \text{w}^<em>(\text{Element } __) ) (Bool And)( \text{w}^</em>(\text{Element } __) )</td>
<td>( \text{1} )(Rel Distance Distance)( \text{2} )</td>
</tr>
<tr>
<td>((Rel _ Distance))( .+ )(( \text{tot} ))</td>
<td>( \text{2} )( \text{1} )</td>
</tr>
</tbody>
</table>

**Table 5.3** Preprocessing rules
5.3.4 **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, with properties (function application) and elements (variables) having the highest priority, followed by arithmetical operators, comparisons and Boolean operators.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operation Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Boolean operator (e.g. <em>and</em>, <em>or</em>)</td>
</tr>
<tr>
<td>4</td>
<td>Comparison (e.g. <em>more than</em>)</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. <em>distance</em>)</td>
</tr>
<tr>
<td>0</td>
<td>Properties (e.g. height), Elements (e.g. <em>wall</em>), etc.</td>
</tr>
</tbody>
</table>

If multiple tokens have the same priority, the first one is chosen. 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. The tree construction algorithm in pseudocode:

```
After tokenization:
Width Window > 2 * Height Window
Assigning precedence:
0 0 4 0 2 0 0 0
After splitting the sentence:
> 2
   Width Window * Height Window
```

*Figure 5.4* Tree construction example
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))

Figure 5.4 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”.

5.3.5 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 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 figure 5.5.

Although correct in structure, there are two problems: the Window and Wall leaves lack the height property (which is needed to compare their heights) and the left leaf of the right branch is missing entirely. In order to fix this, the tree must be modified, in a process that in this thesis is referred to as sanitizing. For sanitizing the tree, a top-down recursive search strategy (Aho et al. 1986) was used. The algorithm starts at the top of the tree looking for a constraint. The And node results in a constraint, so the algorithm moves on to the branches. And itself requires two constraints, so the algorithm is recursively applied to the two branches, but now
the right branch is passed as a source for missing tokens to the checking of the left branch and vice versa, since it is common that a required token can only be found in the other branch. Going to the left branch, a comparison is found, which is one type of constraint. A comparison requires two values. The opposite branch is added to the list of branches that are sources for missing tokens, so that when checking the (Height, Door) branch both (Window) and (< Wall) can be used. To see whether (Height, Door) can produce a value, a list of production rules (Knuth 1986) is checked. These production rules are also used to determine the result type and required branch types in the first two steps. The production rules defined in the prototype are listed below.

▷ Property (which produces an element) + Element = Element
   example: frame + window = window frame

▷ Property (which produces a value) + Element = Value
   example: height + door = height of door

▷ Value (without unit) + Unit = Value
   example: 2000 + mm = 2000 mm

▷ Relationship (which produces a boolean) + Element + Element = Constraint
   example: above + roof + floor = roof must be above floor

▷ Relationship (which produces an element) + Element + Element = Element
   example: in + windows + dormers = windows in dormers

▷ Relationship (any other type) + Element + Element = Value
   example: distance + wall + door = distance between the wall and the door

▷ Operator (with precedence of add/subtract) + Value + Value (of the same type as the first one) = Value
   example: plus + 1000mm + 2000mm = 3000mm

▷ Operator (with precedence of multiply/divide) + Value + Value = Value
   example: times + 10mm + 20mm = 200mm²

▷ Comparison + Value + Value (of the same type as the first one) = Constraint
   example: more than + 20mm + 10mm = 20mm is more than 10mm

▷ Boolean operator + Constraint + Constraint = Constraint
   example: and + x > 10mm + x < 30mm = x > 10mm and x < 30mm
In the case of the current leaf, searching the production rules reveals one that says “Property + Element = Value”. All the required tokens are now present, so we can continue. For the right branch, 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 used earlier satisfies those two conditions, so a property token is needed. This is where the sources for missing tokens mentioned earlier come in.

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, 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.

Continuing with the example, Height is the only available Property node that can be combined with the Window to produce a value, so it is added to the Window leaf. This branch is now completed, so the algorithm continues with the right branch of the And node. Here again a comparison is found, which is correct as before, so the algorithm can move on to the branches. For the missing left branch the aim is 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 is shown in figure 5.6.

In the implementation, one more transformation is applied to this tree by con-
verting the nodes into objects according to the production rules. The main conse-
quence of this is that extraneous tokens are eliminated and that tokens are ordered
consistently. As an example, a Leaf with a Window and Height token would be
converted to a Value object with Property and Element parameters, removing the
need to search a list of tokens for one of the correct type. This is a trivial step, but it
makes the eventual conversion of the tree to executable code easier.

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 lan-
guages, they possess similar grammar. The main difference is the word order in
subordinate clauses — Dutch uses SOV and English uses SVO. The impact of this,
however, is limited since subordinate clauses are not very common in constraints.
Of the 31 building code constraints that were used to test the system, only two
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 not unlikely that the algorithm will also work for other Western Euro-
pean languages, due to the insensitivity to the specific word order.

5.3.6 Interface

The prototype is an implementation of the algorithm described in the previous
section. The interface, shown in figure 5.7, consists of a text box to type the con-
straint into, followed by the results of parsing the sentence. The first part of the
result is the list of recognized words 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. Figure 5.8 shows the window in which the word
type can be chosen (choosing either an existing word or a custom one), followed
by the window in which a new property can be defined. This is the way in which
new information is added to the system. Below that is the initial tree that results
from the tree construction step, followed by the end result after sanitizing the tree.
In order to make the end result more easily readable, the tree is also displayed as
an English sentence.
Conclusions

Parsing a natural language is considerably more difficult than parsing an artificial language such as a programming language. Due to the inherent ambiguity present in natural language, accurate parsing would require a probabilistic rather than a deterministic grammar. However, this does not mean that deterministic grammars are entirely ineffective. Especially when there is a limited context, as is the case in the domain of architectural constraints, the amount of ambiguity is often fairly limited. A bigger problem is the fact that programming language grammars are very specific about word order. Many common natural languages, such as Dutch and English, however, are very flexible with regard to their word order. Compare for example the following sentences that all have the same meaning: “Alice gives the apple to Bob”, “Bob is given the apple by Alice” and “The apple is given by Alice to Bob”. Handling all possible word orders would require a very large gram-
mar. Instead, the architectural language parsing algorithm that was developed in this research to process natural language works around the flexibility of natural language by making the order of the words partially irrelevant—by using production rules to manipulate the syntax tree gaps are filled, words are repositioned and superfluous words can be ignored so that constraints can be interpreted regardless of the way they are phrased. Word order is not entirely irrelevant though. For example, “Alice likes Bob” has a different meaning than “Bob likes Alice”. The algorithm therefore uses the given word order where possible and ignores it where needed. This allows a large variety of input to be processed by a very simple grammar. There are limitations to the system though. For example, emphasis, either on specific words or implied through word order, is not considered. This does not pose a very big problem, however, since none of the constraints that were used to test the system relied on emphasis for their meaning. Currently, the only way for the system to learn new words is to add them manually. Theoretically, it would be possible for the system to learn new words through machine learning, but this has no effect on the algorithm itself, as it is only a different method of populating the database. This addition is not explored further in this project. In the next chapter, this algorithm will be tested on real-world input to see how it performs in practice.
In theory there is no difference between theory and practice. In practice there is.

Yogi Berra
6. **Evaluation**

In order to adopt true mass customization in the building industry, clients should be given the ability to modify the design themselves instead of relying on the architect to come up with all of the design variations. However, their lack of knowledge regarding building codes and regulations requires that their designs are checked for problems. Preferably this check should be automated, as the verification process is time-consuming and error-prone to do manually. Before the computer can verify designs, the design rules need to be entered into the system. In the previous chapter the use of natural language processing as a method of constraint entry and a prototype implementation based on this approach were discussed.

In this chapter the prototype is tested on two types of constraints: *legislation* (which apply to every project) and *design constraints* (which are project-dependent). These test cases were chosen to represent the two most common sources of constraints that are expected to be most prevalent in the final system; the building codes and laws that every building must adhere to and the architect’s rules that are specific to a project. Design constraints are the more important of the two categories, since they will be entered far more frequently than legislative constraints, which often remain unchanged for many years.

Theoretically, the criteria for determining whether or not a constraint can be successfully interpreted are that the constraint is objective and that the required grammatical constructs have been implemented (this will be discussed in more detail later in the chapter). In practice, there are cases where interpretation fails because the implementation of the algorithm in the prototype is not yet completely bug-free.
6.1 **Legislation**

The first use case was 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. 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 the research is currently only targeted at one building). This leaves 31 constraints for the tests. For each of these 31 constraints, the desired resulting tree was specified. The constraints were then passed to the algorithm to see if the correct result was produced. The results of this test are shown below.

9 constraints were interpreted correctly without modification, shown (translated to English) in table 6.1.

<table>
<thead>
<tr>
<th>Original constraint</th>
<th>Correctly interpreted constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance dormer to eaves: &gt;= 0.5m and &lt;= 1 m</td>
<td></td>
</tr>
<tr>
<td>Profile width frames of dormer max. 0.07m</td>
<td></td>
</tr>
<tr>
<td>The distance of the dormer to the front façade must be more than 1 m</td>
<td></td>
</tr>
<tr>
<td>The height of the dormer must be less than 1.50 m</td>
<td></td>
</tr>
<tr>
<td>The bottom of the dormer must be more than 0.5m above the eaves of the roof</td>
<td></td>
</tr>
<tr>
<td>The bottom of the dormer must be less than 1m above the eaves of the roof</td>
<td></td>
</tr>
<tr>
<td>The top of the dormer must be more than 0.5m below the roof ridge</td>
<td></td>
</tr>
<tr>
<td>The sides of the dormer must be more than 0.5 m from the sides of the roof</td>
<td></td>
</tr>
<tr>
<td>Width dormer: max. 1/3 of the width of the house (axis common wall)</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.1** Correctly interpreted constraints
4 constraints required minor modification. For example, in the (translated) constraint “Height dormer: \(\leq 1.5\) m (incl. roof edge), what is meant is that the height of the roof edge should be added to the height of the dormer. However, since the fragment referring to the height of the roof edge is located on the right of the inequality, it gets added to the 1.5 m by default. By moving this fragment over to the left, the problem is prevented. These four constraints, along with the required modifications, are shown (translated to English) in table 6.2.

<table>
<thead>
<tr>
<th>Original constraint</th>
<th>Required modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walls dormer: dark grey</td>
<td>Walls dormer equal to dark grey</td>
</tr>
<tr>
<td>Height dormer: (\leq 1.5) m (incl.</td>
<td>Height dormer (incl. roof edge): (\leq 1.5) m</td>
</tr>
<tr>
<td>roof edge)</td>
<td></td>
</tr>
<tr>
<td>The walls of the dormer must be</td>
<td>The walls of the dormer must be equal to opaque</td>
</tr>
<tr>
<td>opaque</td>
<td></td>
</tr>
<tr>
<td>Walls dormer: wood sheet/wood parts</td>
<td>Material walls dormer: equal to wood sheet or wood parts or</td>
</tr>
<tr>
<td>zinc</td>
<td>zinc</td>
</tr>
</tbody>
</table>

**Table 6.2** Constraints requiring minor modifications

11 constraints were not yet handled correctly. Eight of these could be interpreted by adding three additional grammatical constructs:

- **Conditionals**
  Constraints that only need to be observed if a certain condition holds. Example: “If the roof is steeper than 30 degrees no dormers are allowed.” Conditionals are one way of implementing the decomposition of constraints mentioned in chapter 2.

- **Element filters**
  Constraints that apply only to a subset of all the elements of a give type. For instance, the constraint “Concrete floors should be at least 30 cm thick” refers only to floors that are made of concrete, rather than all floors.

- **Existentials**
  The algorithm currently only supports universal quantifiers, i.e. constraints of the form “for all elements X condition P must hold.” Some constraints require existential quantifiers, i.e. constraints of the form “there must exist an element X for which condition P holds.” Example: “The bathroom must contain a toilet.” The work of Richard Montague (Montague 1974) is relevant in this context.
The other three can be handled by making the grammar probabilistic, the problem for all of them being that the word roof can currently only refer to a roof as a separate element, and not to a roof as the property of another building element. They are shown (translated to English) in table 6.3

<table>
<thead>
<tr>
<th>Original constraint</th>
<th>Required addition(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frames dormer: free window layout / no closed parts / no panels</td>
<td>Existentials</td>
</tr>
<tr>
<td>The dormer must be built on an existing building</td>
<td>Existentials, element filters</td>
</tr>
<tr>
<td>The dormer must have a flat roof</td>
<td>Existentials, element filters</td>
</tr>
<tr>
<td>Distance dormer to common wall: &gt;= 0.5 m</td>
<td>Element filters</td>
</tr>
<tr>
<td>Profile depth plastic frames dormer min. 0.03m, no further requirements</td>
<td>Element filters</td>
</tr>
<tr>
<td>Because of that, placing a dormer on a gable roof with an angle less than 30° is undesirable</td>
<td>Element filters</td>
</tr>
<tr>
<td>Distance dormer to side façade &gt;= 0.5 m side yard not neighbouring public road or green space, &gt;= 2 m side yard neighbouring public road or green space</td>
<td>Conditionals</td>
</tr>
<tr>
<td>Distance dormer to front façade: (only in case of side gable roof) &gt;= 2 m side yard neighbouring public road or green space</td>
<td>Conditionals</td>
</tr>
<tr>
<td>For dormers on half-span roofs the same rules apply as for gable roofs. Depending on the angle of the roof and the height of the ridge and gutter different rules apply. If the angle is less than 30°, a dormer is undesirable. When the angle is less than 45° a dormer is acceptable if the height under the ridge measures more than 2.70 m.</td>
<td>Conditionals, make grammar probabilistic</td>
</tr>
<tr>
<td>Roof edge dormer: wood sheet / wood parts</td>
<td>Make grammar probabilistic</td>
</tr>
<tr>
<td>Roof edge dormer: cream</td>
<td>Make grammar probabilistic</td>
</tr>
</tbody>
</table>

**Table 6.3** Constraints that are not yet handled
The remaining seven constraints, shown (translated to English) in table 6.4, are more complicated and have no desired encoding yet.

<table>
<thead>
<tr>
<th>Original constraint</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>The limited size of the wolf end is unsuitable for additions. The sides are more</td>
<td>The limited size of the wolf end is unsuitable for additions. The sides are</td>
</tr>
<tr>
<td>appropriate for this and are to be treated as the gable roof.</td>
<td>more appropriate for this and are to be treated as the gable roof.</td>
</tr>
<tr>
<td>In case of gable roof with attic no dormer at the height of the attic</td>
<td>In case of gable roof with attic no dormer at the height of the attic</td>
</tr>
<tr>
<td>The character of hip-, tent- and pyramid roofs, with hip rafters tapering towards</td>
<td>The character of hip-, tent- and pyramid roofs, with hip rafters tapering</td>
</tr>
<tr>
<td>the ridge, requires very limited dimensions for the dormer. Placement should</td>
<td>the ridge, requires very limited dimensions for the dormer. Placement</td>
</tr>
<tr>
<td>respect the hip rafters and at least 1 m of roof should remain free, measured</td>
<td>respect the hip rafters and at least 1 m of roof should remain free, measured</td>
</tr>
<tr>
<td>from the top of the dormer.</td>
<td>from the top of the dormer.</td>
</tr>
<tr>
<td>An addition to the roof at the back of a mansard roof is permitted in the lower</td>
<td>An addition to the roof at the back of a mansard roof is permitted in the</td>
</tr>
<tr>
<td>section of the roof. The top connection with the roof should occur at the bend in</td>
<td>section of the roof. The top connection with the roof should occur at the</td>
</tr>
<tr>
<td>the roof.</td>
<td>bend in the roof.</td>
</tr>
<tr>
<td>A dormer high on the roof results in an imbalanced view on an asymmetrical</td>
<td>A dormer high on the roof results in an imbalanced view on an asymmetrical</td>
</tr>
<tr>
<td>roof and is therefore undesirable. The advice here is to move to the other part of</td>
<td>roof and is therefore undesirable. The advice here is to move to the other</td>
</tr>
<tr>
<td>the roof. A dormer of at most half the width of the house is acceptable if the</td>
<td>part of the roof. A dormer of at most half the width of the house is</td>
</tr>
<tr>
<td>lower part of the roof already contains a dormer.</td>
<td>acceptable if the lower part of the roof already contains a dormer.</td>
</tr>
<tr>
<td>The dormer may not be built on a dwelling that is not intended for permanent</td>
<td>The dormer may not be built on a dwelling that is not intended for permanent</td>
</tr>
<tr>
<td>occupation, a temporary construction or a caravan.</td>
<td>occupation, a temporary construction or a caravan.</td>
</tr>
<tr>
<td>The dormer must be built on a rear roof or side roof that is not oriented towards</td>
<td>The dormer must be built on a rear roof or side roof that is not oriented</td>
</tr>
<tr>
<td>the road or a public green space.</td>
<td>the road or a public green space.</td>
</tr>
</tbody>
</table>

**Table 6.4 Constraints lacking a desired encoding**

This results in 42% being interpretable with little to no modification, with an additional 35% percent requiring known additions to the algorithm. The changes that need to be made for the remaining 23% have not yet been considered in depth.

### 6.2 Design constraints

Although the recognition rate for legislation could be better, it is important to note that the type of language used by laws and building codes is different from the type used by architects. It has a tendency to be very formal and can feature fairly complicated sentence structures. In order to get a sample of the input, a user test was conducted on 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. The cases are shown in figures 6.5 through 6.14.

The students were asked to write one or more constraints to prevent these problems for an automated constraint checking system. No details of the system were provided and no limits were placed on the way the input should be formulated,
Figure 6.5  Case 1: gap in wall
Figure 6.6  Case 2: small window
Figure 6.7  Case 3: small toilet
Figure 6.8  Case 4: wrong material
Figure 6.9  Case 5: house too high
Figure 6.10  Case 6: unopenable door
Figure 6.11  Case 7: wrong colour
Figure 6.12  Case 8: shed too high
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 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 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 doesn’t specify whether this applies to the width, the length, the height or any combination of those). In 10 cases, the participant did not consider the design to be problematic. This occurred six times in the case where a house had vertical windows, whereas all of its neighbours had horizontal ones, shown in 6.13. This leaves 128 constraints out of the original 292, or 44%, that are eligible for use in the system, meaning that even with no training or explanation of the system whatsoever, almost half of the constraints are suitable for a constraint checking system. This indicates that, perhaps with a short training, architects should have little difficulty in formulating proper constraints. Figure 6.15 shows the distribution of the different categories of constraints.

As with the dormer constraints, the next step was for me 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 been implemented. Another 17 refer to the neighbouring houses, for which a satisfactory encoding has yet to be found. The remaining 14 also lack a desired encoding, since for varying reasons they are more complicated than the others. Of the 83 constraints, 53 were interpreted correctly. A further 23 could be handled correctly with a minor change in wording, similar to the building code test. The remaining 7 constraints were not yet handled correctly. This means that of the constraints for which a desired parse tree was formed, 92% could be correctly interpreted with little or no changes. Figure 6.16 shows this breakdown of the viable constraints.
Figure 6.15  Constraint categories

Figure 6.16  Valid constraint categories
6.3 Conclusions

The two tests demonstrate that the prototype can already handle a significant portion of the tested constraints, particularly the ones with simpler grammar, as evidenced by the differences in success rates between legislation and design constraints. There are two main ways of improving the success rate:

- **Additional language constructs**
  Three such constructs — conditionals, element filters and existentials — have been discussed already. Implementing these would raise the success rate for legislative constraints from 42% (13 out of 31) to 67% (21 out of 31), assuming none of the eight additional constraints would pose any further problems. The additions that have to be made to further increase the success rate have not yet been considered.

- **Richer BIM**
  A number of constraints currently lack a desired interpretation since there is no simple relationship between the language used and the code that would be required to determine the information. In some cases the required information is already present in the BIM, but not readily available. For instance, there is no simple way to determine which houses are the neighbours of the one being checked, though they are present in the model. One solution for this would be the ability to run spatial queries (Adachi 2003, Borrmann et al. 2006) on the model. In other cases, the information is missing altogether. This will require expanding the BIM standard.

One could say that a roughly three in four success rate is not sufficient for a practical implementation, since architects would too frequently encounter constraints that cannot be handled by the system. However, every constraint that the system can handle is one that can be automatically checked, meaning that it no longer has to be done manually. From that perspective, the system can already verify a significant percentage of constraints. This means that architects can have a greater degree of confidence in the building code compliance of their buildings and that housing projects in which mass customization is used require less focus of the architect on basic constraints, freeing him up to spend more time on the aesthetics and higher level goals of the design.
Conclusions & discussion

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing
In the previous chapters the implementation and subsequent evaluation of a prototype system for specifying constraints for architectural mass customization were discussed. In this chapter, the findings are summarized and directions for future research are identified.

7.1 Thesis summary

Giving buyers more freedom of choice when it comes to the design of their new house will require their participation in the design process. The two existing methods of achieving this – direct consultation and multiple choice housing – both have downsides related to efficiency and the amount of choice, respectively. Choosing a different approach where clients modify the design themselves, after which their designs are checked automatically for design constraints and building code compliance, could solve both of these problems. Although not planned, it is theoretically possible for the buyers to specify their own constraints. In terms of the type of language used and the types of constraints expressed, these are very similar to design constraints and would be treated as such. The practical use of allowing buyers to specify constraints can be debated, however.

Going down this path requires a method of entering the constraints into the computer. The goal of this project is to develop a method for architects to enter design constraints. A first attempt using visual programming was implemented and tested, but feedback from architects revealed this to be an unpromising direction due to the laborious nature of the constraint specification process. The subsequent switch to using natural language, while more complicated from a technical standpoint, is more familiar to architects. This approach significantly reduces the required amount of training since no new techniques will have to be mastered and since it does not significantly differ from the current text-based way of dealing with constraints.
The feasibility of the natural language input system was tested by creating a prototype and conducting two tests. In the first test, a set of existing building codes was used to measure the performance of the prototype on the formal language used in legislation. In order to get a more representative sample of the language used by the system’s target audience, an additional set of constraints was gathered from architecture students in the second test. These students, which were chosen as a representative sample of the target audience of the system, which are practicing architects, were shown a series of undesirable designs, such as a door that cannot be fully opened. Their task was to formulate constraints to prevent such mistakes in the future. Both sets of constraints were tested by comparing the system’s interpretation of the constraints with the desired interpretation to determine the prototype’s success rate. The results of the two tests, although not yet sufficiently successful for deployment in a real-world application, are encouraging for an early prototype. 42% of the building code constraints can be handled correctly with little to no modification, with an expected rise to 67% once the following grammatical structures are added:

- **Conditionals**
  Constraints that only need to be observed if a certain condition holds. Example: “If the roof is steeper than 30 degrees no dormers are allowed.”

- **Element filters**
  Constraints that apply only to a subset of all the elements of a given type. For instance, the constraint “Concrete floors should be at least 30 cm thick” refers only to floors that are made of concrete, rather than all floors.

- **Existentials**
  The algorithm currently only supports universal quantifiers, i.e. constraints of the form “for all elements X condition P must hold.” Some constraints require existential quantifiers, i.e. constraints of the form “there must exist an element X for which condition P holds.” Example: “The bathroom must contain a toilet.”

In the architectural constraints test, 59% required few or no changes to be interpreted correctly (41% requiring no changes and 18% requiring few changes), and is likewise expected to rise to 76% (by including the 11% that require additional grammatical constructs and 6% that are not yet interpreted correctly due to implementation issues) in the future. With these results, we can now answer the two research questions identified at the start of the project:
Is it possible to develop a method of specifying all geometrical and material specifications for a building component within a set of constraints that are determined by its functional or technical properties?

By using a formalized subset of natural language, the majority, if not all, of the desired specifications can be expressed. The fact that a constraint can be expressed does not inherently mean that the constraint can also be interpreted correctly by the computer for use in constraint checking, but initial test results are encouraging and suggest that in the future a majority of constraints will be able to be checked automatically.

Can the constraints be specified in such a way that little or no training is required on the part of the architect to start using it?

By using natural language, which architects are already familiar with, most of the training requirement is eliminated. The only training required would be to teach architects to phrase their constraints in an objective way. A test revealed that almost half of the constraints formulated by architecture students already meet the criteria. Training should be able to further improve this performance.

7.2 Strengths of the proposed approach

The architectural language parsing algorithm developed in this research to parse natural language constraints offers the following advantages:

- No full language grammar required
  The algorithm is designed to partially ignore word order. This has the advantage that a full grammar of the language does not need to be developed. This means that the algorithm can be easily implemented in different applications.

- Robust
  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. This makes the algorithm robust to minor grammatical errors.

- Simple implementation
  The algorithm is relatively simple, which means that it can easily be used in any application that wishes to use natural language processing for entering rules.
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 or domain-specific language.

▷ **Applicable to multiple languages**

The algorithm can theoretically support many languages. It has been tested on Dutch constraints and it is expected that other Western European languages are similar enough not to pose a problem. The algorithm might even work for other types of languages, but this will have to be tested.

### 7.3 Limitations and future research

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. This section lists a few venues of required research.

#### 7.3.1 Further development of the grammar

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 grammar has not yet been fully developed. There are two main omissions. The first is that several common grammatical structures, such as conditional (e.g. “if the angle of the roof is more than 30 degrees, no dormer is allowed”). Secondly, the ability to resolve ambiguities must be added to the algorithm. The most obvious way of achieving this is to apply the principles of PCFGs and give words multiple possibilities, with the probability being determined by using one or multiple criteria, such as trigrams. One way of doing this would be to expand the corpus formed by the constraints that were used to test the performance of the algorithm. These constraints, which are annotated with the desired resulting syntax tree, can be used to determine the most likely interpretation of new constraints.
7.3.2 **Decidability and computability**
Currently, all legislative and design constraints that were used to test the algorithm are decidable, since they consist of simple Boolean and arithmetical equations. As more grammatical constructs are added to the system, it will become possible to express undecidable constraints. Given the expected way in which the system will be used, however, this is not a major issue, as architects are expected to first make a design, after which they will place constraints on it. If any constraints are insoluble, or if they overconstrain the solution space, the architect’s own design will be rejected, which will immediately reveal the problem. The constraints that were used also posed no problems in regards to computability, with the entire testing process taking only a few seconds. Constraints that would produce problems are not too difficult to imagine, however. Especially constraints that would require simulation, such as “the light level in the living room should not be less than 100 lux at 12:00 on any day of the year” could take a long time to calculate.

7.3.3 **CAD interface for clients**
The majority of existing CAD applications is aimed at professionals. In constraint-based mass customization, however, clients are included in the design process. This will require creating a user interface that is aimed at non-expert users. Existing CAD interfaces are typically focused on efficiency, exposing a great deal of functionality to the user. This is a valid approach for users that use the software frequently. Clients buying a house, however, will only use the software once, and the focus of the client interface should therefore be on learnability (the ease with which tasks can be performed the first time) rather than efficiency (the speed with which tasks can be performed once learned).

7.3.4 **Standardizing constraints**
There are many applications in the building industry that work with constraints already, even if they are often not referred to as constraints. Constructive engineering applications indicate whether or not the construction will fail. Building physics software checks whether the climate in a building is acceptable. Projects such as ePlanChecking (IAI 2005) check for building code compliance. Each of these applications, however, only checks constraints within its own domain. There is currently no convenient way of checking, for example, constructive and building physics constraints from within the same application. Checking constraints from different domains currently requires opening the design in multiple programs. Aside from the risk of incorrect results caused by flaws in the importers, this is a very laborious process; when a violation is discovered the designer will have to go back to the CAD application, go to the location of the problem, fix it, re-export the file, check it again, etc.
There are advantages to being able to check constraints from multiple domains in a single application. By not having to switch applications, it is possible to get real-time feedback on the design. The cycle required to fix problems and re-check the design becomes much shorter. This is of particular use in collaborative design projects involving multiple disciplines: the ability to be alerted of constraint violations from different fields reduces the number of required design iterations, meaning that for example the building physics engineer can immediately see that moving a duct would be undesirable from an architectural standpoint. This does not necessarily mean that there should be a single program that does everything; constraints can be delegated to external programs, provided that they can properly import the building model and the constraints. The former can be achieved by improving IFC compatibility. Aside from the representation of the building model, the different applications also need to able to load the relevant constraints. While it is theoretically possible to store them as natural language, the chance of a natural language parser being added to every application is slim. The more likely scenario is that the constraints will be stored in an intermediate format.

More research will need to be conducted on what the most appropriate format for storing constraints would be. While no single option can be recommended as being superior at this moment, several alternatives can be identified. The first option would be to use the IfcPropertyConstraintRelationship in IFC. This option has a few disadvantages, which stem from the fact that constraints can only be specified on properties. The first is an inability to set constraints on attributes defined in the schema, such as the overall height of a door. Second, constraints that include arithmetic, e.g. “The width of the dormer cannot be more than one third of the width of the house” are inexpressible. Finally, constraints that do not reference properties, such as “these two walls must be connected” become significantly harder, or even impossible, to define. These problems make this option a less desirable one.

Another promising venue of research is the SMARTcodes project (ICC 2011), which defines constraints in EXPRESS-X, an extension to the EXPRESS language used by IFC. Storing constraints in EXPRESS-X is not very different from storing them in general-purpose programming code, except for the fact that in the SMARTcodes project the code is automatically generated. A downside of using EXPRESS-X is that it is not a very widely adopted standard. There are fairly few working implementations, which will slow down widespread adoption. The SWOP project (Swop 2009) opts for W3C’s Web Ontology Language (OWL) instead. Constraints are stored in a tree that is very similar to the Abstract Syntax Trees (ASTs) that underlie virtually all programming languages (Louden 1997).
Figures 7.1 and 7.2 (E-Bouw 2009) show the class hierarchy used to create these trees in SWOP. Like most ASTs, they distinguish nullary, unary and binary nodes, i.e. nodes with zero, one or two children. In the SWOP project, however, it is also used when entering them. Constraints are entered by creating the constraint tree, which is not particularly quick or intuitive.

7.3.5 Compatibility with IFC

The prototype demonstrated that constraint checking on IFC models is possible. However, IFC was not explicitly designed for that purpose, as evidenced by the lack of several common properties and classes that would be used for constraints. The missing properties could be added through property sets, a mechanism present in IFC that allows new object properties to be defined. Some examples include the length, width and height of a wall, the texture of a material (rough, smooth, etc.) and the durability class of wooden materials (which determine whether or
not they can be used in the exterior of a dwelling). The missing classes, such as walk-in closets, dormers or sheds, pose a greater problem, as there is no easy way of extending the IFC hierarchy. It is difficult to estimate how many new properties and classes would be necessary to represent everything architects will need. As an illustration, the documentation page for IfcExtendedMaterialProperties, a class which facilitates the definition of material properties that do not yet exist in IFC, lists (among others) the thermal gradient coefficient for moisture capacity and the index of refraction for solar rays. The only way to get a good overview of the missing information would be to do an experiment with a large group of architects and noting what concepts they use to define the constraints.

There are three possible ways of working around the limitations present in IFC. The first disadvantage is to create an information source separate from IFC from which the missing information, such as the durability class of a material, can be gathered. This approach was used to a limited extent in the prototype (a few extra properties, such as colour, were hardcoded). Such a database could consist of a series of tables with data (e.g. colours, material types, etc.) and one large table that links together a property name and an object with a value, e.g. “IfcDurabilityClass, Maple, 5”. This is not the most attractive option, since the fact that it is not a part of the IFC standard means that there is a very large chance that other software will not be compatible with it, which is exactly the problem that IFC is supposed to solve.

Alternatively, an extension to the IFC standard could be devised. Due to the nature of ISO certification, however, this will not be a quick solution. This extension would most likely consist of a series of classes to store the AST of constraints, much like the ones used in the SWOP project, i.e. classes to define binary operations (e.g. addition), property access, etc. as well as additional classes and properties to fill in the current gaps. Finally, a new standard that is tailored for constraint checking could be created. The main improvement over IFC would be to make this new standard extensible, similar to OWL (Lacy 2005). This allows programmers to add their own classes or properties to the model without having to wait for ISO standardization or having to make a distinction between properties defined in property sets and attributes from the schema. The main downside of this latter solution is that it will take a lot of work to reach the adoption rate that IFC has.

7.3.6 Functional specifications

Although in this research mass customization was motivated as an application area that could be promoted by the availability of a method to express architectural constraints, other application areas that could benefit from the presented method are design performances, building codes and product specifications.
New types of contracting have emerged over the last decades. One of these is performance based design. Instead of specifying the spatial layout and materialization of a building, performance criteria are expressed in a client’s brief. Performance criteria provide more freedom for the designers, engineers, and contractor to employ their expertise in the project. Performance criteria can by checked during the design process and make the project more controllable with regard to the client intentions and the financial conditions.

Although the concept is appealing, in practice it is far from easy to express a desired building without becoming very abstract. Because investors are very cautious to end up with a building that does not conform the market standard, design performance documents tend to be very lengthy reports stating any kind of performance criteria that should be met by the design. Because the documents are hard to cope with, there is a serious risk that not all performance criteria are seriously considered. Thus in contrast with its intentions, performance-based design does not always lead to better design, because the performance criteria are hard to manage during the project. Such performance criteria are often expressed as constraints. If constraints are expressed in a computational format instead of textual reports, the performance criteria can be checked against a digital model of the design during the design process. Computer software is needed that supports this checking process, but this research has shown that this is technically feasible.

Building codes are relatively stable examples of long lists of constraints. Since building regulations that are relevant in a building project are often spread out over different building codes it is hard to ensure that all regulations are checked. Because these constraints do not change frequently, books and computer programs are developed to cover common building code domains such as fire safety, sustainability, etc. Development of these books and applications requires interpretation of the building code by building experts. The building code books themselves are usually written by people with legal skills. Transferring the intentions of the government into regulations that can be used by practitioners is a long process with a serious risk of loss of information and lack of consistency.

A method to formalize building regulations into expressions that can be checked with regard to their ambiguity and consistency is a step forward in quality management and assurance of building codes. Building codes are essentially constraints and thus the research presented in this thesis is a good basis. From the presented digital dormer example we know that typically very long regulations need to be split up into shorter ones, and that qualitative constraints cannot be specified at all.
Design constraints do not only apply to complete designs, but also to their constituent parts, namely the building products. Traditionally, information about building products is distributed by suppliers to architects and engineers through brochures and lately through product databases. In practice, the architect or engineer picks a product instance and includes the 2D/3D drawing and its specifications in the building model. However, in this process all information about other product instances is lost. If, instead, a parameterized, constrained product instance is available, the product can inherit this knowledge while it is included in a building model. If the user makes changes to the model, the system can check whether the instantiated product is still valid, or if another instance better meets the designer’s conditions. To enable this process, building product suppliers need a tool to express their product constraints such as presented in this thesis. Furthermore, computer programs used by architects and engineers must allow interpreting these constraints.
A man will turn over half a library to make one book.

Samuel Johnson
**A. Legislative Constraints**

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afstand dakkapel tot dakvoet:</td>
<td>&gt;= 0.5 m en &lt;= 1 m</td>
</tr>
<tr>
<td>Afstand dakkapel tot woningscheidende muur:</td>
<td>&gt;= 0.5 m</td>
</tr>
<tr>
<td>Afstand dakkapel tot zijgevelijn:</td>
<td>&gt;= 0.5 m zijerf niet grenzend aan openbare weg of groen, &gt;= 2 m zijerf grenzend aan openbare weg of groen</td>
</tr>
<tr>
<td>Afstand dakkapel tot voorgevelijn: (alleen bij dwarskap)</td>
<td>&gt;= 2 m zijerf grenzend aan openbare weg of groen</td>
</tr>
<tr>
<td>Hoogte dakkapel:</td>
<td>&lt;= 1.5 m (incl. dakrand)</td>
</tr>
<tr>
<td>Breedte dakkapel:</td>
<td>max. 1/3 van de woningbreedte (as bouwmuur)</td>
</tr>
<tr>
<td>Kozijnen dakkapel:</td>
<td>raamindeling vrij / geen dichte delen / geen panelen</td>
</tr>
<tr>
<td>Wanden dakkapel:</td>
<td>houten plaatmateriaal/ houten delen/ zink</td>
</tr>
<tr>
<td>Dakrand dakkapel:</td>
<td>houten plaatmateriaal/ houten delen</td>
</tr>
<tr>
<td>Wanden dakkapel:</td>
<td>donkergrijs</td>
</tr>
<tr>
<td>Dakrand dakkapel:</td>
<td>crèmewit</td>
</tr>
<tr>
<td>Profielbreedte kozijnen dakkapel max.</td>
<td>0.07m</td>
</tr>
<tr>
<td>Profieldiepte kunststofkozijnen dakkapel min.</td>
<td>0.03m, geen nadere welstandseisen</td>
</tr>
<tr>
<td>Daarom is het plaatsen van een dakkapel op een zadeldak met een helling kleiner dan 30° welstandshalve ongewenst.</td>
<td></td>
</tr>
<tr>
<td>De beperkte maat van het wolfseind is ongeschikt voor toevoegingen. De zijdakvlakken zijn hiervoor meer geschikt en dienen te worden behandeld als het zadeldak.</td>
<td></td>
</tr>
<tr>
<td>Bij zadeldak met vliering dakkvorm dus geen dakkapellen op dakvlak ter hoogte van vliering.</td>
<td></td>
</tr>
</tbody>
</table>
Het karakter van schild-, tent- en pyramidedaken, met naar de nok toelopende hoekkepers, vereist een zeer beperkte afmeting van de dakkapel. Bij situering dient er respect te zijn voor de hoekkepers en dient minimaal 1 m dakvlak vrij te blijven, gemeten aan de bovenzijde van de dakkapel.

Een daktoevoeging aan de achterkant van een mansardedak is toegestaan in het onderste deel van het dakvlak. De bovenaansluiting met het dakvlak dient op de knik van het dakvlak plaats te vinden.

Voor dakkapellen op lessenaardaken gelden dezelfde uitgangspunten als voor zadeldaken. Afhankelijk van de hoek van het dak en de nok- en goothoogte gelden verschillende regels. Wanneer de hoek kleiner is dan 30° is een dakkapel welstandshalve niet wenselijk. Bij een hoek kleiner dan 45° is een dakkapel aanvaardbaar wanneer de hoogte onder de nok meer dan 2.70 m meet.

Een dakkapel hoog in het dakvlak geeft bij een asymmetrisch dakvlak een onevenwichtig beeld en is welstandshalve niet gewenst. Het advies hier is omzetten naar het andere dakvlak. Acceptabel is, ingeval op het benedendakvlak reeds een dakkapel staat, een dakkapel van maximaal de helft van de woningbreedte (trendsetter).

De dakkapel moet gebouwd worden op een bestaand gebouw

De dakkapel mag niet gebouwd worden op een niet voor permanente bewoning bestemde woning, een tijdelijk bouwwerk of een woonwagen

De dakkapel moet gebouwd worden op een achter-dakvlak of een zij dakvlak dat niet naar de weg of openbaar groen is gekeerd

De afstand van de dakkapel tot de voorgevel moet meer zijn dan 1 m

De dakkapel moet voorzien zijn van een plat dak

De wanden van de dakkapel moeten ondoorzichtig zijn

De hoogte van de dakkapel moet minder dan 1.50 m zijn

De onderzijde van de dakkapel moet meer dan 0.5m boven de dakvoet liggen

De onderzijde van de dakkapel moet minder dan 1m boven de dakvoet liggen

De bovenzijde van de dakkapel moet meer dan 0.5 meter onder de dakknap liggen

De zijkanten van de dakkapel moeten meer dan 0.5 m van de zijkanten van het dak liggen
**B. Design constraints**

### Misaligned Walls

<table>
<thead>
<tr>
<th>Misaligned walls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Muren (incl ramen) moeten de binnenruimte omsluiten</td>
</tr>
<tr>
<td>muren moeten aansluiten of er moet minstens 1.5 meter tussen zitten</td>
</tr>
<tr>
<td>Een ruimte moet helemaal afgesloten kunnen worden, wat betekent dat muren op elkaar moeten aansluiten.</td>
</tr>
<tr>
<td>Uitvoerende bedrijven moeten betere opletten wat ze doen en daardoor fouten in de bouw voorkomen.</td>
</tr>
<tr>
<td>Huizen moeten volledig afgesloten kunnen worden</td>
</tr>
<tr>
<td>Betere Maatvoering</td>
</tr>
<tr>
<td>Een meer gedetailleerd prefab element</td>
</tr>
<tr>
<td>Openingen minimaal de breedte hebben om er doorheen te kunnen lopen, dus 60cm</td>
</tr>
<tr>
<td>Afscheidingsmuren dienen aaneengesloten te zijn.</td>
</tr>
<tr>
<td>Alle wanden, vloeren en daken moeten bij elkaar een gesloten ruimte creëren.</td>
</tr>
<tr>
<td>Minimum eisen aan afdichten van ruimtes.</td>
</tr>
<tr>
<td>Muren moeten op elkaar aansluiten</td>
</tr>
<tr>
<td>Glas plaatsen of muur doortrekken</td>
</tr>
<tr>
<td>- Een gebouw moet geen kieren bevatten</td>
</tr>
<tr>
<td>vrije doorgang van minimaal (bijvoorbeeld) 900mm</td>
</tr>
<tr>
<td>openingen moeten gesloten zijn of gesloten kunnen worden met een deur of een raam</td>
</tr>
<tr>
<td>de buitenschil dient gesloten te zijn</td>
</tr>
<tr>
<td>De gevel moet termisch gesloten zijn</td>
</tr>
<tr>
<td>Een woning dient water en winddicht te zijn</td>
</tr>
<tr>
<td>muren moeten op elkaar aansluiten</td>
</tr>
</tbody>
</table>
1. Een binnenruimte heeft geen open verbinding met een buitenruimte.
2. Een binnenruimte is altijd te scheiden van een buitenruimte.
   Een opening naar de buitenruimte moet een raam of een deur bevatten.
   Tussen binnenruimte en buitenruimte mag geen open verbinding zijn.
   Buitenmuren moeten op elkaar aansluiten.
   Binnenruimte moet niet doorlopen naar buiten.

<table>
<thead>
<tr>
<th>B.2 SMALL WINDOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Er moet voldoende lichtinval zijn in een kamer.</td>
</tr>
<tr>
<td>Het raamoppervlakte in een woonkamer moet 1/4 zijn van het muroppervlak.</td>
</tr>
<tr>
<td>Er dienen minimaal 2 wanden te zijn met daarin openingen.</td>
</tr>
<tr>
<td>De openingen in het raam moeten minimaal een oppervlakte hebben van 2 vierkante meter.</td>
</tr>
<tr>
<td>De openingen mogen alleen gesitueerd zijn vanaf een meter vanaf de grond.</td>
</tr>
<tr>
<td>Er moet een minimaal aan 6000 lux binnen vallen.</td>
</tr>
<tr>
<td>Het raamoppervlak moet minimaal 10% van het vloeroppervlak zijn.</td>
</tr>
<tr>
<td>Raam moet dusdanige afmetingen hebben dat de kamer voor 80% met daglicht verlicht kan worden.</td>
</tr>
<tr>
<td>In een woonkamer moet minstens ...lux binnenvallen</td>
</tr>
<tr>
<td>raam groter op ooghoogte voor uitzicht</td>
</tr>
<tr>
<td>Raamoppervlakte moet minstens 10% van vloeroppervlakte zijn.</td>
</tr>
<tr>
<td>De verlichtingsterkte vanuit de omgeving op het horizontale vlak moet tussen de 300-900 lux liggen. De onderzijde van het raamkozijn dient maximaal 500 mm boven het vloeroppervlak gepositioneerd te zijn en de bovenzijde van het raamkozijn dient maximaal 400 mm onder het plafondniveau gepositioneerd te zijn. Een absoluut minimum van het raamoppervlak is 25% van het totale wandoppervlak.</td>
</tr>
<tr>
<td>Het oppervlak van de ramen mag niet minder zijn dan de 1/3 van het geveloppervlak.</td>
</tr>
<tr>
<td>de daglichtfactor moet aan een bepaalde waarde voldoen</td>
</tr>
<tr>
<td>Ramen van minimaal 2 m2 in minimaal 1 gevel.</td>
</tr>
<tr>
<td>Raam op hoogte dat er doorheen kan worden gekomen en de grootte zodat er voldoende licht naar binnen valt</td>
</tr>
<tr>
<td>Het oppervlak van het raam geliefd even groot als de smalste wand van de kamer.</td>
</tr>
<tr>
<td>Lichtintensiteit minimum invoeren</td>
</tr>
<tr>
<td>Raamgrootte moet 20% van kamer-vloer oppervlakte beslaan en zodanige geplaatst dat men er op ooghoogte doorheen kan kijken.</td>
</tr>
</tbody>
</table>
In een ruimte moet minimaal 30% van de muur een raam zijn

Woonkamers dienen minstens een groot raam te hebben (800x1200,1)

Raam moet groter, of er moeten meer ramen in de ruimte komen

Ruimtes moeten voldoende zonlicht krijgen.

- Raamopening dient op ooghoogte aanwezig te zijn, met uitzondering van de WC wellicht

- Er moet een minimumgrens aan het totale raamoppervlak te zitten, bijvoorbeeld 60% van het muuroppervlak in de woonkamer

- Een andere manier zou zijn om de totale hoeveelheid lichtinval als maatstaf te nemen

- Een functionele indeling van de woonkamer moet mogelijk zijn, dus het beste is de afstand van bank tot TV minimaal 2 / 2,5 meter en maximaal 5 meter.

Verblijfsgebieden (slaapkamers, woonkamer, keuken e.d.) moeten een glasoppervlak hebben van minimaal 10% van het vloeroppervlak van dezelfde ruimte.

het raam moet 2,5 m breed worden en 1,5 m hoog

1. Vanuit een woonkamer moet men horizontaal naar buiten kunnen kijken.
2. Een woonkamer heeft altijd minstens 2 m2 daglichttoetreding

Ik kan hier geen regel voor verzinnen.

Een raam in buitengevel van de woonkamer moet een oppervlakte hebben van tenmiste 1,5 m2

woonkamer moet minimaal een bepaald vastgesteld raam oppervlak hebben grenzend aan de buitenruimte.

positioneer tv scherm zodanig dat er Weinig of geen lichtinval op het scherm valt.

### Small Toilet

Er is niet voldoende gebruiksruimte.

De gebruiksruimte van de objecten overlappen elkaar.

Voor een wasbak dient een ruimte van 600mm gereserveerd te zijn om te kunnen staan.

Toiletruimte moet ruimere afmetingen hebben.

Een toiletruimte moet minstens 1,2 meter lang zijn.

WC-ruimte moet minstens 3 vierkante meter zijn. Om zo voldoende bewegingsruimte te hebben

Ruimte rondom wastafel mag niet minder zijn dan de halve meter zodat men comfortabel de handen kan wassen

de loopruimte moet minimaal 60 cm zijn

Toilet Moet groter dan 2 m2.
WC-fonteintje moet bereikbaar terwijl men er recht voor staat.

Toiletten moeten minstens 1,5m² groot zijn en 80cm aanzet ruimte te hebben om fatsoenlijk te kunnen zitten.

Binnen 1 meter afstand van de deur mag geen meubilair, apparatuur en dergelijk zich bevinden.

Ruimte voor knieen is wenselijk

Staan ruimte ook

Toilet dient tenminste 1,5m² groot te zijn.

toilet ruimte minimaal 1,20 bij 0,90m

- Een toilet moet minimaal 1.20x2.40 breed zijn
- WC moet minimaal 1 bij 1 meter zijn (oid, zie bouwnormen?)
- Er dient ruimte vrijgehouden te worden voor de wastafel om te staan
- Er dient een ruimte vrijgehouden te worden voor de WC om te kunnen zitten 20-30cm?

toiletruimte moet minimaal 1,2m diep zijn

afmetingen toilet minimaal 1200 x 900

Een toilet heeft een minimale afmeting van 0,9 x 1,2 meter

toiletruimte 1,2 m diep

leidingschacht achter toiletpot plaatsen

het vloeroppervlak moet 1,5 keer zo groot worden

1. Een toilet heeft een plaatsingsruimte en een gebruiksruimte.
2. Een toilet staat met de achterzijde gecentreerd in de plaatsingsruimte.
3. De plaatsingsruimte is minimaal 0.6 x 0.6 meter.
4. De aangrenzende gebruiksruimte is minimaal 0.3 x 0.6 meter.
5. Een fonteintje heeft een strook van 0.3 meter nodig naast de plaatsings ruimte van het toilet.

De lengte en de breedte van de toiletruimte moet minimaal 1,5 zijn?

De afstand van de voorkant van het toilet tot de deur moet minimaal 0.80 meter zijn.

De deur van het toiletruimte moet op slot kunnen.

De vloeroppervlakte van een toilet moet minimaal 0,9 m² bedragen bij een naar buiten draaiende deur.

Bij een naar binnen draaiende toiletdeur mag de deur de toiletpot niet raken.

toilet moet aan minimale afmetingen voldoen, en

er moet minimaal een vastgestelde bewegingsruimte zijn
**Wrong material façade**

| Stenen van een buitengevel in een wijk moeten van dezelfde afmeting zijn. |
| De uitstraling van een rijtjeswoning moet hetzelfde zijn als de andere woningen in dat rijtje |
| Alle huizen in een straat/wijk moeten bijdragen aan een correct straatbeeld |
| Als een algemeen uiterlijk wordt toegepast, mag geen woning daarvan afwijken |
| De materialisatie van de gevel in een bouwblok dient uniform te zijn. E.g: de gevel dient in hetzelfde materiaal opgebouwd te worden als de aangrenzende gevels in hetzelfde bouwblok. |
| Alle aangrenzende gevels moeten van hetzelfde materiaal zijn gemaakt, zodat er geen ‘uitspringende gevels’ te zien zijn. |
| De gevelbekleding dient van .. materiaal te zijn |
| woningen in een bouwblok moeten uit eenzeld materiaal worden vervaardigd. |
| Het materiaalkieze en dus het gevelbeeld mag in een bouwblok per huis niet afwijken |
| bij een rijtjeswijk mag maar een type gevelbekleding gebruikt worden |
| Onwenselijk??? |
| bij bouw en woning toezicht dienen kleur veranderingen aan de straat zijde aangegeven te worden. |
| - Beslissingen aan het exterieur van een huis dienen gemaakt te worden in overeenstemming met esthetische overwegingen van een architect/stedenbouwkundige/gemeentemedewerker |
| gevelbekleding mag niet veranderen van textuur en moet binnen RAL... en ... kleuren blijven |
| - De gevelafwerking mag in kleur / reflectiviteit niet teveel afwijken van andere woningen, misschien nog wel verder gaan en gewoon hetzelfde materiaal eisen |
| Woning dient te voldoen aan de eisen van welstand |
| De gevelmaterialen moeten overeenstemmen met de omliggende gebouwen |
| gevelmateriaal dient baksteen te zijn? |
| materiaal buitenblad moet overal gelijk zijn |
| overal dezelfde steen gebruiken |
| Aan het uiterlijk van de woning mogen geen veranderingen worden aangebracht |
| Het materiaal van een gevel moet gelijk zijn aan het materiaal van de gevel van een andere woning. |
| de gevelbekleding van de woning mag niet afwijken van de gevelbekleding van de naastliggende woningen. |
| een straat met rijtjeshuizen moet een zelfde uitstraling hebben |
### House too high

| De hoogte van de woning moet hetzelfde zijn als de 2 woningen die ernaast gesitueerd zijn. |
| De woninge moet even veel verdiepingen hebben als de 2 woningen die ernaast gesitueerd zijn. |
| De hoogte van alle woningen in een rijtje moet hetzelfde zijn. Daken van aaneensluitende huizen mogen niet meer dan een halve meter in hoogte verschillen. |
| De maximale bouwhoogte in deze straat is 9 meter. |
| Omringende gebouwen moeten bijna hetzelfde zijn. Alleen als er een duidelijke grens is hoeft dat niet. |
| De maximale hoogte van het huis is 14 meter. |
| De omlijning binnen een bouwblok, met name de hoogte, mag per woning niet afwijken. |
| Rijtjesbouw moet op de zelfde hoogte orden uitgevoerd. |
| De uniformiteit is saai, verspringingen in kleur, vorm, textuur maakt een groot verschil. Daarnaast. Een uniform bouwblok dat onderhevig is aan 1 accent kan daarnaast wel interessant blijven. |
| Omringende bebouwing niet meer dan 2m verschil met omliggende bebouwing. |
| Niet toelaten van dit soort uitbouwen (gemeente). |
| - Zie voorgaande regel |
| Goothoogte 7.05m |
| Nokhooogte 9.65m |
| Schuinte dak (volgt hieruit) 45graden |
| Prima niks mis mee (denk ik) |
| Woningen maximaal 3 lagen met kap |
| De dakrand moet op 6m hoogte zitten. De helling van het dak dient 60 graden te zijn. |
| Een huis heeft maximaal 3 verdiepingen |
| Een huis heeft minimaal en maximaal 5 verdiepingen |
| De inhoud (bouwmassa) van de woning mag niet worden gewijzigd. |
| Er zijn twee etages. |
| De woning mag in de hoogte worden uitgebreid met maximaal 1 verdieping. |
| De verdiepingshoogte is hierbij hetzelfde (met een afwijking van max. 10%) als de bestaande verdiepingshoogte van de bovenste versiering. |
| In een straat rijtjeswoningen dient er geen dissonant in de vormgeving voor te komen. |
| De hoogte van de rijtjeswoningen moet identiek zijn. |
## Unopenable Door

<table>
<thead>
<tr>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>De ruimte voor een deur moet minimaal even groot zijn als de deur open kan.</td>
</tr>
<tr>
<td>Een deur moet ten minste 90 graden open kunnen draaien.</td>
</tr>
<tr>
<td>Er moet zich een ruimte van minimaal 2 meter zitten tussen de deur en de trap.</td>
</tr>
<tr>
<td>Een openstaande deur mag het gangpad niet blokkeren.</td>
</tr>
<tr>
<td>De deur moet vrij kunnen draaien.</td>
</tr>
<tr>
<td>De deur moet naar de andere kant open gaan.</td>
</tr>
<tr>
<td>Een deur uitkomend op een gang moet volledig open en dicht kunnen, zonder obstakels.</td>
</tr>
<tr>
<td>de deur moet in zijn geheel helemaal openkunnen voor een goede doorgang.</td>
</tr>
<tr>
<td>opening naar de andere kant voorkomt dit probleem.</td>
</tr>
<tr>
<td>ook moet de loop naast de trap breder zijn</td>
</tr>
<tr>
<td>Deuren moeten tenminste een ruimte van 900mm bij 900mm voor er achter de deuropening hebben.</td>
</tr>
<tr>
<td>Tussen niet-verplaatsbare topologische objecten (gefixeerde objecten) dient een minimale loopruimte van 800 mm aanwezig te zijn. Gefixeerde objecten met daarin scharnierende objecten (e.g. deuren, tuimelkozijnen) mogen geen enkel obstakel in hun pad hebben. De productspecificaties bepalen dan de minimaal benodigde loopruimte. In dit geval is de uitzwenking van de deur bepalend voor de minimale loopruimte (tussen 1200-1300 mm).</td>
</tr>
<tr>
<td>Gangpaden dienen minstens 1,5 meter breed te zijn</td>
</tr>
<tr>
<td>Geen objecten die het openen of sluiten van een deur kunnen hinderen voor een deur plaatsen</td>
</tr>
<tr>
<td>De deur moet op zijn minst 90 graden open kunnen, wanneer deze op een gang uitkomt het liefst 180 graden met een misschien wel een wandelpad speling. Een deur niet laten eindigen bij het begin van de trap</td>
</tr>
<tr>
<td>Deuren moeten over de volle 180 graden vrij kunnen draaien.</td>
</tr>
<tr>
<td>Deuren mogen geopend de doorstroming van een gang of hal niet blokkeren.</td>
</tr>
<tr>
<td>Een deur moet aan beide kanten minimaal 2m2 oppervlakte ruimte hebben.</td>
</tr>
<tr>
<td>Deur moet altijd open kunnen draaien.</td>
</tr>
<tr>
<td>Verkeersruimte moet minimaal 1,0 meter zijn.</td>
</tr>
<tr>
<td>alle deuren dienen minimaal 90 graden open te kunnen.</td>
</tr>
<tr>
<td>- Een deur moet open kunnen slaan, dus er dient een ruimte gelijk aan de breedte van de deur loodrecht op de plaats van het scharnier vrijgehouden te worden. Wellicht geen rechthoekig vlak maar een cirkel.</td>
</tr>
<tr>
<td>- Een doorgang in een woning (het gedeelte van de gang waar gelopen kan worden, dus de oppervlakte van de gang - de oppervlakte van de trap waar deze lager dan 2 meter is) dient minimaal ~80cm te zijn.</td>
</tr>
</tbody>
</table>
afstand tussen trap en deur minimaal de deurbreedte
ruimte waarin de deur draait moet vrij zijn
minimaal een doorloop van 90 cm achter deuren of deuren naar de andere kant op laten draaien
de hal moet 3 keer zo breed worden
Een doorgang heeft altijd een vrije ruimte van 0,9 meter
Een deur moet een slot hebben.
De breedte van de gang min de breedte van de trap moet meer dan 1 meter zijn.
Elke deur moet kunnen worden opgedraaid tot een hoek van minimaal 90 graden.
de deur naar de trappenhal moet naar buiten toe opengaan.
de deur naar de trappenhal moet zodanig geplaatst zijn dat deze helemaal open kan gaan.

**Wrong colour dormer**

| De uitsplitsing van de dakkapellen binnen een woonblok dienen dezelfde kleur te zijn. |
| De dakkapel moet dezelfde kleur hebben als de dakkapellen in de rest van het woningblok. |
| alle dakkapels moeten dezelfde kleur hebben |
| Dakkappelen moet in het wit uitgevoerd worden. |
| Als er een algemeen uiterlijk is gekozen, mogen aanpassingen aan de woning alleen worden toegepast als deze binnen het algemeen uiterlijk passen. |
| Gevels die aan elkaar grenzen moeten een eenheid vormen met de anderen. Of iedereen moet een andere kleur nemen, of allemaal dezelfde kiezen. Maak hierover samen een overeenkomst |
| de kleuren van uw woning dienen neutraal te zijn. |
| de woningen in een bouwblok moeten binnen een algemene uitsplitsing passen |
| de kleur van de extra bebouwing moet schriftelijk vast gesteld worden zodat de eenheid van het bouwblok behouden blijft |
| ik vind dit geen onwenselijke situatie, op deze manier wordt een eigen identiteit aan een woning gegeven |
| Kleurgebruik van nieuwbouw moet aansluiten bij bestaande situatie. |
| Dakkapellen mogen niet naar eigen kleur gekozen worden |
| Dakramen dienen in het straatbeeld te passen, wanneer mensen dit maken moeten ze hier bij de gemeente toestemming te moeten vragen |
| Onwenselijk?? |
| Woning moet goedgekeurd worden door Welstandscommissie. |
Dakkapel dient aangevraagd te worden bij bouw en woning toezicht, dit geldt ook voor kleur en materiaal veranderingen aan de straatzijde.

- Zie voorgaande regel

Dakkapel in zelfde lijnen als bestaande (buur) dakkapellen, dwz hoogte boei-boord en hoogte onderkozijn

RAL kleuren zelfde als bestaande dakkapel

Indien een dakkapel gewenst is moet de standaard dakkapel voor de rij waar het huis in staat gebruikt worden.

de dakkapel moet 2 keer zo groot worden

de lijsten van elke dakkapel moet wit zijn

Alle aan- of bij-bouwen moeten qua grootte, vorm, struktuur en kleur gelijk zijn aan de reeds bestaande.

De kleur van een dakkapel moet wit zijn.

De kleur van een dakkapel mag in het RGB kluerenspectrum niet meer dan 10% afwijken van de kleur van de naasteliggende dakkapelen, indien aanwezig.

Alles woningen dienen gelijke kleurschakering van de dakkapellen te hebben

---

**b.8 SHED TOO HIGH**

Buiten het woonblok mag er geen bebouwing hoger zijn dan de voorgeschreven schuttinghoogte

De hoogte van elk tuinhuisje in een rijtjeswoning moet hetzelfde zijn

Schuren mogen niet hoger dan 3 meter zijn

De schuren mogen niet boven de omheining uitkomen dit haalt uitzicht weg en geeft naast gelgen tuin meer schaduw

Achtertuinen moeten qua hoogte het zelfde blijven als bij de buren.

In een bouwblok met een uniformiteit van opzet, dit impliceert een gelijk oppervlakte van woning, gelijke materialisatie van gevel, gelijke bouwkavels, etc. dienen alle topografische elementen in dimensies op elkaar afgestemd te zijn. Dit wil zeggen: gelijke dimensies in x,y,z richting.

De hoogte van uw schuur mag maximaal 4 meter bedragen

De berging mag niet boven de schutting uit komen

Het beeld moet per woning hetzelfde zijn. m.a.w. er mogen niet of nauwelijks afwijking in de vorm, en dus beeld per woning zijn

Rijtjes bouw moet op de de zelfde hoogte worden uitgevoerd

Er mogen geen objecten in de tuin worden geplaatst waar de buren last van hebben. In dit geval in verband met zonlicht.

1 zwart schaap over de brug, dan de rest ook. DUS. laat ze lekker

Bijgebouwen(zoals schuren) niet hoger dan 1 verdieping
schuurtjes mogen maximaal 4 meter hoog zijn
Niet toelaten door de gemeente
Hogere schutting plaatsen
schuur maximale afmetingen 3x3x2,5m
minimaal 50cm van rooilijn
bijgebouwen mogen niet boven de erfgrens uitsteken
bijgebouwen mogen niet hoger zijn dan 3 meter
maximale hoogte buitenberging is 3000
een schuur moet 1 verdieping hebben
Alle aan- of bij-bouwen moeten qua grootte, vorm, struktuur en kleur gelijk zijn aan de reeds bestaande.
De hoogte van een tuinhuisje mag niet meer dan twee meter zijn.
De hoogte van het hoogste punt van een schuur mag niet meer zijn dan de de bovenkant de de vloer van de eerste verdieping van bijbehorende woning.
de de hoogte van de schuur/berghok moet 2,50m zijn

### B.9 Wrong window orientation

| Alle ramen in een gevel moeten dezelfde orientatie hebben |
| de gevel is een geheel waarin alle raam openingen in het ritme moeten passen |
| Dezelfde ramen in dezelfde gevel, of juiste heel veel verschillende ramen gebruiken |
| Raamopeningen moeten in dezelfde richting georiënteerd zijn, danwel horizontaal danwel verticaal. |
| niks op aan te merken |
| In woningen moet een minimale hoeveelheid licht binnendringen per etmaal |
| geen onwenselijke situatie |
| Op etage's mogen ramen niet tot de vloer worden getrokken wanneer dit niet in de gevel past. |
| Hier is duidelijk te zien dat er een horizontale richting is gekozen voor het gevelspel, |
| Onwenselijk?? |
| Bouw en woning toezicht dient bij bouwaanvraag dit aan te kaarten. |
| geen idee wat hier mis mee is |
| woningen niet met een entree aan de stoep |
| Ramen dienen horizontaal geplaatst te worden. (maw, breedte is > hoogte) |
| het raam moet uit drie delen bestaan |
| het raam moet breder dan hoog zijn |
| **Aan het uiterlijk van de woning mogen geen veranderingen worden aangebracht.** |
| **De de breedte gedeeld door de hoogte moet groter dan één zijn.** |
| **De onderzijde van het raam moet zich bevinden op minimaal 1,2 m van de bovenzijde van de verdiepingsvloer.** |
| **De bovenzijde van het raam moet zich bevinden op maximaal 2,2 m van de bovenzijde van de verdiepingsvloer.** |
| **de “raamstructuur” dient voor de woningen uniform te zijn** |

| **B.10 STAIR TOO STEEP** |
| **Een trap in een woning moet de 45 graden niet overstijgen** |
| **De trap moet onder een hoek staan van 30 graden staan.** |
| **Een trap moet (volgens het bouwbesluit) een acceptabele op- en aantrede hebben.** |
| **optreden mag maximaal 250mm zijn** |
| **Aantrede van de trap dient in verhouding te zijn met 2 x optrede + 1x aantrede = 610 tot 630 mm** |
| **Tussen verschillende bouwverdiepingen (verdieping gedefinieerd als hoogte van vloerniveau t/m plafondniveau + eventuele materiaaldikte (structurele) verdiepingsvloer) waarbij een connectie noodzakelijk is dient een maximaal stijgingspercentage gedefinieerd te worden. Dit uit zich in het feit dat de op- (O) en aantrede (A) maximaal aan elkaar gelijk mogen (O=A=true) zijn maar dat de optrede nooit groter mag zijn dan de aantrede (O>A=false). Aantrede mag groter zijn optrede (A>O=true).** |
| **Matrix:** |
| \( (A>O=true) \rightarrow \text{ACCEPTABEL} \) |
| \( (O=A=true) \rightarrow \text{GRENSTOEVESTAND} \) |
| \( (O>A=false) \rightarrow \text{ONACCEPTABEL} \) |
| **1 keer de aantrede, 2 keer de optrede.** |
| **1 x de aantrede en 2x de optrede = 570 tot 700 deze is hier niet bereikt de trap is te stijl** |
| **De stijlde van een trap moet maximaal .... graden zijn.** |
| **Trappen dienen te voldoen aan de eisen die gesteld worden aan een trap. Zoals de juiste verhouding tussen op en aantreden.** |
| **Aan- en optredes hebben minimale afmetingen** |
| **De woning niet toelaten door de gemeente met behulp van het bouwbesluit aantrede+optrede > 35cm** |
| **waarbij aantrede > 20cm** |
- De trapformule \((2 \times \text{optrede} + 1 \times \text{aantrede}) = 570 - 630\) mm dient van toepassing te zijn op trappen.

<table>
<thead>
<tr>
<th>Trapformule: (2 \times \text{optrede} + 1 \times \text{aantrede} = 570-630)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Een trap voldoet aan de volgende regels:</td>
</tr>
<tr>
<td>een vrije hoogte van minimaal 2,3 meter;</td>
</tr>
<tr>
<td>een breedte van minimaal 0,8 meter;</td>
</tr>
<tr>
<td>overbragt maximaal 4,0 meter zonder tussenplateau;</td>
</tr>
<tr>
<td>een minimum aantrede van 0,22 meter;</td>
</tr>
<tr>
<td>een maximum optrede van 0,185 meter.</td>
</tr>
<tr>
<td>Trappen met een aantrede van 180mm minimaal</td>
</tr>
<tr>
<td>de maximale helling van een trap is 35 graden. 2x aan + 1x op is 63 cm.</td>
</tr>
<tr>
<td>De trap moet 2 keer minder steil zijn</td>
</tr>
<tr>
<td>Een optrede en aantrede van een trap moet de lengte hebben van een normale wandelpas.</td>
</tr>
<tr>
<td>De trap???? moet niet meer dan ??% zijn.</td>
</tr>
<tr>
<td>De maximale helling van de trap is 45 graden.</td>
</tr>
<tr>
<td>de trap moet een vastgestelde helling te hebben.</td>
</tr>
<tr>
<td>de treden van de trap moeten een vastgestelde diepte hebben</td>
</tr>
</tbody>
</table>
Bibliography


Aho, A. & Ullman, J. (1972), The theory of parsing, translation, and compiling, Prentice-Hall, Inc..


Backhouse, R. (1986), Constructie en verificatie van programma’s, Prentice Hall, Inc..


van den Boogaard, W.; Huyghe, G. & Kolpa, J. (1990), Demontabel bouwen, CUR, Zoetermeer.


TNO (2011), iBUILD. http://www.ibuild.nl/
Volkswagen (2011), *Die Gläserne Manufaktur (The Transparent Factory).*
http://www.glaesernemanufaktur.de/gmd.jsp?dok=&lang=&docid=&ap=


Yahoo! News (2010), *Chinese workers build 15-story hotel in just six days.*
List of figures

Figure 1.1 Prototype overview 5
Table 2.1 Customization types according to Pine II 11
Figure 2.2 Dom-ino house 15
Figure 2.3 Rietveld-Schröder house interior 17
Figure 3.1 SMARTcodes constraint highlighting 29
Figure 3.2 SMARTcodes constraint checking 30
Figure 4.1 Process overview 38
Figure 4.2 Constraint checking prototype 40
Figure 4.3 Abstract Syntax Tree for the example expressions 40
Figure 4.4 Prototype BIM class diagram 41
Table 4.5 Constraint types implemented in the prototype 42
Figure 4.6 Constraints used in the test case 43
Figure 4.7 Design 1 44
Figure 4.8 Design 2 44
Figure 4.9 Design 3 44
Figure 4.10 Design 4 44
Figure 4.11 Yahoo Pipes 48
Figure 4.12 Lego Mindstorms NXT 49
Figure 4.13 Block-structured sentence 49
Figure 4.14 The CUBE programming language 50
Table 4.15 Example translation of building code constraints 53
Figure 4.16 Prototype main screen 54
Figure 4.17 Puzzle piece editor 54
Figure 4.18 Autocompletion example 54
Figure 5.1 Algorithm steps 69
Table 5.2 Word types 70
Table 5.3 Preprocessing rules 72
Figure 5.4 Tree construction example 73
Figure 5.5 Example syntax tree 74
Figure 5.6 Final syntax tree 76
Figure 5.7 Prototype interface 78
Figure 5.8 Adding a word definition 79

Table 6.1 Correctly interpreted constraints 83
Table 6.2 Constraints requiring minor modifications 84
Table 6.3 Constraints that are not yet handled 85
Table 6.4 Constraints lacking a desired encoding 86
Figure 6.5 Case 1: gap in wall 87
Figure 6.6 Case 2: small window 87
Figure 6.7 Case 3: small toilet 87
Figure 6.8 Case 4: wrong material 87
Figure 6.9 Case 5: house too high 87
Figure 6.10 Case 6: unopenable door 87
Figure 6.11 Case 7: wrong colour 87
Figure 6.12 Case 8: shed too high 87
Figure 6.13 Case 9: wrong windows 88
Figure 6.14 Case 10: stair too steep 88
Figure 6.15 Constraint categories 89
Figure 6.16 Valid constraint categories 89

Figure 7.1 SWOP class hierarchy 100
Figure 7.2 SWOP operators 100

133
Subject index

AST 39·41, 74, 76, 79, 88, 97, 99-101
BIM 4, 6, 11, 19·21, 29, 31, 41, 56, 88, 90
building codes
  4, 6, 20, 28-29, 32-33, 52·53, 57,
  62, 77, 82-83, 86, 88, 90, 94·95,
  98, 101-102
CAD 4, 10, 20-21, 26, 28, 31, 34, 58, 98
CFG 63
computability 34, 98
constraint
  3·7, 19, 21·23, 25·35, 37·47, 49·53,
  55·59, 61·62, 67·71, 74·75, 77·79,
  82·86, 88·90, 94·103
constraint-based design
  5, 7, 28, 38·39, 58·59
constraint checking
  4·7, 30·31, 34·35, 37·40, 42,
  44·45, 57, 86, 88, 96, 100·101
constraint entry
  4·7, 38, 45·47, 49·51, 58·59, 62,
  82
constraint satisfaction 22
constraint solving 4, 6, 33·35
decidability 34, 98
design verification 19·20, 26
DSL 46·47, 49
feature based modelling 28
IFC 20·21, 29, 56·57, 99·101
lambda function 39·40, 55·56
list comprehension 51·52
machine learning 66, 79
mass customization
  2, 4, 6·7, 9·21, 23, 26, 35, 82, 90,
  94, 97·98, 101
mass production 2, 10, 15·17, 23
modular customization 12·14
natural language processing
  6·7, 47, 62, 65·67, 82
parametric design 19, 27·28
parse tree 64, 68, 88
parsing 6, 55, 62·69, 77·79, 96, 99
part-of-speech tagging 65
PCFG 66, 97
predicate 52
production rule 63, 66, 75·77, 79
rapid prototyping 5, 38
reasoning 3·4
sanitizing 74, 77
semantic dependency 7
solution space 6, 33·34, 98
systems engineering 31, 38
token 41, 63·65, 70·71, 73·77
transduction 67·68
user-driven design 18
visual constraint entry 49, 51
visual programming 5, 7, 49·50, 94
The following photos have been used under the Creative Commons license:

“Warm Sand” by Mr eNil
http://www.flickr.com/photos/enil/462690449/

“Can’t choose” by LEO-ONE.
http://www.flickr.com/photos/leo-one/4480337116/

“River Bure” by lordspudz
http://www.flickr.com/photos/lordspudz/2826017422/

“Pen and Paper” by Ernst Vikne
http://www.flickr.com/photos/iboy/3122612737/

“The Tree (Explore Highest #261)” by Paul Swann
http://www.flickr.com/photos/pdjsphotos/4065496314/

“Test tubes and other recipients in chemistry lab” by Horia Varlan
http://www.flickr.com/photos/horiavarlan/4273968004/

“Stop Sign” by Phil Gilbert
http://www.flickr.com/photos/z6p6tist6/501709581/

“Stack of Books” by Ludovic Poitou
http://www.flickr.com/photos/ludovicpoitou/5022348921/
Summary

Constraint specification in architecture: A user-oriented approach for mass customization

The last several decades, particularly those after the end of World War II, have seen an increasing industrialization of the housing industry. This was partially driven by the large demand for new houses that resulted from the baby boom and the destruction caused by the war. By adopting mass production the building industry was able to meet the demand. A downside of this development is a decrease in the amount of input buyers have in the design of their house. Due to the economies of scale that are the basis of mass production, there is a tendency to make houses identical as much as possible, reducing the price at the cost of flexibility. This decrease in consumer choice is common when an industry shifts to mass production, as illustrated by Henry Ford’s famous quote about the Model T Ford: “Any customer can have a car painted any colour that he wants — so long as it is black.”

Currently, people buying a new house typically have to either accept the architect’s design, or they are offered a limited amount of alternatives to choose between. The limited number of these choices and the fact that in most cases they are not based on input of the buyers, however, mean that the alternatives do not match the buyers’ needs as well as they might. This in contrast to many other industries, such as the food, fashion and automotive industries where customers are presented with a wide range of alternatives. These industries prove that mass production does not necessarily result in less flexibility. This mixture of mass production and customization is called mass customization. In chapter 2, mass customization is explored in more detail by looking at the various categorizations that have been made of mass customization. Different subcategories can be identified based on criteria such as the degree of standardization or the extent to which the client is involved in the design process. Subsequently, the use of mass customization in the building
industries is examined. Although not particularly common, there have been studies into this phenomenon, among others by famous architects such as Le Corbusier and John Habraken. In order to make the adoption of mass customization in the building industry more feasible from a design perspective, it is preferable to use a design representation that allows for a more high-level view of the design than the traditional collection of two-dimensional lines. The ability to refer to building elements such as walls and dormers as a whole makes modifications easier to perform and greatly simplifies automated model verification. For this reason, a discussion of Building Information Models in general and IFC, as the de facto standard for BIMs in the architecture domain, in particular is included in this chapter.

As mentioned, BIMs facilitate automated model verification. This is particularly useful in relation to mass customization, since part of the design process will now be performed by non-expert users. It is to be expected that the designs made by the buyers will not be in full accordance with either the building codes or the architect’s intentions. Checking the designs for problems can be done manually, but this is a laborious and error-prone process. It would be preferable to be able to automate this checking process and ideally to perform it in real time while the buyer is modifying the design. This way the architect can be guaranteed that the resulting designs have no trivial problems, freeing him up to look at more complicated criteria that are difficult or impossible to automate, such as aesthetics. Automated model verification is performed by establishing constraints that the design must satisfy. In chapter 3 the use of constraints in other industries, such as software and electrical engineering, is discussed. So far, use of constraints in the building industry has been limited. Some projects in which they have been used are listed and some possible explanations for this slow adoption are presented. Constraints can be used to either check existing designs or to automatically generate new ones. In this thesis, only the former is explored.

The goal in this research process was to examine the use of constraint checking in the building industry with the aim of supporting mass customization and, more specifically, to develop a method for specifying these constraints. To this purpose, several prototypes were developed to explore various ideas. Two of the prototypes are described in chapter 4. The first of these two prototypes was used to test the viability of performing constraint checking on building models. No fundamental technical problems were discovered. The prototype fulfilled the intended function of accepting allowed designs and rejecting designs that violated any of the constraints. Having concluded that constraint checking in the building industry is possible, the next step was to devise a method for entering constraints into the system. Since constraints such as building codes are currently predominantly stored in natural language, allowing architects to specify the constraints in natural lan-
language was one of the more obvious approaches. This approach, however, presents a significant technical challenge, since interpreting natural language is very complicated. After an analysis of possible alternatives (ranging from programming languages to visual constraint entry), the first attempt at a constraint entry system was based on the same basic idea as natural language, but with some modifications to reduce the technical complexity. Rather than allowing free-form natural language input, sentences are instead constructed using puzzle pieces. Each puzzle piece contains a single word or short sentence fragment, which are linked together to form the full constraint. This has the effect of limiting the range of grammars that can be used, making the system fairly simple to implement. User testing conducted on architects revealed that although the prototype performed reasonably well in terms of usability, the process of constructing the sentences was deemed to be too laborious to be used in practice.

After rejecting the puzzle piece-based method, the decision was made to see if it was possible to approach natural language parsing to a sufficient degree to be used for constraint entry. This method allows architects to simply type in the constraint, which solves the problem of the entry method being too laborious. In chapter 5, an architectural language parsing algorithm is described where the syntax tree of the constraint is modified using a very simple grammar in order to fix problems such as redundant or omitted words. The advantages of this method are that it is very flexible with regards to the grammar used, that it doesn’t require a full grammar of the language being used, that little training is required in order to use it and that it can theoretically be used in multiple (Western European) languages. Additionally, the general nature of the algorithm means that it is not applicable solely to the building industry, but to any industry in which constraints need to be specified.

To test the algorithm, two tests were conducted, one testing the system’s performance on legal constraints such as building codes and one testing the performance on design constraints, i.e. the constraints written by architects. Chapter 6 describes the results of these two tests. The legal constraints were drawn from the building codes regarding dormers of the municipality of Rotterdam. For the design constraints, a user test was conducted in which architecture students were presented with building designs that were in some way flawed and asked to provide objective constraints to prevent these problems in the future. This was done in order to get a representative sample of the type of language used by architects. In both cases, the desired resulting syntax tree was specified for each constraint to test how many constraints were interpreted correctly by the system. Despite being a very early prototype, the system is already able to correctly interpret about half of both the legal and design constraints. Chapter 7, finally, provides discussion and identifies potential avenues for future research.
Remco Niemeijer was born on the 3rd of February 1984 in Dordrecht, The Netherlands. In 2001 he started his Bachelor of Science (BSc.) in Architecture, Building and Planning at the Eindhoven University of Technology. Here, his interest in Computer Aided Design was sparked by a course on 3D Studio. After achieving his Bachelor’s degree in 2004, he continued his studies at the university with a Master of Science (MSc.) in Design and Decision Support Systems. In his Master studies, he became increasingly interested in the field of mass customization, which was the topic of several projects as well as an internship at the architecture firm BBVH, located in Rotterdam. After submitting his proposal for his Master thesis, he was offered the opportunity to become a PhD Student at the Design Systems group by prof. dr. ir. Bauke de Vries. This project was conducted between 2007 and 2011.

His research interests are in the fields of mass customization, building information systems and computer aided design. His teaching activities comprise graduate courses in computer supported design and visualization.