Check-mate: automatic constraint checking of IFC models

R.A. Niemeijer, B. de Vries & J. Beetz
Eindhoven University of Technology, Eindhoven, Netherlands

ABSTRACT: Building Information Models (BIMs) allow for computationally checking whether a building design satisfies all the building codes, requirements, etc. if constraints are included in the model. One application for this is mass customization in the housing sector, when clients modify the design without help from the architect. This paper describes the technical aspects of checking constraints on a building model. Specifically, we look at the feasibility of checking constraints on an IFC model by creating a prototype in which constraints can be entered and checked on an imported IFC model. Conclusions are drawn on the suitability of the IFC model and how IFC can be extended or adjusted to support constraint checking.

1 INTRODUCTION

Adoption of CAD (Computer Aided Design) in the building industry has so far focused mainly on replacing drawing lines on paper with drawing lines on a computer. Although this is an improvement – making changes is now much easier – it uses only a fraction of the potential of CAD. Checking whether a building complies with building codes and other legislation, for instance, is still left up to the architects and building committees. Adding semantic information to the current CAD models opens up (among others) the possibility of delegating this task to the computer.

While designing the building, the architect could get immediate feedback on whether or not his design violates any of the constraints imposed by building codes. Although this would be useful in its own right, such an approach offers even more potential for the field of mass customization (van den Thillart 2004; Huang & Krawczyk 2007). The idea of mass customization is to allow client input in the product design without losing the benefits of mass production. The limited adoption of mass customization in the building industry has so far focused on providing the client with a few alternatives to choose from. In this new approach, however, the client would have significantly more freedom without requiring a corresponding increase in effort from the architect.

In this paper we describe a method of automatically checking constraints on IFC (Industry Foundation Classes) files.

2 SYSTEM DESCRIPTION

The prototype discussed in this paper is part of a larger system that aims to make mass customization easier for architects. The basic philosophy is that the architect makes a design as usual, but in addition to that he also specifies the requirements he wants the final, customized, design to meet. For instance: “this wall must contain at least three windows” or “the façade material must be brick”. These rules are known as constraints (Kelleners 1999; Strömberg 2006; Donath & Böhme 2007). The design is then presented to the buyers, who are free to make changes, as long as these changes are not prohibited by the rules of the architect, building codes, laws, etc. Whenever they make a change that violates any of the constraints, they are presented with a warning. The final design is not accepted until all such warnings have been resolved.

Ideally the architect’s design would be made in a CAD package that allows both real-time constraint checking and constraint creation. That way the architect can get the most benefit from this approach. Creating a full CAD package, however, is far from trivial. In the meantime we therefore focus on checking constraints on models created in existing CAD packages. In order to make the approach as application-agnostic as possible we focus on the international IFC standard, which is the most widely supported file format among CAD applications that preserves semantic information (i.e. information other than just geometry).

3 RELATED RESEARCH

The idea of using constraints to check designs has been accepted in the mechanical engineering industry for many years (Anderl & Mendgen 1996; Gross 1996; Bettig & Shah 2003). In the building industry, however, it has yet to be widely adopted. The two closest analogues to our proposed system are probably the commercial CAD package Revit (Strömberg 2006) and the SMARTcodes project (Wix et al. 2008), but there are also several important differences. Revit – and indeed most research on constraints (Martini 1995; Eggink et al. 2001; Hoffman & Kim 2001; Belblidia & Alby 2003; Nassara et al. 2003) – focuses only on geometrical constraints. Our system also considers non-geometrical constraints, such as price, color and heat transmission. And whereas SMARTcodes focuses exclusively on building codes and laws, we also include constraints from other domains, such as architecture, constructive engineering, building physics, common sense, etc.

Additionally, little attention has so far been given to the method of specifying constraints. We want to provide a simple method for architects to enter these constraints. One of the examples of a system that tries to facilitate constraint specification is the SMARTcodes Builder of the SMARTcodes project (AEC 3 2009). In this program, constraints are entered by highlighting parts of regulations text with one of four colors. These four colors represent four functions (somewhat similar to the different puzzle piece types in our system), which are used to convert to convert the highlighted regulations text to computer-checkable constraints. A screenshot of this application is shown in Figure 1. Although the underlying translation of text to constraints appears very difficult from a technical standpoint, given the apparent freeform nature of the text, this is one option to consider for a future system.

4 CONSTRAINTS

A constraint is an assertion about building elements, such as “every window must be less than 1200 mm high”. A constraint is effectively a function that takes as its argument a building element and returns true (the element satisfies the constraint) or false (it does not). Checking whether a certain design satisfies all the constraints is therefore a matter of applying all constraints to all elements in the design, and checking whether they all return true. Some constraints, such as a constraint on the distance between two elements, may take multiple building elements as arguments.

Constraints are comparable to universal quantifiers in the field of mathematics. For instance, the quantification

$$\forall x \in \mathbb{N}(x \geq 0)$$

means “for every x in the set of natural numbers, x is more than or equal to 0”. If we rephrase our example constraint to “for every window, the height of that window must be at least 1200 mm”, the similarity

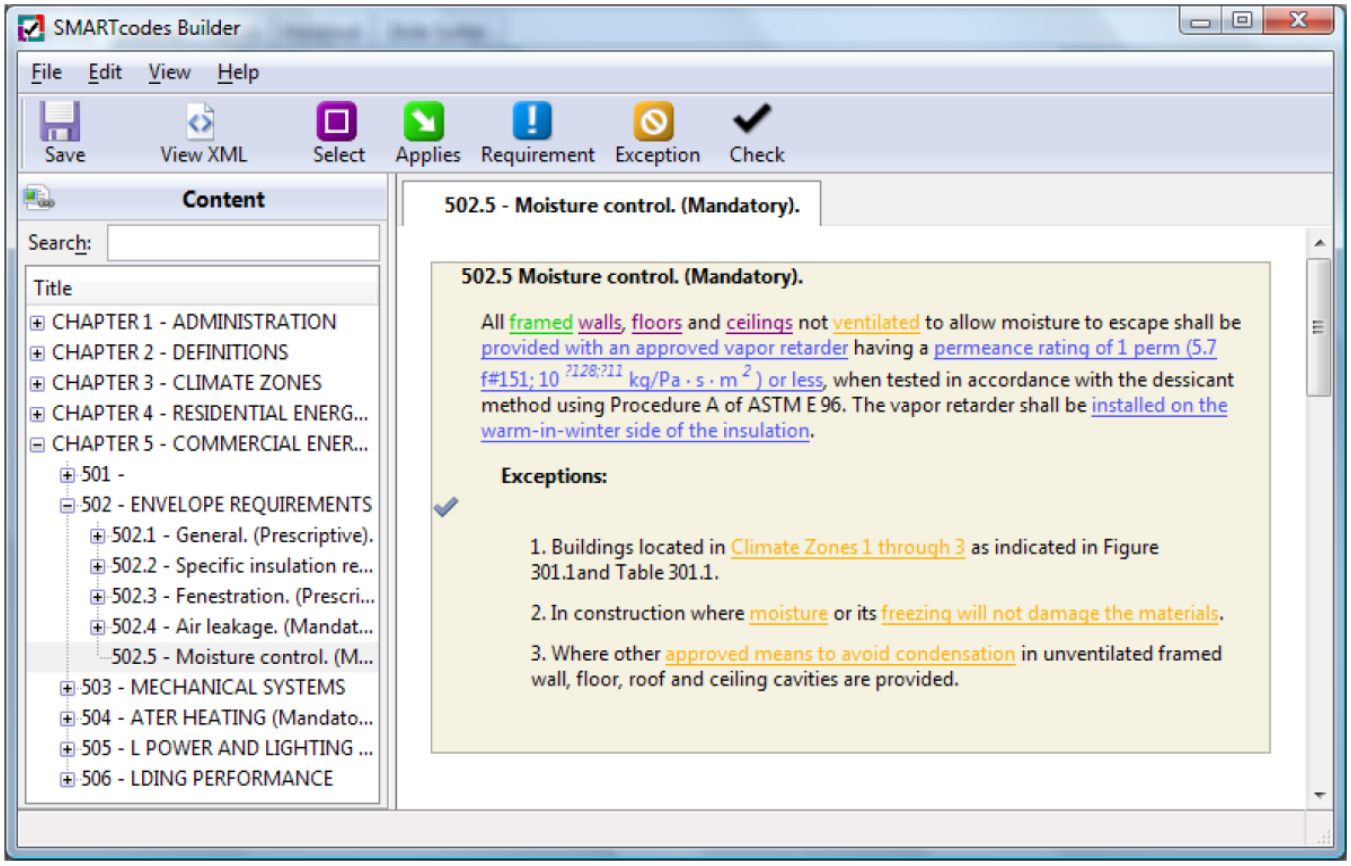

Figure 1. SMARTcodes Builder (AEC 3 2009)

between the two is even easier to see.

4.1 *Limitations of Constraints*

Although a lot of building codes can be checked by the computer in this way, there are exceptions. As an example, the constraint "The quality of the new dormer must at least be equal to that of the existing surrounding buildings" could in theory be entered, but since it is not clearly defined it cannot be checked by the computer. After all, what is meant by quality? Technical quality? Aesthetics? Something as subjective as aesthetical quality cannot objectively be assessed. A solution to this is to add a type of constraint that is not evaluated by the computer, but instead poses the question to the user whether or not the constraint has been met. At first glance this might seem to be useless, but it does serve a purpose. Constraints of this type serve as a sort of checklist, so that none of the hundreds of rules the architect has to follow is forgotten. Even the worst-case scenario, where every constraint is a question constraint, would still be an improvement on the current situation, as it prevents overlooked constraints and the subsequent failure costs.

4.2 *Constraint-based design*

It is important to distinguish between constraint-based design and constraint solving. In constraint solving the system tries to automatically find a design that matches all the constraints. In building design this approach is fairly impractical for two reasons: Firstly, the solution space in building design is very large. Even the simplest of houses will have millions of possible solutions, which makes it infeasible to try to solve this on today's computers. Secondly, the client has little, if any, influence on the resulting design, which ignores the philosophy behind mass customization. Constraint-based design, on the other hand, only verifies whether the design that was created by the architect or the client does not violate any of the given constraints; it is a validation rather than a creation tool.

One other thing to note is that the system can only indicate if all the constraints have been satisfied; not whether it is a good design. It should be possible to reach a good technical and functional quality, provided that enough constraints are entered. The aesthetical quality, on the other hand, will remain an entirely human judgment. Some aspects, such as symmetry or color schemes, could be enforced by constraints, but that will check taste rather than objective beauty. We therefore choose to focus on the technical side of the constraint system, and consequently the functional quality of the design, rather than on value judgments of the design.

5 BUILDING INFORMATION MODELS

Checking constraints on a building requires a representation of the building that carries the needed information. The traditional way of storing a building design as a collection of lines – i.e. a direct replacement of paper drawings – is insufficient for this purpose. Instead, the building should be stored in a way that preserves the semantic information. This can be achieved by using a so-called Building Information Model, or BIM. In a BIM the design is stored as a collection of objects with associated properties (e.g. a wall that has a length, height and material) instead of a collection of lines. This building element based approach makes it possible to check constraints, since all the required data is present.

5.1 *IFC*

In the last few years, building information modeling has become popular in the building industry, and as a result of that there is now a wealth of different building information models, with most CAD packages having an internal BIM. Unfortunately these BIMs are not compatible; they all store different information and save to proprietary formats, making data exchange very difficult. As a solution for this, the Industry Foundation Classes (IFC) were created. 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 most 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 is an open standard, as opposed to the mostly closed standards of the CAD software manufacturers, making it easier to implement.

5.2 *Constraints in IFC*

The IFC schema is defined in a language called EXPRESS. This language already supports defining constraints. As a simple example, the IfcPositiveLengthMeasure is defined as follows (IAI 2009):

```
TYPE IfcPositiveLengthMeasure =
    IfcLengthMeasure;
  WHERE
    WR1 : SELF > 0.;
END_TYPE;
```

The where clause restricts the values of this type to positive ones. Although this appears to be relatively similar to what we need, we unfortunately cannot use it, since these constraints apply only to the schema level. A change in IfcWall, for example, would affect every wall in the design. What we need

are constraints on the instance level, to affect only specific walls.

6 PROTOTYPE

The goal of the prototype is to import a design made in a commercial CAD package that was exported as an IFC file and check constraints on it.

6.1 *Constraint entry*

The constraints that the system checks need to be entered into the system at some point. Since this will be done mostly by architects, the input method has to be easy for them to work with. The method we chose was to create the constraints by using puzzle pieces. This is effectively a Domain Specific Language (DSL) for constraints (Spinellis 1999). Each puzzle piece contains one or more words. By linking these puzzle pieces together you create the sentences that make up the constraints. The reason for choosing this approach was that the resulting rules are grammatically correct English (which makes it easier for people to understand) while limiting the grammar that can be used, which makes it easy for the computer to understand as well. Ideally it would be possible to enter the constraints in natural language, as this requires no learning on the part of the user, but unfortunately this is not yet technically feasible due to the difficulty of parsing natural language.

The left side of Figure 2 shows an example constraint. Every constraint is divided into four sections for the sake of readability and ease of entry. The first section specifies which element types the constraint applies to, such as walls or rooms. In the second section definitions can be created. Originally inspired by clauses from legal contracts such as "…Mr. Henry Woolworth-Kensington, hereafter to be referred to as the buyer…", definitions can be compared to constants in programming. They serve to refer to a complex concept by an easy name. In the third section we can make a further refinement of the selection of elements given in the first section. In the example in Figure 3 we narrow the initial selection of all walls down to those with an area of over 20 m^2. The final section contains the actual constraints that the selected elements must adhere to.

As mentioned earlier, each of the sentences from Figure 3 is created by linking together puzzle pieces. Figure 2 shows the associated puzzle pieces for the last sentence from Figure 3. 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.

Testing this prototype on architects revealed that although not very complicated to learn, this method is rather laborious. For use in practice it will have to be further refined or replaced altogether. One suggested improvement was that over time the architect builds up a library of constraints so that he does not have to start from scratch for each project.

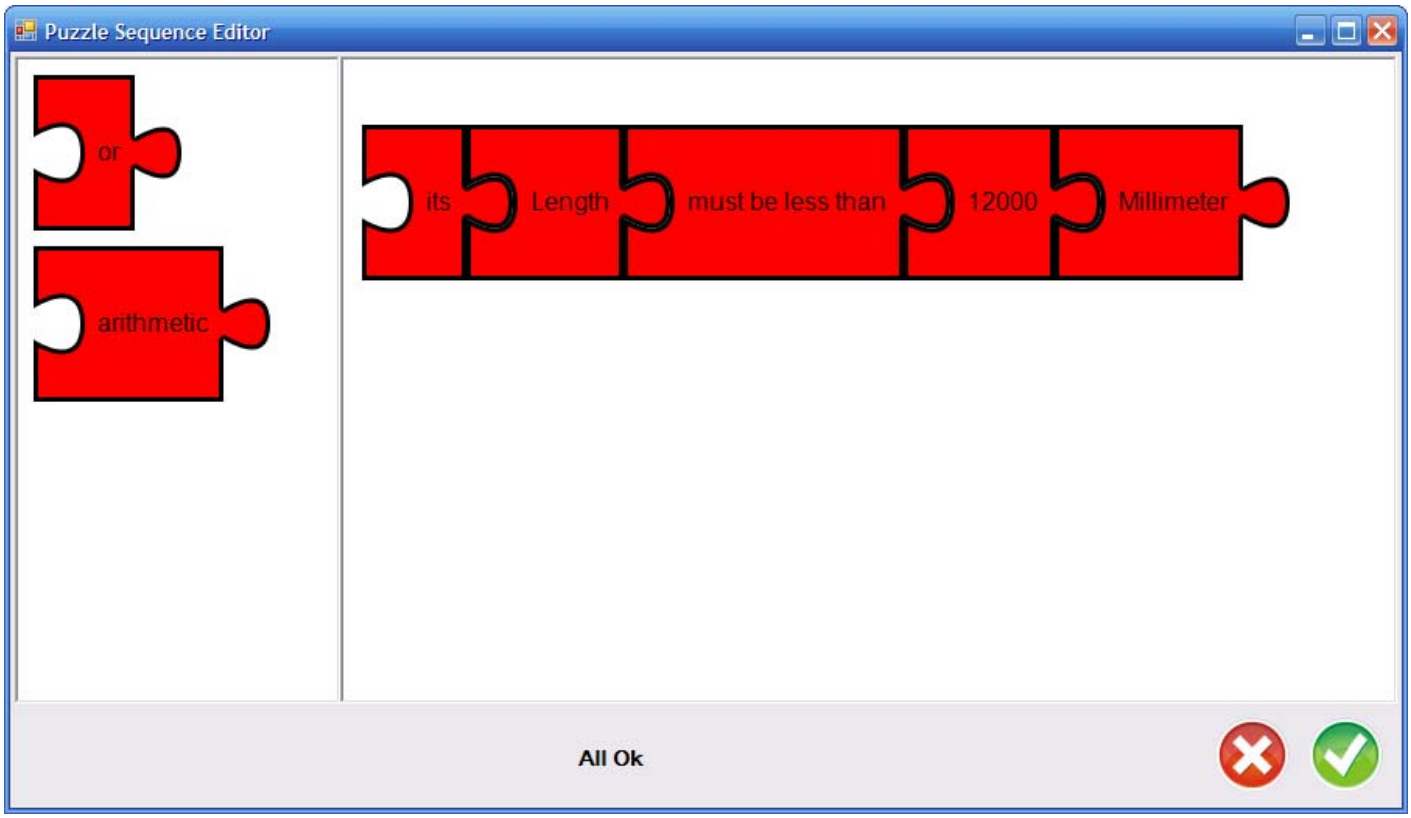

Figure 2. Puzzle piece editor

6.2 *Prototype implementation*

Since a good library for importing IFC's native file format (based on the EXPRESS language defined by STEP) was not yet available for Microsoft's .NET framework, we instead chose to work with the ifcXML format. Like IFC, this format is widely supported, but it has the advantage that it can be handled by the XML parser in the .NET framework.

Like the standard EXPRESS-based file format, the ifcXML format uses a lot of references; if an object is used more than once, it is defined once and then referenced whenever it is needed. Although this result 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, we replace all references 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. 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.

After adding the needed properties to all the objects, they can be used for constraint checking. In the prototype, the human-readable constraints are converted to computer-executable code 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;PROPNHeight;COMPINEQcgt;INT50;UNITmm
```

i.e. an Its piece (no input), a numeric property piece (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 is also how the library of pieces is filtered, since the parser can indicate which strings (pieces) it expects at any point. The parser converts

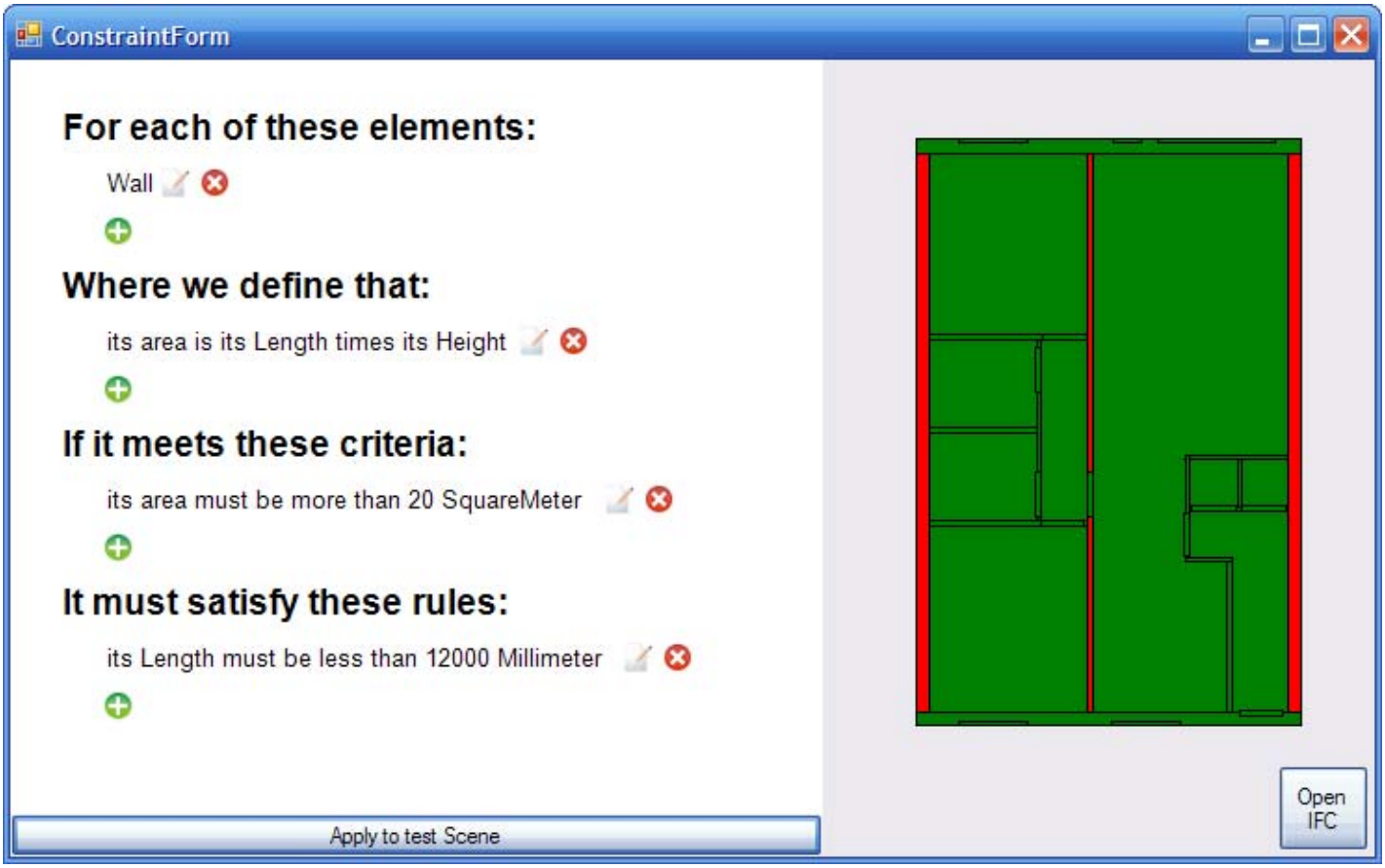

Figure 3. Prototype main screen

each (group of) puzzle piece(s) to a lambda function (a lambda function is virtually the same as a regular function; the only difference is that it does not have a name; lambda functions are frequently used in functional programming languages such as lisp or Haskell). The example above would be converted as follows (slightly simplified and using C# syntax):

```
;ITS;PROPNHeight becomes:
e => e.GetProperty("Height")

;COMPINEQcgt becomes:
e => ((a, b) => a(e).MoreThan(b(e)))

;INT50;UNITmm becomes:
e => new MillimeterValue(50)

So the whole constraint becomes:
element =>
 ((e => e.GetProperty("Height"))(element))
 .MoreThan
 ((e => new MillimeterValue(50))(element))

or, simplified:
element => element.GetProperty("Height")
    .MoreThan(new MillimeterValue(50));
```

which is a function that takes a building element and returns a Boolean, i.e. a functions 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.

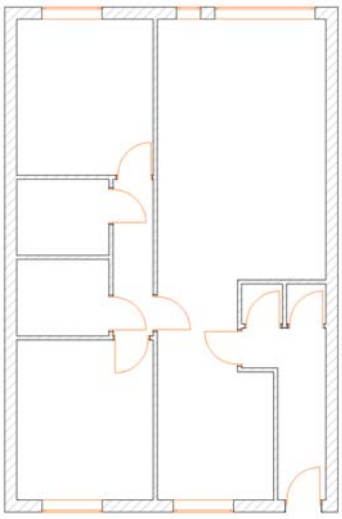

Figure 4. Apartment floor plan

6.3 *Test case*

As a test case we used the floor plan of a typical apartment, shown in Figure 4. There are four types of elements in this scene: walls, doors, windows and rooms (not shown in the image). This design was created in ArchiCAD and exported as an ifcXML file.

Figure 3 shows the results of importing this design into the prototype application, defining a constraint and checking it. Elements that violate the constraint are highlighted in red, other elements are colored green. In the case of the example constraint, the wall in the middle is the only violation, since the other walls that are longer than 6 m are all outer walls, which are 300 mm thick.

7 DISCUSSION

For the future of this project there are four main tasks:

- improve the constraint entry method for architects
- develop a CAD interface that makes it easy for clients to modify the design.
- choose a standardized constraint representation
- find a way to add the required information to IFC

7.1 *Improving constraint entry*

One of the alternative methods of entering constraints would be the solution based on highlighting mentioned in paragraph 3. Another solution would be to have architects type in the constraints with the help of autocompletion. Different approaches will have to be tested to determine which one offers the best balance between intuitiveness and speed.

7.2 *CAD interface for clients*

Although a good CAD interface will be needed to use this system in practice, it is only of limited interest from a scientific standpoint, since there are no real technical challenges to be solved. It is primarily an exercise in user interface design, with the main goal being to discover the minimum required set of features so that users will not feel overwhelmed by the program.

7.3 *Standardizing constraints*

There are many applications in the building industry that work with constraints already, even if they do not identify them as such. Constructive engineering applications indicate whether or not the construction will fail. Building physics software checks whether the building will not get too hot or too cold. Projects such as ePlanChecking (IAI 2005) check for building code compliance. These constraints are typically hardcoded, or at least stored in incompatible data formats, meaning that a design will have to be imported into multiple different packages to see whether all the constraints have been met. Aside from the risk of incorrect results 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.

By standardizing a representation of constraints that works for all domains of the building industry these problems disappear. One application can verify the entire design, which means you can get real-time feedback on the design. The cycle required to fix problems and re-check the design becomes much shorter. Aside from saving the architect a lot of time, this also facilitates incorporating client input into the design. Because errors are reported by the application, clients can modify designs without the need for direct contact with the architect.

There are several options for standardizing constraint representations. The first option would be to use the IfcPropertyConstraintRelationship in IFC. However, this option has a few disadvantages, which stem from the fact that you can only set constraints 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 option is the SMARTcodes project (ICC Online 2009), 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 (AST) that underlie virtually all programming languages (Louden 1997). Figures 5 and 6 (E-Bouw 2009) show the class hierarchy used to create these trees in SWOP. Like most ASTs, they distinguish nullary, unary and binary nodes. Since this approach is so common, it is a good method of storing constraints. 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. Both of these last two alternatives are viable options for storing constraints.

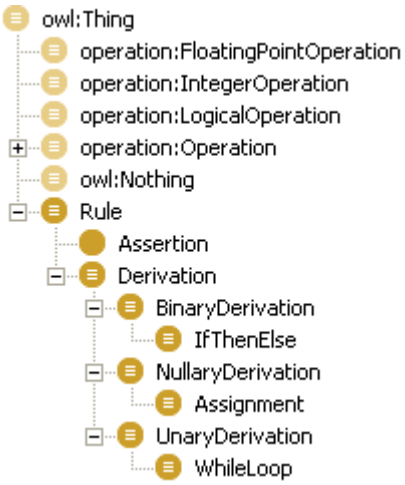

Figure 5. SWOP class hierarchy for rules

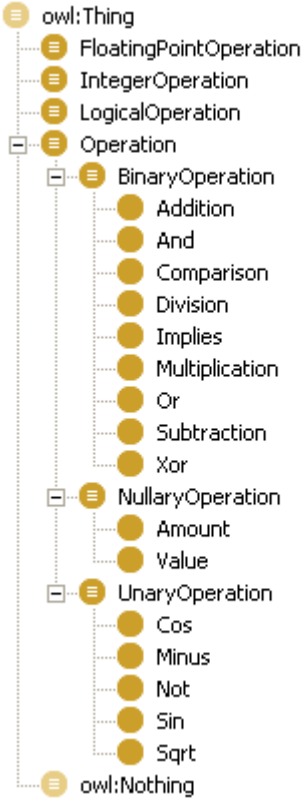

Figure 6. SWOP class hierarchy for operations

7.4 *IFC*

Although constraint checking on IFC models has proven to be possible, IFC was not explicitly designed for that purpose. As a consequence, several common parameters that architects want to use when specifying constraints either have to be inferred from other properties or are missing altogether. 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 (the durability class determines whether or not wood can be used in the exterior of a dwelling).

Additionally, many classes are not yet present, making it difficult or impossible to refer to such things as walk-in closets, dormers or sheds. 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) viscosity temperature derivative, thermal gradient coefficient for moisture capacity, thermal conductivity temperature derivative 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 to solve this problem. The first is a separate database with the missing information (e.g. maple wood has durability class 5). This approach was used to a limited extent in the prototype (a few extra properties, such as color, were hardcoded). Such a database would consist of a series of tables with data (e.g. colors, material types, etc.) and one large table that links together a property name and an object with a value, e.g. “IfcDurabilityClass”, “Maple”, 5.

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.

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 final solution is that it will take a lot of work to reach the adoption rate that IFC has.

8 REFERENCES

AEC 3 2009. *Building Codes & Regulations*. www.aec3.com/downloads/BuildingRegulations.pdf

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

Belblidia, S. & Alby, E. 2003. Implicit handling of geometric relations in an existing modeler. In *CAADRIA 2003 Conference, Bangkok, Thailand.*

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

Donath, D. & Böhme, L.F.G. 2007. Constraint-Based Design in Participatory Housing Planning. In *eCAADe 2007 Conference, Frankfurt am Main, Germany.*

E-Bouw 2009. *The SWOP Semantic Product Modelling Approach*. http://wiki.e-bouw.org/images/4/4a/SWOP_D23_WP2_T2300_TNO_2008-04-15_v12.doc

Eggink, D. et al. 2001. Smart Objects: Constraints and Behaviors in a 3D Design Environment. In *eCAADe 2001 Conference, Helsinki, Finland.*

Gross, M.D. 1996. Elements That Follow Your Rules: Constraint Based CAD Layout. In *ACADIA 1996 Conference, Tuscon, USA.*

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

Huang, C. & Krawczyk, R. 2007. A Choice Model of Consumer Participatory Design for Modular Houses. In *eCAADe 2007 Conference, Frankfurt am Main, Germany.*

IAI 2005. *CORENET e-Plan Check System*. www.iai.no/2005_buildingSMART_oslo/Session%2001/eSubmission_eplancheck_Singapore_Case.pdf

IAI 2009. *IfcPositiveLengthMeasure*. http://www.iai-tech.org/ifc/IFC2x3/TC1/html/ifcmeasureresource/lexical/ifcpositivelengthmeasure.htm

ICC Online 2009. *ICC Online | SMARTcodes™*. http://www.iccsafe.org/news/102006smartcodes.html

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

Lacy, L.W. 2005. *OWL: Representing Information Using the Web Ontology Language*. Victoria: Trafford

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

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

Nassara, K. et al. 2003. Building assembly detailing using constraint-based modeling *Automation in Construction* 12 : 365– 379

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

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

Swop 2009. *SWOP - Semantic Web-based Open engineering* Platform. http://www.swop-project.eu/

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

Wix, J. et al. 2008. Using Constraints to Validate and Check Building Information Models. In *ECPPM 2008 Conference, Sophia Antipolis, France.*