Rule Language Based Automated Compliance Checking for Interior Generative Design Using BIM

by

Christoph Peter Sydora

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Building Information Modeling (BIM) has become an integral part of the design process, as all building data is accessible in a digital representation and can be viewed in a 3D environment prior to construction. This supports the capability of evaluating or checking a model against building codes or design rules, imposed from construction codes to cultural preferences to the owners' styles and aesthetics, a procedure necessary to ensure a building meets the functional and safety requirements for occupants. However, as building regulations are typically represented in natural language, to date they have not been created with regard to the digital BIM design process. Therefore, checking a model against these design rules is still a time consuming and error-prone task involving knowledgeable individuals reading rule documents and manually assessing a building design. Furthermore, design rules are subject to interpretation rather than structured for machine interpretation allowing for rule assessments to differ among individuals.

To automate the design evaluation of a building model, this thesis describes a simple, yet extendable, domain-specific language for computationally representing building rules. We describe how implicit information is extracted from the BIM model, as necessary for the rule evaluation on the building model. Previous approaches to model-checking generally require experienced coding knowledge and have been tailored to meet specific building regulations with minimal support for rule creation.

The model evaluation also provides opportunity for automatically generating multiple valid alternative solutions, compliant to the design rules. Due to time constraints, designers typically only explore a few options; computers can efficiently create a potentially much larger number of alternatives, which users can then compare to select the design that best suits their individual preferences, depending on cost, environmental impact, and aesthetics. Therefore, unlike the previous methods of rule-based model checking and automated layout designs, this thesis bridges the concepts of automated model checking and design using a single unified rule language.

The research is evaluated on two particular instances of the general generative design problem. The first is the task of generating 3D kitchen layouts, based on a BIM model of the kitchen space, a product catalog of 3D models of kitchen furnishings, and a set of design rules. The generative-design method starts with an empty kitchen and implements a heuristic search of the solution space by incrementally selecting and placing a required item and checking the degree to which the resulting model complies with the given kitchen design rules. We have demonstrated the effectiveness of our method by comparing the designs it produces against a set of real-world kitchen examples, obtained from architecture diagrams available online.

The second task is to generate a living room, using rules interpreted from previous literature on layout design that were not written in a language-based evaluation method. Using the same method, we create living rooms that meet the design requirements from the previous literature.

Preface

This thesis is an original work by Christoph Sydora. Segments from this thesis has been published in the following literature:

- **C. Sydora** and E. Stroulia, “Towards Rule-Based Model Checking of Building Information Models,” Proceedings of the International Symposium on Automation and Robotics in Construction (ISARC), Vol. 36, pp. 1327-1333, 2019.

- **C. Sydora** and E. Stroulia (in press), “Generative Interior Design using BIM,” Poster presented at: 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys), 2019.

Acknowledgements

First and foremost, I would like to thank my advisor, Eleni Stroulia. It has been a privilege to work under such a kind, thoughtful, and enthusiastic professor, who is always available with constructive and helpful feedback.

I would like to thank Omid Ardakanian and Marek Reformat for serving on my defense committee.

My sincerest thanks to Mohamed Al-Hussein for introducing me to the area of building modeling and supporting my research over the years. I like to extend my thanks to Hexu Liu and Grujeet Singh for all the help they have provided, and to Ray Groothuizen and the Cadsoft software development team for their interest in my work.

For a stimulating lab environment, a special thanks to Kalvin Eng, Dillam Diaz Romero, Farnoosh Fatemi Pour, Shadan Golestan-Irani, Mashrura Tasnim, Ashley Herman, Diego Serrano Suarez, Samuel Jaeger, and Victor Fernandez Cervantes.

Finally, being surrounded by an amazing and supportive group of friends and family has been imperative for maintaining sanity (somewhat) throughout.

Thank you.
Christoph Sydora

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

One of the benefits of Building Information Modeling (BIM) is the ability to create 3D building models that capture the imagination and creativity of the architect, which has led to captivating building designs and layouts. However, although visually appealing, a building and its features must still meet their functional requirements in order to be considered practical. Building design guidelines and regulations exist to assist designers in the process of creating compliant models, as these rules are made to ensure the safety and well-being of the build occupants [9].

These guidelines are typically written in the form of natural language text documents which can be ambiguous to the reader [11]. Since they are often not written in a computer interpretable form or embedded in the BIM model editor, checking a building model for design flaws is a separate task that must be performed once the final design is complete. Even with the building information available in a BIM model, rules can require information implicitly defined based on the geometry of one or more model components. Given the potential size of the model, and the complexity of the rules, this task can be tedious, time-consuming, and error-prone [18]. Some efforts exist to automate the model-check of a BIM model, however, they typically hard-code a set of rules, a task that requires programming experience and knowledge of the internal structure of the BIM model. A better solution is a rule language that can describe a wide range of building regulations simply and efficiently, thereby reducing the total time needed to create rules and check for them against a building for design flaws.

Should an error be found, making design corrections can be difficult, as it may require the shifting and modification of multiple model items followed by rechecking the modified model. Furthermore, the designer's initial design decisions may cause them to be focused on a sub-optimal solution when other potentially better design alternatives exist. Designers may also be unaware of other design alternatives because of design tendencies due to prior experience.

Having computer-readable rules and guidelines, that can be used to assess designs and provide model compliance evaluations, would help the users decision process. In addition, they could be further used to automate the placements of items in accordance with the design rules and guidelines.

Domain Specific Languages (DSLs) have been proposed which allow for the creation and execution of computer readable rules. Of these, the most prominent are the Building Environment Rule and Analysis language (BERA) [22], [23], KBim [21], [31], BIM Rule Language (BIMRL) [44], and Visual Code Checking Language (VCCL) [34]–[36]. Each of these methods has been built to accommodate a certain regulation ruleset, and therefore, it is impossible to compare them and none of them can be broadly adopted. Other previous attempts at code compliance checking center around general purpose languages built as either add-ons to existing BIM software tools, such as Autodesk Revit [3], or as standalone applications. Solibiri [40] has been at the forefront of the Industry Foundation Classes (IFC) based rule checking industry, however, the rules are typically hard coded in software procedures and it is difficult to evolve them. To our knowledge, none of these methods have been placed in the context of generative or automated design.

State of the art automated layout design for interior spaces outlined in previous literature rely on mathematical formulations of rules which consequently depend on general programming languages. The baseline for automated interior layout based on design guidelines is in the work of [29] who used an empirically weighted summation of each mathematically expressed guideline. Each rule is implemented in a software procedure and a number of user inputs are required for defining object relations and properties. Other methods for interior layout generation, such as [46], have centered around learning relations based on existing interior layouts, however, this cannot be used to validate a model, as it is only an approximation based on previous layouts, nor can new user specific input constraints be easily added.

Therefore, the question we wish to address is: How can we computationally formulate design rules such that they are easily implemented for the purpose of validating a design and automatically generating new compliant designs?

This thesis proposes a Domain Specific Language (DSL) for building model-checking which can be further utilized for automated interior design. The language is expressive in terms of its domain of interior layouts, making use of computational logical to formulate complex rules. It uses an additional information layer that makes it extendable to new types, properties, and relations of objects. These explicitly defined, machine-interpretable rules are used for compliance checking of the model thus expediting the evaluation process for the building designer. As this evaluation can be performed rapidly relative to manual checking, this thesis further utilizes the rules for the purpose of generative layout design, through which a number of code compliant designs can be generated.

The contributions of this thesis are:

- An expressive **Rule Language** for representing design rules for the layout of elements in interior spaces;
- A **Model-Checking Algorithm** for evaluating the compliance of a BIM model against a set of design rules;
- A **Generative Design Algorithm** for automating the placement of elements in interior spaces, relying on design rules;
- A service-oriented implementation of the above in a **Software Toolkit** that can be invoked by external client applications.

Chapter 2 provides a review of related research and an environmental scan of existing software. Background to Building Information Modeling (BIM) and related concepts are explained in Chapter 3. The rule language is introduced and outlined in Chapter 4, followed by the model-checking outline in Chapter 5. Chapter 6 with give an overview and evaluation of the Generative Design approach. Finally, Chapter 7 provides a summary and concluding remarks.

Chapter 2

Related Work

In this chapter we look at previous methods for developing rule languages and their respective model-checking approaches to support the languages. Rule Languages and Model Checking go hand in hand as rule languages have been developed exclusively for checking models, therefore, they are assessed in a single section. Following this is an overview of automated and generative design for interiors and building components. Generative design is a broad field in the CAD domain and thus we only select those related works directly relevant to our own method. For a more comprehensive review of design-optimization algorithms, the reader should review [45] and [43].

2.1 Tools

Standalone Software

When researching compliance-checking tools, Solibri Model Checker (SMC) [40] is frequently mentioned as it is one of the few tools specifically built for the purpose of checking BIM models. SMC takes as input a building model in the form of the BIM industry standard of IFC. While the available rulesets, initially from the Norwegian Statsbygg handbook [42], can be modified by the end user by combining rule sets and deleting rules, support for editing individual rules is limited to changing the parameters or the provided rules. Additionally, there are a few rule templates for creating new rules, however, full customization of rules can only be done through the SMC Application Program Interface (API), which is not publicly available. There are a number of research papers that report how different checks might be implemented using the available rule templates [19], [24], and [13]; however, these are black-box approaches and it is impossible to comment on their accuracy, efficiency, generality, and expressiveness.

Model-checking tools have been implemented for the purpose of evaluating requirements of governing bodies, with differing levels of success. Singapore's CORENET ePlanCheck [15] has been noted as the most successful implementation, since, at one point, it was mandatory as part of the government's building requirement legislation [11], [18]. In Australia, DesignCheck [10] was built on the Express Data Manager (EDM) Model Server but, to the best of the authors' knowledge has since lost support. The General Services Administration (GSA) in the United States mandates that their project models be checked with rules implemented within SMC [11].

BIM API

While not specifically model-checking tools, BIM editors, such as Autodesk Revit [3] and Graphisoft ArchiCAD [14], provide APIs for add-on development, allowing access to the model's internal structure and object database and therefore, can, in principle, be used for model checking. This requires a high level of programming knowledge, even for the simplest checks. To address this challenge, some tools have been developed to perform the same functionality in a visual environment. These include tools such as Autodesk Dynamo [2], which works on the Revit platform, and Rhino Grasshopper [8]. These two tools are both graph-based visual editors that have some scripting available - Dynamo's Python scripting rather than C# as the Revit API. BIMServer [4], an open source IFC model repository platform, has a model-checking plugin, however, it requires direct coding in JavaScript. The scripts are then linked to the model for execution. This also requires programmatic coding knowledge and a strong understanding of the IFC vocabulary and syntax.

Table 2.1: Rule Languages and Model Checking Related Work

Year	Paper	Rule Interpretation	Evaluated Rule Domain
2009	[30]	Check-mate: Puzzle based GUI for simple rule checks. .NET application using the ifcXML file format	Simple checks on properties of walls, doors, windows and rooms
2011	[32]	Semantic Web Ontology based approach using Notation-3 (N3) syntax for the IFC/RDF graph	Acoustic performance checking
2011	[22]	BERA: runs on JVM using Solibri as IFC Engine	Building circulation and spatial rules
2013	[27]	BIMQL: SQL style query language built in the BIMServer environment. Can perform “Select” and “Set” operations	Unspecified
2016	[44]	BIMRL: SQL influenced language using xBIM as the IFC Engine for translating to Oracle schema.	Hospital visibility, Singapore’s Environment and Safety Rule, Room and Space exits, Accessibility Path Analysis using Circulation Graph
2016	[31]	KBim: Broken up into KBmLogic for Natural Language Parsing, KBimCode for rule syntax, and KBimAssess as the IFC Engine and check execution environment.	Korean Building Act Regulations
2017	[36]	VCCL: Node-based editor of rules, internally represented as XML rules	Korean Building Act Regulations

2.2 Rule Languages and Model Checking

Query Languages and Semantic Web Ontologies

As model-checking is, in theory, a query on a BIM model, and given the emergence of semantic-web technologies, there have been a number of methods using semantic web languages as the basis for the checks. Specifically, methods have worked with extendable IFC based ontologies of the BIM model to query for design flaws. In [32] the authors perform acoustic regulation compliance checking for BIM models using Resource Description Framework (RDF) graphs of the building model. The process requires a model to be passed through and IFC-to-RDF converter which can then be queried by a rule described in the Notation-3 (N3) syntax. A more general overview of these types of methods can be found in [33]. While this technology can be useful in extending the data schema, the query languages require a steeper learning curve and a data converter from IFC to RDF data models which is not a straightforward task.

BIM Query Language (BIMQL) [27] is another query language, built on the BIMServer platform. The language uses a syntax very similar to SQL that reads an IFC model, the goal being an easier transition for users more familiar with query languages as opposed to learning a programming language. BIMQL has the capability to check the existence of model elements and some minor model manipulation. Therefore, while query languages can perform the task of model-checking, they focus on the exiting data using complex query languages and are likely better suited for Quantity Take-off (QTO), that is to count the number of item instances in a model, than model-checking.

Rule-Checking Languages

The Building Environment Rule and Analysis Language (BERA) [22] was developed as a domain-specific programming language for model checking. The concept is built on providing model-checking capabilities without the need for precise knowledge of general-purpose programming languages [23]. However, the language derives heavily from Java which may be difficult for non-programmers and it is built on SMC as an IFC engine, and therefore is still quite opaque. The language also has a focus on aggregation relations rather than geometrical relations, therefore, it is difficult to see how a simple relation check like the distance between two objects would be represented.

BIM Rule Language (BIMRL) [44] represents another rule language approach. This method draws influence heavily from SQL, therefore, for a non-programmer the language can appear complex. This language does contribute some key concepts such as the representation of the data from complex IFC data to simplified shape representations and the use of temporary geometry for spacial based evaluations [41].

KBim [21], [31] was built specifically for expressing the Korean building act legislation's into commutable form. The method is broken down into KBim-Logic [21], a tool for assisting users in the natural language parsing and information extraction of the rules based on objects, properties, and high-level methods stored in SQL database tables, and KBimCode, which then further converts the KBimLogic structured rules into computer executable code [31]. The code checking system is called KBimAssess-lite [6]. While the KBimCode language structure is comparable to ours, the language is more closely related to that of a programming language compared to ours with a list of functions pre-coded.

While these represent the most comparable solutions, we believe our implementation represents an even lower level of complexity as it has a focus specifically on interior design rules. We do anticipate that our language would be able to support rules more akin to those used in the case studies, such as the Korean Building Act, however, this will be left to future evaluation and research. For the purpose of our automated design, the rule language is expressive enough for our test cases as evident in the generative design section.

Furthermore, unlike these previous methods, our language supports the ability to scale the results of rules which is necessary for a generative design model. Should a rule fail, our language is able to additionally determine a severity of the failure. This concept is necessary for the generative design to adjust and improve configurations of the model layout such that a locally optimal solution can be found.

Visual Programming Languages (VPL)

Some approaches have taken the Rule Languages one step further by adding a visual component to them, in the same sense that Dynamo is a visual language for Revit's API. This is intended to allow for more complex rules to be created without adding the need to code programming, although, to our knowledge, this has not been tested for ease of use.

Check-mate [30] first introduced this as a very simple puzzle-based interface that allowed connecting pieces that together would form a structured rule, however, the expressiveness of this language was limited. The Visual Code Checking Language (VCCL) took a node-based approach, calling it a "white-box" approach with the available nodes to be extendable as the project matures [34], [35]. The language was then refined to support more complex rules by modularizing nodes, thus allowing for nodes to build around other predefined nodes [36]. In similar fashion, KBim has also since implemented a VPL version of KBimCode to improve ease of use [20], although, to our knowledge there have been no studies to verify this does improve usability.

Creating a VPL from our language will be implemented in future work, however, as with the Rule-Checking Languages, these methods have yet to be explored in the capacity of generative design.

Natural Language Processing (NLP)

Attempts have been made to parse natural language rules from design handbooks and regulation texts. While such approaches could potentially simplify the rule-creation process, many of the natural language rules lack clarity and unambiguity required to be directly parsed without any human intervention or interpretation. One of the more commonly cited approaches in this vein is that of [16], [17] which used a four-sentence component classification to parse natural language rules, namely Requirement, Applicability, Selection, Exception (RASE). Another use of NLP has been to identify information from rules that is missing or may need to be added to models [47]. Such methods could potentially be antecedent to our method and will be explored in the future.

2.3 Generative Design

Germer and Schwartz [12] developed a method that uses an object hierarchy, relating every object with a parent object and placing objects in alignment with their parents objects. The automated furniture arrangement starts with a fully furnished room and evolves at each iteration towards a more compliant layout. Rooms have a predetermined furniture count and style. The possible alignments of an object to its parent are above, below, in font of, behind, left to, and right to, with objects being placed directly touching and aligned with their parent. New rules can be added using these alignments but no other rules are possible to create (such as distance between objects). Each object, or agent as they describe it, is in one of three possible states, namely search, arrange, or rest. When in the search state, the agent looks for an object that can serve as its parent, based on the hierarchy. When one is found, it will enter the arrange state and identify if there is an available side available on the parent object such that it can align with it. If an alignment is possible, the agent enters the rest state and it will no longer move; if not it repeats back to the search state. If all parents are full, the object is deleted. When all remaining objects have found a parent, the model is said to be full and the process concludes. The alignment rules used in their method could be represented in our language using the same alignment checks (above, right of, etc.) with a distance of 0 between the objects.

Merrell et al. [28] proposed a generative building design process, based on rules, such as room adjacency. Features learned from a database of existing building layouts include total square footage, footprint, room, per-room area, per-room aspect ration, room-to-room adjacency, and room-to-room adjacency type, each of these features being hard-coded. These determined the initial rooms in the building and their sizes. The layout rules were based on hard-coded accessibility, dimensions, floors, and shape of the building. The generation-process operations were limited to “sliding a wall” and “swapping rooms”. The evaluation metric is a weighted combination of the design rules. Although the task differs from our implementation, as we focus on the inte-

Table 2.2: Generative Design Related Work

Year	Paper	Initial State	Operations	Evaluation Rules
2009	[12]	Fully furnished room	Agents alternate between search, arrange, and rest states until all agents are in the rest state. Search state is when an object looks for a parent object to latch on to	Direct alignment based on bounding box face availability
2010	[28]	Building floor plan predetermined by previously observed building models. Number of rooms of each type and size are set before operation begins	Operations include “sliding a wall” and “swapping rooms”	Harde-coded rules on accessibility, dimensions, floors, and shape of the building
2011	[46]	Randomly furnished room	The possible actions are object translation, rotation, or an object swap with another object in the scene, as well as the movement of controlled pathway points for a pathway object	Weighted sum of coded cost functions for accessibility, visibility, pathways connecting doors, similarity to prior configurations, and objects pairwise relations
2011	[29]	User places all furnishing in the model before layout generation	Variations at each step include rotations and translations of objects on the floor plane, and swapping of object locations and orientations	Hard-coded rules on clearance, circulation, pairwise relations, conversation, visual balance, alignment, and emphasis.
2013	[1]	Randomly furnished room	Crossovers and mutations of models were a “gene” represents a X, Y, or θ (angle) value	Hard-coded functions for accessibility, visibility, pairwise distance and angle. Also considers users model ranking

rior of building rather than the room adjacency configurations, we believe the rules can be described using our language assuming spaces are updated in the building model. As their layouts are represented in 2D, expanding to a 3D BIM might be a difficult task as a room moving effects the supporting rooms walls.

In a similar methodological vein, Yu et at.[46] propose a method for automatically arranging interior furniture, based on relationships learned from previous arrangements. The rooms start with all objects placed in the model. The features extracted from previous arrangements are the distance and orientation to the nearest wall. Other rules are mathematically calculated using spatial relationships for objects based on an object accessibility and visibility spaces. Additionally, there must be a pathway connecting doors and object pairwise relationships which the user inputs along with the desired distance and orientation of the pair. Additional features and rule functions would require manual encoding. The possible actions are object translation, rotation, or an object swap with another object in the scene, as well as the movement of controlled pathway points for a pathway object. The method uses simulated annealing with a Metropolis-Hastings state-search step to determine the actions to take towards a local optima. At each step, the model is evaluated based on mathematically coded cost functions for accessibility, visibility, pathways connecting doors, similarity to prior configurations, and objects pairwise relations. The overall cost function is a empirically determined weighted summation of each individual cost function. Our language would be able to support their rules using alignment and distance relation properties for visibility, accessibility, and pairwise relations. The rules for the pathway would require implicit object creation (Virtual Objects) in the same way they create their pathways, with a check on the geometrical property of the pathway. Any prior features values, such as distance to walls, would be a separate rule with the learned features acting as the property check value and, therefore, using prior distances and orientations could easily be incorporated in our rule language.

In [29], the user first selects and places a number of objects, such as couches, chairs, tables, and shelves in the scene. The user has the option to lock the placement of some objects before starting the generation process. Their method uses seven design constraints, i.e., *clearance*, *circulation*, *pairwise relations*, *conversation*, *visual balance*, *alignment*, and *emphasis*. These design constraints are evaluated in the 2D floor plane of the space, using hard-coded mathematical formulas based on center distances, area intersection, direction vector angles, and a number of implicitly created objects, such as room centroid and free configuration space. A number of user inputs are required, such as object groupings and a focal point with axis. Over each iteration in the genetic algorithm, the variations to the model objects include rotations and translations of objects on the floor plane, and swapping of object locations and orientations. The method adopts a Markov Chain Monte Carlo sampler approach, to pursue solutions closer to the objective function. A number of iterations are completed which output a number of possible solutions. Each of the constraint evaluation results in a value between 0 to 1. The overall design quality is an empirically formulated weighted sum of the seven design constraint values. We use this method as an evaluation to determine if their rules can be recreated using our rules language. The results are outlined in the evaluation of Chapter 6.

Akase and Okada [1] proposed an interactive method for automated interior-space layout, beginning with a random pre-populated layout. Using rules similar to [46], the rules hard-coded functions for accessibility, visibility, pairwise distance and angle. In addition to the rules, the user evaluated each intermediate generated layout, which is added to the fitness functions from the design rules. The search selects highly rated generated models and performs crossovers and mutations of those models. The resulting models from these operations are displayed to the user for further evaluation and repeats until a model is satisfactory to the user. In effect, the method requires the users to evaluate the designs and guide the process towards better solutions, which our method does not take into account, although, the rules, as with [46], can be recreated.

Interestingly, each of the previously described methods is built around CAD or 3D game development, with [1] using XML to store object attributes. None of them make use of BIM as part of the process although it contains the same data in a standardized representation.

Approaches using BIM include work by [39] who created a generative-design system on top of Revit that manipulates window sizes and invokes the Green Building Studio APIs to determine the resulting energy-analysis metric. In a subsequent work [37], the authors used Autodesk Dynamo [2], a visual-programming tool, to solve a similar problem, namely that of discrete window-size optimization for reducing LEED daylight usage and energy consumption. Finally, [38] created a framework for optimization BPOpt based on Dynamo, which breaks the generative design into fives phases: Decision variables (input), Initial Random Population (initial setup), Fitness Functions (evaluation), Generation Loop (decision making), and writing to CSV File (output). Depending on the problem the user plans to optimize, they can alter the input parameters and develop the appropriate fitness functions, thereby making the problem seemingly independent of the type of generative design.

Chapter 3

BIM Concepts

The purpose of this chapter is to briefly outline some basic principles which will be referred to throughout the thesis. First, a BIM model, as opposed to a strictly CAD model, contains classifications of building components and relations among them. Additionally, each BIM model component contains properties or attributes which can extend beyond the geometrical. These are vital for evaluating the compliance of a model. We consider a BIM model to have three main sets of information, namely objects, relations, and properties (also known as attributes), providing the basic information required for the rule language.

- Objects contain a unique identifier (such as a name or Guid), a class type (henceforth known simply as type), a shape representation (or mesh), and a set of properties
- Properties are defined as name-value pairs
- Relations are defined in terms of a list of objects and a set of geometrical relation properties

While in practice, each software vendor has its own internal representation of a BIM model, these concepts are typically in some form present in every BIM structure, exported by most modern tools. Relations are typically used for object association and assignment but not for geometrical relations.

Furthermore, the proposed rule language draws heavily from Industry Foundation Classes (IFC), an open standard for BIM model representation and file structure, the most recent being IFC4 [5]. As the name implies, IFC comprises of a standard set of classes for BIM concepts, deriving from the concept of Object-Oriented Programming (OOP) where each model object is represented as a class. In this sense, the class and type of the object are synonymous and are an inherent feature for each building object. We utilize the class hierarchy of IFC as the basis for object types which serve as a means of classifying and filtering objects in the model.

As IFC is the more popular standard BIM representation, our implementation and toolkit are based around importing, exporting, and utilizing its class hierarchical structure. However, as IFC shape representations are complex in nature, the imported model must go through a parsing phase in order to convert them into more computationally efficient structures.

We created built in-house Unity-based IFC building interior editor that is currently capable of reading an IFC file, displaying the model in 3D, enabling a user to review an object catalogue and add *IfcFurnishing* elements to the IFC model, and save the new model containing the added furnishing. The IFC editor uses our in house IFC Engine package, build on top of the open GeometryGymIFC Library [25], which can parse an IFC file and convert the complex IFC shape representations into Unity supported triangular meshes. Over time, this IFC Engine will need to be updated to ensure compatibility with the latest IFC versions.

Chapter 4

The Rule Language

The rule language is designed to express basic rules about

1. Geometrical and non-geometrical properties of objects that are either actual or virtual IFC elements;

2. Geometrical relations between pairs of elements;

3. Complex rules that are logical compositions of the above two types of basic rules.

As many rules are inherently logic-based, our language derives much of its structure from mathematical logical reasoning. As [30] suggested, it is easy to see the similarity between a statement "For every x in Real Numbers..." in mathematics with "For every Window..." in building regulations. We also use simple boolean logic for basic composition of checks.

The following sections will describe the syntax of the rule, how a rule is evaluated based on the multiple instances of the relevant rule objects, and one implementation of an editor for the rule language. We conclude the chapter with a discussion and summary.

4.1 Rule Syntax

To demonstrate the syntax of our language, we present an example of a rule in our language and its natural language description is shown in Table 4.1. The natural language form of rule is "The refrigerator must have 16" of counter space on one side". They relevant objects here are "refrigerator" and "counter space". The key property implied is the width of the counter space, which is checked against being 16 inches or greater. The relation is that the counter space is "on one side" of the refrigerator. Furthermore, it is important that both these checks are satisfied in order for this whole statement to be true and thus the rule passing. The following will show how each of these is structured in our language to formulate the rule.

Table 4.1: Example rule in natural language and in the proposed rule language.

Natural Language
"The refrigerator must have 16inches of counter space on one side", which is re-formulated as follows in our language: "For every object of type "Refrigerator" in the space, there must exist another object of type "CounterTop" that is at least 16inches wide and is placed right next to the refrigerator.

Rule Language
ALL Obj0 $\in$ IfcFurnishing {FunctionOfObj = "Refrigerator"}; ANY Obj1 $\in$ IfcCounterTop; (Obj1 MustHave Width $\geq 16INCH$ AND Obj0 and Obj1 MustHave BoundingBoxDistance $\leq 1FT$ AND Obj1 and Obj0 MustHave IsNextTo = TRUE)

Property Check

Object properties contain a name for defining the feature, and a value, which can be of three types: boolean, strings, or numeric. Some properties can be present in the model, such as a name or comment, however, many of the geometric property values are implicit and therefore not contained in the property set of the object. In those cases it is required to calculate some of the basic geometrical properties of individual objects.

Table 4.2 contains a list of geometrical functions for calculating properties of object as required by the rules. Each of these determines a value based on the geometry of the object or in some cases the name of the object. Most of the properties of the objects required for the checks are simple functions such as determining an objects "Width". Generally, these are performed on the local coordinate of the object as they are often properties on the object irrespective of the model.

Therefore, if a rule, such as in our example in Table 4.1, performs a check on the width of an object, the "Width" function from Table 4.2 is called with the object as the parameter. When the value is calculated, a new property is created with the function name, in this case "Width", as the new properties name, and the function result as the properties value. The created property is then added to the objects property set and the new property's value is compared against the check-value, which results in either true or false. Additionally, the result can be negated depending on if the check "MustHave" or "MustNotHave" be true. The property check in the Table 4.1 is:

$$\text{Obj1 MustHave Width} \geq 16INCH \tag{4.1}$$

Width is the property name, $\geq 16INCH$ is the check value and unit, and *MustHave* is the possible negation term. *Obj1* is a placeholder for the object that is being checked which we explain in the Rule Evaluation section.

Because a newly created property is added to the objects property set, it allows for the reuse of the property later, should another function call for it. Therefore, before a property is calculated for an object, a search on the objects existing properties is performed. If the property with a matching property name exists, then that existing properties value is used rather than requiring any calculation. Thus, properties are in a sense cached by the object during the check process, thereby speeding up the check. However, they are not saved to the model as model-checking does not modify the model but only evaluates it.

Relation Check

For the case of relations between objects in our language, we initially restrict them to involving pairs of objects. A relation is simply a container for the properties of the relationship of the objects. An example of a relationship property is the distance between the objects or if the objects overlap. These relationship properties work in the same fashion to object properties, in that

Table 4.2: List of geometrical properties

Name	Description
FunctionOfObj	Returns a string value corresponding to the type of object. This is determined by a keyword search of the objects name. As IFC types do not extend past *IfcFurnishing*, this serves as a type check.
Width	The width of the object, determined by the difference between the largest X value and the smallest X value of the objects local vertices
Depth	The depth of the object, determined by the difference between the largest Y value and the smallest Y value of the objects local vertices
Height	The width of the object, determined by the difference between the largest Z value and the smallest Z value of the objects local vertices
MinEdgeLength	The length of the smallest edge of the mesh of the object
MaxEdgeLength	The length of the largest edge of the mesh of the object
TotalEdgeLength	The sum of edge lengths of the mesh of the object

both are defined by a name and a value. Therefore, similar to object properties, if a rule checks a relationship property, such as “IsNextTo”, the function “IsNextTo” from Table 4.3 is called, for the case of relations however passing both the objects as function parameters. When the result is calculated, a relation with the first object and the second object is created with a new property, having the name of the function as the property name and the functions return value as the property value. The value can then be checked against the relation check value. The relation checks in the Table 4.1 are:

$$\text{Obj0 and Obj1 MustHave BoundingBoxDistance} \leq 1FT \tag{4.2}$$

and

$$\text{Obj1 and Obj0 MustHave IsNextTo} = \text{TRUE} \tag{4.3}$$

As with object properties, if a rule requires a relation check for two objects, it will first search the existing relations to find if a relation between the two specific objects already exists, in the same object order as the rule specifies. If so, it will further search that relation’s property set for a property that matches the required relationship property.

Because some of the relation properties are directional, the order or direction of the objects in the relation is critical. For example, for the property of “IsBehind”, object1 being behind object2 does not imply object2 is behind object1. Therefore, two relations between object1 and object2 would exist, one being object1-to-object2 and object2-to-object1, potentially containing different properties and value. Explicit function naming is important to assist in determining the order to this as best as possible.

Logical Composition Check

While the basis of the rule check is on the property values of the objects and relations, rules often posses conditional logic based on the combined results of each individual property and relation check. In the example in Table 4.1, this is the expression:

$$((4.1) \text{ AND } (4.2) \text{ AND } (4.3)) \tag{4.4}$$

The result of this expression is the aggregation of the encapsulated property check and relation checks, dependent on the logical operation, in this case the logical *AND* operator. Other logical operations supported in the language are the *OR* and *XOR* operators. In addition to property checks and relation checks within an expression, a logical expression can also have nested logical expression, containing different logical operators. Thus, when determining a logical expressions result, all nested results must first be determined.

4.2 Rule Evaluation

The property check, relation check, and logical composition of the two provides a means to determine the result of an instance of a rule, namely for a particular *Obj0* and *Obj1*. In this case *Obj0* and *Obj1* being place holders for instances of the relevant objects which are in this case refrigerators and countertops. However, a model often contains multiple object instances that are relevant to a rule. Thus, in addition to determining the result of each pair of relevant objects, each of those results must be aggregated to determine a single pass or fail result for the total rule.

Table 4.3: List of geometrical relations

Name	Description
BoundingBoxDistance	The distance between the bounding boxes of the two objects.
CenterDistanceXY	The distance between the centers of the two objects, projected onto the XY plane.
FrontCenterDistance	The distance between the front centers of the bounding boxes of the two objects.
FacingAngle	The angle between the forward direction vector of the first object with the vector from the center of the first object to the center of the second object.
AlignmentAngle	The angle between the forward direction vector of the first object and the forward direction vector of the second object.
IsBehind	A boolean value which determines whether the first object is behind the second object. Calculated by checking if a ray in the backwards direction of the second object from each of the bounding box vertices and the center of the bounding box intersects with the bounding box of the first object.
IsInFrontOf	A boolean value which determines whether the first object is in front of the second object.
IsLeftOf	A boolean value which determines whether the first object is left of the second object.
IsRightOf	A boolean value which determines whether the first object is right of the second object.
IsNextTo	A boolean value which determines whether the first object is next to the second object. Calculation is the same as IsBehind although in both the left and right directions. Note this does not imply the objects are touching.
IsAbove	A boolean value which determines whether the first object is above the second object.
IsBelow	A boolean value which determines whether the first object is below the second object.
IsInside	A boolean value which determines whether the first object is inside the second object.
AreOverlapping	A boolean value which determines whether the first object and the second object are overlapping.

Therefore, the rule-evaluation process is broken down into three steps. First the model must be searched to find all object sets relevant to the rule and build up a rule instance table using the cartesian product of the relevant object sets. Each row in the rule instance table is an instance of a rule and a column is a relevant object set. Next, the result of each instance of the rule is determined based on the individual property checks of the object and relations, aggregated based on the logical composition of the rule. Finally, a single result for the rule as a whole is determined based on the clause of each of the relevant object sets.

Identifying Relevant Object Instances

The first and third steps both pertain to the relevant object sets and thus we combine them into a single rule component known as an existential clause. The existential clause takes the full set of objects in the model and filters it based on the type of the object, such as *IfcFurnishing*, and a set of the property checks that it must pass. As the IFC schema provides a type hierarchy for building object classes and their super- and sub-classes and our checks are performed on models represented in IFC, we use their type classification as opposed to defining our own. In the case of our example in Figure 4.1 table (5) this is the two statements:

$$\text{ALL Obj0} \in \text{IfcFurnishing } \{\text{FunctionOfObj} = \text{“Refrigerator”}\}; \tag{4.5}$$

Where in this case we search through all model object, selecting only the *IfcFurnishing* objects that have a property *FunctionOfObj* value equal to *Refrigerator*. And

$$\text{ANY Obj1} \in \text{IfcCounterTop}; \tag{4.6}$$

where only the *IfcCounterTop* objects are selected.

Each existential clause set corresponds to a subset of the objects in the model. Once each subset of filtered objects is created, all objects from separate subsets are combined via a cartesian product resulting in a table of rule instances. Each column of the rule instance table corresponds to a existential clause object set and each row an instance of the rule. It is possible for an object to be in multiple subsets if it passes the filters of multiple existential clauses. In these cases, no instance is created, as we do not consider self relations of objects; in a sense self relations would be no different than an object property. Figure 4.1(5) shows an example of a rule instance table resulting from the existential clauses of the example rule in Figure 4.1(4). In this case there is a single refrigerator object in the model and four countertops.

Rule Evaluation on an Object-Instance Tuple

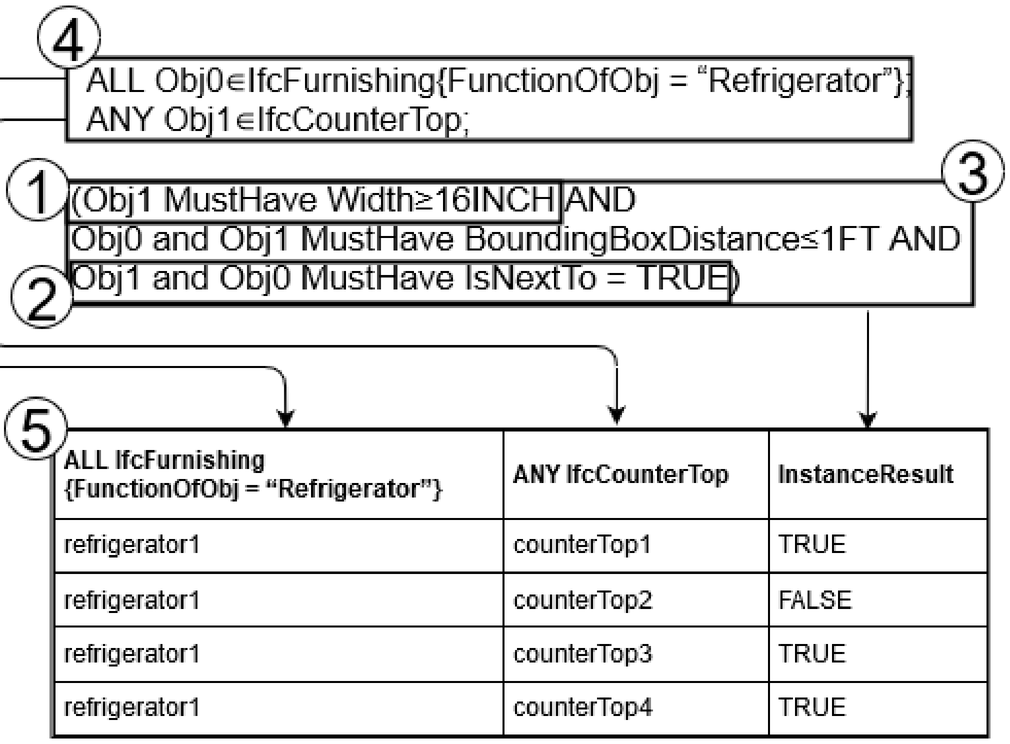
In order to evaluate an instance of the rule, and determine the resulting true or false value in the rightmost column of Figure 4.1(5), we use the rules property and relation checks and the logical expression which correspond to (1), (2), and (3) in Figure 4.1 respectively. This requires mapping the relevant object set objects in each rule instance row to the placeholder object in the logical expression. For instance, in the previous Equations 4.1, 4.2, and 4.3, the objects *Obj0* and *Obj1* represent the objects from the relevant object sets in 4.5 and 4.6 respectively. Thus, each rule instance result is the result of the logical expression, with the object in the rule instance passed as the parameters of the logical expression of the rule. The result is a true or false for each rule instance which is placed in the rightmost column of the rule instance table.

Overall Rule Evaluation

Some rules require that all rule instances pass, while others depend on only the condition that one instance of the rules passes. For example, in the rule in Table 4.1, all refrigerators require one countertop space next to them, however, the rule does not imply refrigerators require all countertops in the model to be next to them.

Therefore, once each of the results of the rule instance, or rows in the rule instance result table, have been determined, the final result of the rule is determined using the ALL/ANY/NONE clause of the existential clause(s).

We describe this algorithm in pseudocode in Algorithm 1. The algorithm takes as input the rule instance table and the existential clauses, specifically a list of their clauses. The algorithm starts with the leftmost column of the


4
ALL Obj0∈IfcFurnishing{FunctionOfObj = "Refrigerator"};
ANY Obj1∈IfcCounterTop;
1
3
(Obj1 MustHave Width≥16INCH AND
Obj0 and Obj1 MustHave BoundingBoxDistance≤1FT AND
Obj1 and Obj0 MustHave IsNextTo = TRUE)
2
5

ALL IfcFurnishing {FunctionOfObj = "Refrigerator"}	ANY IfcCounterTop	InstanceResult
refrigerator1	counterTop1	TRUE
refrigerator1	counterTop2	FALSE
refrigerator1	counterTop3	TRUE
refrigerator1	counterTop4	TRUE

Figure 4.1: Rule instances extracted from a model based on a sample rule with instance evaluation based on the rules logical expression term.

rule instance table and groups the rows by equal objects in that column (lines 10-19). Then for each grouping of rows, or sub-table, recursively calls the evaluation method (line 22), not including the current existential clause set or table column. The recursive call ends when a single row and column exist in the sub-table, meaning only the rule instance result remains which is the returned boolean value (lines 7-9). Then, the results of each sub-table are aggregated based on the clause of the existential clause for the grouped column (lines 23-31). When all sub-tables have been evaluated, the final result is returned (line 33). In the case where no relevant objects exist, meaning one of the existential clause sets is empty, then the rules returns the default value, which in this case is set to *FALSE* (line 4-6).

Note that the existential clauses are non-commutative. This means that the order of the ALL/ANY/NONE can change the final rule result. An example of this is ALL microwaves must be on ANY countertop. Using this order, every microwave must be on a countertop that does not need to be the same

Algorithm 1: Recursive function for ALL/ANY/NONE.

```
1  Function EvaluateRule(RuleInstanceTable, Clauses):
       Input:
           $RuleInstanceTable$
           $Clauses$
       Output:
           $Result$
2      $rowCount \leftarrow RuleInstanceTable.RowCount$
3      $columnCount \leftarrow RuleInstanceTable.ColumnCount$
4      if $rowCount == 0$ then
5          $Result \leftarrow FALSE$
6      end
7      if $rowCount == 1 \wedge columnCount == 1$ then
8          $Result \leftarrow RuleInstanceTable[0][0]$
9      end
10     $Column0Split \leftarrow newDictionary\{key \leftarrow object, value \leftarrow$
        $rowList\{\}\}$
11     foreach $row \in RuleInstanceTable.Rows$ do
12         $Object \leftarrow row[0]$
13         $rowMinusObject \leftarrow row[1 : end]$
14         if $Column0Splitcontainskey == Object$ then
15             $Column0Split[Object].Add(rowMinusObject)$
16         else
17             $Column0Split.Add(\{Object, rowMinusObject\})$
18         end
19     end
20     $returnResult \leftarrow TRUE$
21     foreach $subTable \in Column0Split.Values$ do
22         $methodResult \leftarrow$ EvaluateRule($subTable, Clauses[1 : end]$)
23         if $Clauses[0] == ALL$ then
24             $returnResult \leftarrow returnResult \wedge methodResult$
25         end
26         if $Clauses[0] == ANY$ then
27             $returnResult \leftarrow returnResult \vee methodResult$
28         end
29         if $Clauses[0] == NONE$ then
30             $returnResult \leftarrow returnResult \wedge \neg methodResult$
31         end
32     end
33     $Result \leftarrow returnResult$
```

countertop. If the order were ANY countertop must have ALL microwaves on it, then a single countertop must have each of the microwaves on it. Therefore, if two microwaves sit on different countertops, the first example would pass where as the second would fail.

Virtual Objects

There is also the case where a rule requires a check on an object that is not explicit in the model. Objects such as walking-space or corners fall into this category as one does not model a corner but rather they are implied as the connection locations of walls. Rules can often reference or check against these object types in the same way they would for an explicit object.

Our approach to address this is to have a collection of methods for creating the objects that can be called by the rule-checker, similar to the object and relationship properties. If the rule references a virtual type, the types can be created and added to the model as objects. We define these implicit types as Virtual Objects (VOs). As part of the existential clause, the type is used to filter for relevant objects for the rule. The type is selected from the the existing IFC defined classes, however, in a addition, the type can also be selected from a set of available VO types. For our implementation we created a number of additional virtual types, which can be seen in Table 4.4.

When a model-check is initialized with a given model and ruleset, the initialization process will first check all rules and find all unique VO types. For each of these VO types the initialization will call the function corresponding to the type name. The function takes in the list of all model objects as the function parameter. The result of the functions is a list of all that type of VO created based on the current state of the model. The VOs are then added to the list of all model objects and for the remainder of the model-check treated the same as any other object.

In the example in Table 4.1, as seen in the second existential clause 4.6, we call for a type *IfcCounterTop*. This type does not exist in the current IFC schema, however, it is one of the types we have defined in our VO table. This is because in practice, countertops are modeled differently by different users, and thus they are difficult to consistently check in a model. For instance, some countertops are modeled as a single object that surrounds the sink, thus, determining whether there is a certain amount on each side of the sink cannot be done by the width and would require extra computation. Therefore, we created a function that takes in the objects in the model and determines segments of flat surfaces that can act as a countertop or landing area. We create VOs as model objects, by which geometrical properties can be calculated as with real model objects.

Table 4.4: Virtual Objects for kitchen and living room rules. All Virtual Objects are sub-types of the IFC type *IfcSpatialElement*.

Virtual Object	Description
IfcKitchenTriangle	Object that connects the front center points of the sink, range, and refrigerator. The resulting mesh is the front center point of each of the objects projected onto the floor the objects are placed on. The mesh has only three vertices and three edges.
IfcCounterTop	Object that represents the flat countertop surfaces in a kitchen. The mesh is created by finding neighbouring similar cabinets and creating a single flat surface above all of them. This is because countertops are often modeled as part of the top of the cabinets.
IfcCorner	Object that represents the intersection of walls. Objects mesh is the mesh intersection of the walls.
IfcFurnishingCoM	Object that represents the center of mass of all *IfcFurnishin* items in a model. the mass is calculated as the geometrical volumn of the object (width x depth x height). Result is a single point in 3D space.
IfcRoomCentroid	Object that represents the center of an *IfcSpace*, projected onto the floor below the space. Result is a single point in 3D space.

4.3 Rule Editor

In our system, rules are edited through a special-purpose syntax-aware editor. This editor restricts users from entering in invalid rules by guiding the users through each existence clause, property and relation check, and logical expression addition. Rules must be grouped into rulesets which serve to group rules based on common functionality or projects. Rulesets have an ID, title and descriptions, while in addition, rules have an associated Error Level, such as Error, Warning, Recommendation, etc. Each of the rulesets from the editor can be exported to JSON format, allowing for sharing of rulesets among different users.

The Rule Editor uses a empirically defined subset of the IFC hierarchy as the possible types the existential clause can select to filter the relevant model objects. Types from Table 4.4 are available from the VO function list in addition to the IFC types. The available geometrical properties in the Table 4.2 and 4.3 provide the possible properties to check. However, if the user is aware of properties that have been predefined such as user entered tags or comments on each object, they can enter those values in manually and perform checks on them as well. Finally, the Rule Editor output rulesets in the form of JSON files.

4.4 Discussion

Rules inherently have a level of ambiguity to them. A simple example is when a rule refers to the distance between two objects. While intuition might say the distance refers to the length of the shortest path between the objects, often times rules are in reference to either the center of objects or the distance in 2D space. VOs are also ambiguous in their definition as the exact location of a corner in not definite. Therefore, the geometric properties and relational properties outlined in Table 4.2 and Table 4.3 and the VOs in Table 4.4 are only best guess solutions to the true intention of the rule. These tables were constructed to satisfy the rules for the case studies exemplified throughout this thesis. As new rules emerge, there will inevitably be rules that cannot be comprised of the existing properties, relations, and implicit objects we have defined. Thus, as the language matures these tables will be required to be expanded, however, the reusability and logical composition of these properties allows for a abundance of different rule scenarios to be constructed.

Object Types

The types used in the rule language follow types defined in the IFC schema. IFC has a predefined class hierarchy, beginning with the *IfcRoot* class and branching out to many subclasses. Although many class types are supported, such as *IfcStandardWall* and *IfcDoor*, many standard building objects often are defined by generic types such as *IfcBuildingElementProxy* and *IfcFurnishing*, as these might represent the lowest branch available that fit the types classification. For instance, a couch would fall under they type *IfcFurnishing* as no *IfcCouch* type exist. Most standard rules for interior design require checking items types more specific than furnishing. Therefore, we propose two potential solutions for further filtering objects by type and identify the key benefits and shortcomings of each.

The first is to extend the IFC class hierarchy to include new types. This would require introducing new classes and specifying the existing class that they are a subclass of, i.e., inherent from. For instance, an *IfcCouch* type could be introduced as a new type that inherits from the existing *IfcFurnishing* type. Figure A.1 illustrates some of the IFC existing types (A) along with additional types that could be introduced (B). The benefit for this method would be the ability to easily perform type filtering based on exiting intermediate types, such as *IfcFurnishing*, which encompasses many different types underneath it, or newly created intermediate types such as *IfcSeatingFurnishing*, which could be a superclass of chairs, couches, and any other object that supports seating. One drawback however, is that the new type hierarchy would need to be sent along with the model in order for another application using the rule language to be able to identify the new object types. Additionally, this would require a merging strategy for the new type hierarchy with the existing hierarchy. Having a single standardizes hierarchy would assist with these limitations.

The alternative is to keep the hierarchy as is and include the more indepth types as a property of the object. Therefore, a couch would remain as type *IfcFurnishing* but have a property that states it is a couch, either by a string property with value “couch” or a boolean property “IsCouch”. The latter would be inefficient for our model check implementation as it would result in an “IsCouch” property being created for every object when the model-check searches an instance.

As there were few cases in rules for our case studies where type grouping were advantageous, we opted for a type string property attached to each object, over the IFC hierarchy extension. The property for this method is “FucntionOfObj” as seen in Table 4.2. It uses a keyword search on the name string to determine the object type and if none is determined returning a value of “Unknown”. Therefore, there is a large onus on the names of the objects to be explicit with the type. In an ideal scenario, “FunctionOfObj” would infer and classify the object type or affordance based on the surface representation of the object. However, the existence of such a method has not been explored at this time.

4.5 Summary

Our rule language expresses basic rules as checks on the properties (i.e., height, width, etc.) of individual objects, and geometrical properties of relations between pairs of objects (i.e., centre-distance between objects, object1 being in front of object2, etc.). These checks can be further composed in complex rules using logic operations. The rules are expressed based on the following core utilities for this language.

- The relevant object filter or existential clause searches through all objects in the input model and filters them into relevant object sets. A rule instance table is created based on the cartesian product of theses sets. Virtual objects can be created and added to the model check such that they can also be search as real objects.

- The rule instance evaluation first identifies the basic property checks applicable to each (pair of) object(s), and evaluates them based on the object properties and interrelations. Then applies the logic composition operators to each of the property evaluations involved, providing a result for each rule instance.

- The final rule evaluation is determined based on the aggregation of each of the individual rule instance results, dependent on the clause (ALL/ANY/NONE) of the existential clause search.

Future work might be to further explore the necessity and feasibility of an extendable type hierarchy. Additionally, there could be a type mapping implementation, meaning rather than using IFC class names, such as *IfcStandardWall*, one could use simply "Wall". We will also continue to explore alternative rule editors for better user interaction.

The next chapter will describe the model-checking method that supports these rules and how we implement a model-checking tool for checking real models.

Chapter 5

Model Checking

Checking a BIM model against a set of design rules has been a major topic in BIM research for over a decade, and yet no broadly available solutions exist to support rules from a variety of sources, such as governing agencies, handbooks, and builders. While some model-checking software systems exist, they either require that their users possess a strong software-programming knowledge to configure them with rules of interest, or they are black boxes, not configurable at all. Since it is unlikely that all stakeholders will ever be able to agree on a single immutable set of rules, applicable to all buildings, these products are fundamentally limiting the wider adoption of automated model-checking of buildings.

Rule languages offer a means of rule creation and customization without the need for strong software-programming knowledge. While there have been previously proposed languages, such as BERA, KBim, and BIMRL, they still possess a higher level of complexity or have generally minimal support and extended uses. We believe our rule language, described in the previous chapter is a more user friendly language, as the language reads more closely to the natural language from rule codes. It also supports use beyond model-checking as we will describe in chapter 6, thus making it more broadly useable.

This chapter focuses on automated model-checking, which is defined as “software that does not modify a building design, but rather assesses a design on the basis of the configuration of objects, their relations or attributes” [11]. The chapter outlines a model-checking implementation using our proposed rule language exemplifying well-defined model-checking concepts from [11] and previous related research.

Building off the rule language outlined in the previous chapter, this chapter outlines the proposed model-checking methodology in section 5.1, followed by the current implementation in Section 5.2. Section 5.3 contains discussion points on our methodological assumptions and how the language and rules fit in the larger picture. Finally, concluding remarks are in Section 5.4.

5.1 Methodology

Rule-based model compliance checking, as outlined in [11], involves four stages: (a) rule interpretation, (b) building-model preparation, (c) rule execution, and (d) reporting of the checking results. The first step involves converting natural language design constraints into a computer-interpretable rule language, which is vital for the automation of the rule language. The second stage involves the information present in the model. Model information can be disorganized and not explicitly defined, making checking slow and tedious. Therefore, the second stage is to organize and summarize the information in the model for faster, more efficient check results. The rule execution stage is to sequentially determine the results for each rule-based on the organized model information. And finally, the user must get a report of the result, indicating the compliance evaluation of the model, as a rule-by-rule evaluation and, in addition, the information was used to determine a fault.

5.1.1 Rule Interpretation

Rule interpretation involves the creation of rules in a computer interpretable form. This includes the translation of rules from natural language text based formats or other documents. In an ideal scenario, rules could be directly interpreted using some form of Natural Language Processing (NLP) to extract the rules logic, however, this is largely infeasible due to the vagueness of the language used to describe rules. Efforts into mandating regulatory texts into machine interpretable writing have not been successful thus far [9].

Therefore, the interpretation phase in this study, as outlined in the previous chapter, is through a rule editor interface by manually interpreting the rules logic into that of the rule language. When all rules have been created, they are exported to a file which the model check application reads. The exception is object overlap, which is by default always included in the checks as there are no cases where non-virtual object overlapping is valid.

5.1.2 Model Preparation

There are two major inputs into the model check. They are the model and the ruleset. The ruleset is a file which has been exported from the rule editor which the model language can parse back into the original rule for execution. We use JSON for the ruleset file serialization and deserialization.

For this research, the model is in the form of an IFC file, and as explained in the previous chapter, the rule language utilizes the IFC types for object filtering. While other BIM schemas exist, this research opted to be IFC based as it is the primary open standard and many BIM tools have IFC exporting capabilities.

Model Check Preparation

Data in IFC is structured in a highly complex manner as an objects' surface representation can take the form of extruded solid, Boundary Representation (BREP), or their combinations. This implies that, before the rules can be evaluated, the BIM data must first be transformed into structural objects that support efficient geometric calculations. Similar to [41], our method parses the model into an internal object structure that includes a global triangulated mesh, a local triangulated mesh, and a global bounding box that contains the direction and dimensions of the object in 3D space; a mesh being a series of vertices grouped into sets of three forming triangular boundary faces. The global bounding box is typically used for the geometric calculations as it reduces the computational speed of the model check drastically at the cost of lower accuracy. For a global mesh, the vertices are defined in global coordinates relative to the whole building model, while local meshes are relative to the coordinate system of the object. In a local coordinate system the X represents Left to right, the Y represents forwards and backwards, and the Z represents up and down. Property checks are performed on the local mesh of an object while relation checks are performed on the global meshes of the involved objects.

Each object whose type is nested under *IfcProduct* contains a surface representation. Therefore, from the input model, every object with type nested under *IfcProduct* is extracted, has its mesh representation parsed into the model-check internal mesh representation, and added to the set of objects that can be checked.

Algorithm 2 outlines the initialization of the model check process, taking in the model file, converting the model objects that are subtypes of *IfcProduct* to structures objects (line 4) and adding them to the list of model objects that can be searched (line 5).

Algorithm 2: Algorithm for initializing the model-check.

```
1 Function InitializeModelCheckObjects(IfcModel):
      Input:
          IfcModel
      Output:
          ModelObjects
2     ModelObjects ← {}
3     foreach IfcProduct ∈ IfcModel do
4         Obj ← Convert(IfcProduct)
5         ModelObjects.Add(Obj)
6     end
```

Included in the model-check initialization is also the ruleset preparation, which comprises of removing rules that might be semantically invalid and pre-compiling the rules the generic programming language. Algorithm 3 outlines the validation and compiling of rules. Rule validation (line 4) includes checks that might always result in true or false. Currently the only validations we have is whether a rule has at least one existential clause and one logical expression and to check that the index of the property check and relation check are in the range of 0 to the number of existential clauses, since this must correspond to a column number in the rule instance table. The rule generic programming language code must be generated from the structured rule syntax (line 8) and then compiled (line 9). These compiled rules can then be executed by the model-check.

Algorithm 3: Algorithm for validating and pre-compiling the rules.

```
1  Function ModelCheckRulePrep($Ruleset$):
       Input:
           $Ruleset$
       Output:
           $CompiledRules$
2      $CompiledRules \leftarrow \{\}$
3      foreach $Rule \in Rulesets$ do
4          $RuleValid \leftarrow$ Valid($Rule$)
5          if $RuleValid == FALSE$ then
6              $Skip$
7          end
8          $RuleCode \leftarrow$ GenerateRuleCode($Rule$)
9          $CompiledRule \leftarrow$ Compile($RuleCode$)
10         $CompiledRules.Add(CompiledRule)$
11     end
```

Pre-check Model Enhancement

Once all the IFC objects have been read, our method constructs and adds to the model several different types of Virtual Objects (VOs), based on whether or not they are required for the input ruleset.

To reiterate, VOs are implicit model objects generally representing complex, multi-object relationships. By this definition, some VO types are already included in the IFC vocabulary, such as *IfcSpace* and *IfcSite* for example, which are nested under *IfcSpatialElement*. As VOs by definition also have geometric bounds, we represent them internally in the same category as *IfcSpatialElements*. Given that they also are nested under *IfcProducts*, *IfcSpatialElements* have a surface representation and thus existing the *IfcSpatialElements* are reconstructed into the internal shape representation when the model is read. After the model is read, all VO that are required by the input rules are created and stored with the existing model objects.

While both nested under *IfcProduct*, the major difference between the *IfcSpatialElements* and *IfcElements* is that *IfcSpatialElements*, such as *IfcBuilding* or *IfcSpace* are typically created based on *IfcElements* thus depending on them in order to create their own representations, whereas *IfcElements* have no strict dependence relations. If an *IfcElements* object is updated, any *IfcSpatialElements* that were created based on that object would require recalculation. While VOs created using the rule language are not saved to the model after creation, *IfcSpatialElements* should be checked and recalculated if they do not properly encompass their composed *IfcElements*. [7] proposes one such method for checking the validity of these spaces in IFC.

This process is outline in Algorithm 4. The algorithm takes in the rules and iterates over each one (line 4) and each of their existential clauses (line 5). First it must check that it is a virtual object and if so that it has not been created already to ensure no duplicate virtual objects are created (line 6). The new virtual objects are then added to a list of new objects (line 10) and once all have been created, they are added to the list of model objects (line 14).

Intra-check Model Enhancement

Relations do not function in the same way as model objects. Fundamentally they are relationship property storage items defined by their object set, having no geometric shape representation. Therefore, they are not stored in the model objects set but rather in a separate model relation set at runtime. While IFC does contain a *IfcRelation* type, they are not strictly geometrically based. As a result, we only store relations in run-time data.

For the implementation, we set the object set size of a relation to two, as this covers the large majority of relational properties that are checked; VOs would be added instead for relations that exceed two objects. Additionally, relations are directional, meaning a relation of object1 and object2 is separate from the relation object2 and object1. This is due to the nature of the properties, such as "InFrontOf" where the order of the objects matter.

While all possible relations, corresponding to all possible pairs of objects, and all the relationship property data could potentially be added in the ini-

Algorithm 4: Algorithm for adding virtual objects to the set of model objects.

```
1  Function PreModelCheckEnhance($ModelObjects$):
       Input:
           $ModelObjects$
       Output:
           $ModelObjects$
2      $AddedVOTypes \leftarrow \{\}$
3      $VOs \leftarrow \{\}$
4      foreach $Rule \in Rulesets$ do
5          foreach $EC \in Rule.ExistentialClauses$ do
6              if $IfcTypes.Contains(EC.Type) \vee$
                $AddedVOTypes.Contains(EC.Type)$ then
7                  $Skip$
8              end
9              $NewVOs \leftarrow \text{CreateVO}(EC.Type, ModelObjects)$
10             $VOs.Add(newVOs)$
11             $AddedVOTypes.Add(EC.Type)$
12         end
13     end
14     $ModelObjects.Add(VOs)$
```

tialization phase of the model check, for computational efficiency, we decided to embed the creation of object relations and property calculations in the execution phase.

As an example, the distance between two objects is not calculated unless it is necessary for a rule. When a rule calls for the distance of two object instances, it first checks the model relations for a relation where the first object matches the first object in the relation and the second object matches the second object in the relation. If none exists, a new relation is created. Then the specific distance function required by the rule is called and executed with the two object instances as parameters and a value is returned. The function name and the return value make up a new property that is added to the relation. On the other hand, if a matching relation is found, then the existing properties of the relation are checked for a property that matches the distance function name. If one is found then that properties value is used, if not then it must be calculated as before. Therefore, properties are cached and can be reused as necessary, until the complete set of rules has been evaluated and the building model-checking is complete.

This happens during the rule instance evaluations, as that is the stage in the model-check when the individual property and relation checks are performed. In Algorithm 6 these are the line 5 for property checks and lines 12 and 13 for the relation check.

5.1.3 Execution

Algorithm 5 outlines the execution process, which is referred to in section 4.2 as the Rule Evaluation. The method takes as input the model objects and compiled rules. For each of the rules, an instance table is created (line 5) in the form of Figure 4.1 (5) using the cartesian product of the relevant object sets. However, initially the rule instance result is empty as it has not been evaluated. Thus, for each row in the rule instance table (line 6) the rule instance result, which is the last row in the table, is calculated by the rule instance evaluate method (line 7), outlined in Algorithm 6. Then, the full rule result is evaluated (line 9) using the method in Algorithm 1. The return of the execution is the list of all rules with the corresponding evaluation result of the rule and the table of rule instances.

The detailed pseudocode for the rule instance evaluation can be seen in Algorithm 6. This takes in the rules instance, which is the row in the table which contains a ordered list of objects, the relation list for checking existing relations, and the logical expression that will evaluate the rule instance. First, the evaluation goes through each property check contained in the logical expression (line 3), and for each one, gets the object at the index of the checks existential clause index (line 4). Next, it searches for the property in the objects property set; if it finds it then it uses that property, if not it must create the property and calculate its value (line 5). The algorithm proceeds to compares that value against the property check to determine the check value (line 6) and adds that checks result to the list of all check results (line 7). In a similar fashion, all of the relations in the logical expression are checked, this time using two objects (lines 10 and 11) and first checking if the relation exists

Algorithm 5: Algorithm for executing the Model Check.

```
1  Function ExecuteModelCheck(ModelObjects, CompiledRules):
       Input:
           $ModelObjects$
           $CompiledRules$
       Output:
           $RuleResults$
2      $RuleResults \leftarrow \{\}$
3      $ModelRelations \leftarrow \{\}$
4      foreach $Rule \in CompiledRules$ do
5          $RITable \leftarrow \text{CreateRI}(ModelObjects, Rule.EC)$
6          foreach $RInst_i \in RITable.Rows$ do
7              $RITable[i, end] \leftarrow$
                 $\text{RuleInstanceEval}(RInst_i, ModelRelations, Rule.LE)$
8          end
9          $RuleResult \leftarrow \text{EvaluateRule}(RITable, Rule.EC)$
10         $Ruleresults.Add(Rule, RuleResult, RITable)$
11     end
```

in the model relations set. If so, it uses that relation, otherwise it creates a new one and adds it to the model relation set (line 12). As with the property check, it searches the relation for a property that matches the relation check property. If it exists, the program will use that property, if not then it will create a new one, calculate that value and return it (line 13). The algorithm compares the value against the check value (line 14) and adds it to the check results (line 15). Finally, all nested logical expression must be calculated, calling the rule instance evaluation recursively (line 18). The result of all nested logical expression are added the the check set (line 19). Finally, the result of the logical expression is the aggregation of all the checks using the logical operator of the expression.

5.1.4 Reporting

Finally, each result of the rules is relayed back to the end user or application. In addition to pass/fail value, we return a list of all instances of the rule along with the rule for reference. The reports may also be narrowed down to only the failed rule instances as these are typically more important to know from the

Algorithm 6: Algorithm for evaluating a rule instance.

```
1  Function RuleInstanceEval(RInstObjs, ModelRelations, LogicExpr):
       Input:
           $RInstObjs$
           $ModelRelations$
           $LogicExpr$
       Output:
           $RuleInstanceResult$
2      $CheckResults \leftarrow \{\}$
3      foreach $PropCheck \in LogicExpr.PropChecks$ do
4          $CheckObj \leftarrow RInstObjs[PropCheck.EcIndex]$
5          $PropVal \leftarrow$
            FindOrCreateObjProp($CheckObj, PropCheck.FuncName$)
6          $PCResult \leftarrow$
            Compare($PropVal, PropCheck.Op, PropCheck.Value$)
7          $CheckResults.Add(PCResult)$
8      end
9      foreach $RelCheck \in LogicExpr.RelChecks$ do
10         $RelObj1 \leftarrow RInstObjs[RelCheck.EcIndex1]$
11         $RelObj2 \leftarrow RInstObjs[RelCheck.EcIndex2]$
12         $Rel \leftarrow$ FindOrCreateRel($ModelRelations, RelObj1, RelObj2$)
13         $PropVal \leftarrow$ FindOrCreateRelProp($Rel, RelCheck.FuncName$)
14         $RCResult \leftarrow$
            Compare($PropVal, RelCheck.Op, RelCheck.Value$)
15         $CheckResults.Add(RCResult)$
16     end
17     foreach $LogicExprNest \in LogicExpr.NestLE$ do
18         $LEResult \leftarrow$
            RuleInstanceEval($RInstObjs, ModelRelations, LogicExprNest$)
19         $CheckResults.Add(LEResult)$
20     end
21     $RuleInstanceResult \leftarrow$ LEEval($LogicExpr.Op, CheckResults$)
```

users perspective. The client application can parse the result information and graphically display the objects that failed the rule and the rule information such as the error level of the rule and description.

5.2 Implementation

As validation for the proposed research, we have created a prototype model-checking .NET WindowsForm application. This application takes in a IFC model file and a ruleset JSON file which can be exported from the Rule Editor application.

Additionally, the model check application contains a public method library by which another .NET application can call the model checking methods. To demonstrate this, we used a previously built in-house Unity-based IFC model editor to test and visualize the results. The editor calls the model checking method by passing along the edited IFC model and selecting the rule set exported from the rule language editor. We have developed a library of basic functionalities for computation of VOs and numerous geometric functions (i.e., for calculating object height and width, and distance between objects) which can be seen in Tables 4.2, 4.3, and 4.4. Both the IFC editor and the Model Checker utilize our in house IFC Engine package for parsing models and extracting the simplified geometry from IFC model objects.

After execution, the model-checking method returns the results for each of the rules. A rule result includes a pass/fail boolean, the rule for reference which contains a description of the rule, and each rule instance and the result of that instance. Each of these result instances can be selected and the corresponding objects and rule information will be highlighted and displayed respectively as seen in Figure 5.1. Currently, the report does not include the property and relation values used by the rule instances, however, if they were saved to the model, they could be accessible and referenced in the report. The VO meshes are stored and accessible and can optionally be created and displayed in the IFC editor, however they are also not saved to the model.

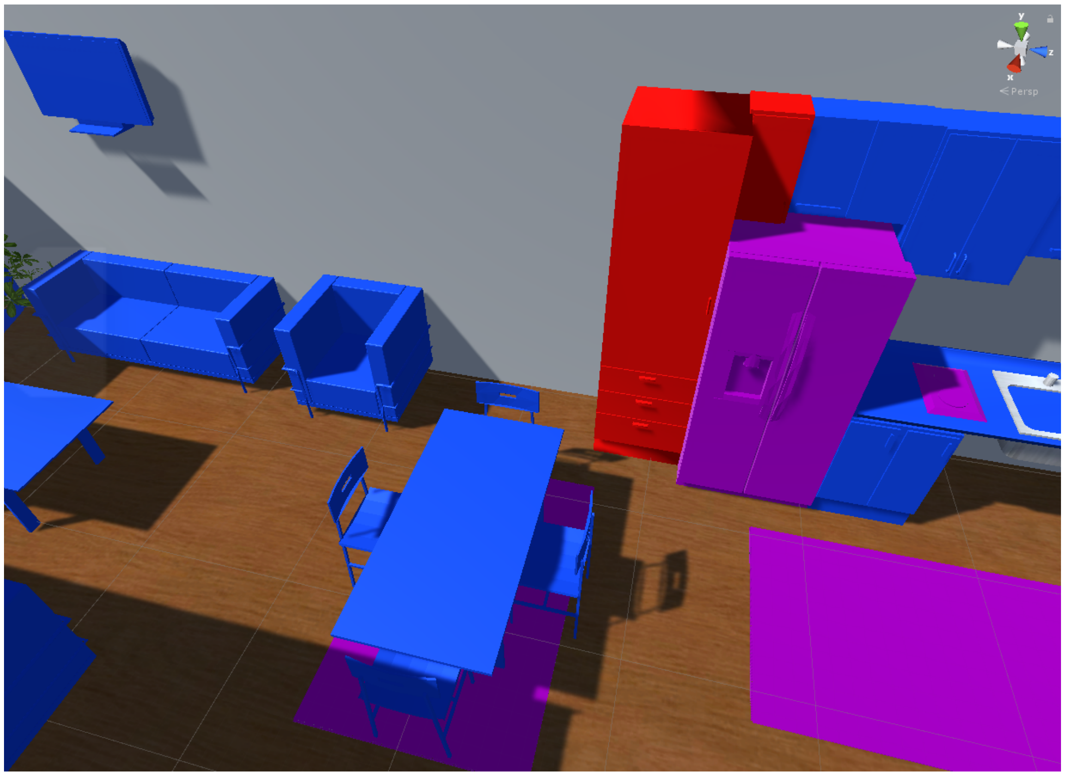


Figure 5.1: Unity interface highlighting objects red.

5.3 Discussion

Archiving Implicit Computations An interesting challenge is how to archive the computations performed, i.e., the VOs and the relations among objects, in support of the complete model-checking process. In principle, there are two choices: (a) they may be saved with the building model itself, or (b) they may be saved in a separate data structure but with references to the building model. Should the VOs and properties be saved to the model, it would be necessary to develop a management process to remove the results of individual rule evaluations as the objects to which the rules apply are modified. For instance, if the “Distance” relation property was calculated but the dishwasher has been moved in the new model version, then the original “Distance” property should be removed and recalculated if required.

If the building model editor is capable of flagging the objects that have been modified since the last model check, the model check could recalculate the VOs and properties that depend on those modified objects. This would theoretically expedite the subsequent model checks. The safer, more conservative, choice is to assume the building model has not been checked previously and that all VOs and properties must be newly calculated and, if existing, overwritten by the new values. This is the current practice in our prototype, however, we are currently investigating the most efficient way to save and flag changes in our editor.

Object Type Issues For the purpose of this study, we used the object name to determine automatically an objects classification, however, the onus is on a more intelligent BIM editor to infer the most specific type of the object, beyond *IfcElement*. For instance, many objects created in CAD environments then imported into Revit are placed in the Generic Family type. When these types are exported they are then classified as *IfcBuildingElementProxy* elements, as the exact type is unknown by Revit, (unless specifically created in or placed in another Family type). It is also imperative that objects do not fall under multiple categories or are compositions of multiple other objects. For instance, difficulties can arise when a collection of objects, such as multiple chairs surrounding a table, are modeled as a single object. This can make checking table to chair distances problematic. Therefore, good modeling practices should be adhered to the largest extent possible.

Standard Modeling Practices Other issues encountered included the orientation of the objects not always being standardized in IFC, or at least by the BIM editors that export the models. This happens when the front face of an object is not in line with the intended front of the object, due to an inconsistent local coordinate system for each object. The result is relationship properties such as behind and in front of, which appear frequently in our rules, return unintentional values. We see this as an error in the objects design, since from an end user viewing the model, this error may not be visually apparent when placing the object.

We also acknowledge that, while many other subtypes exist, for the purpose of this thesis, *IfcElement* and *IfcSpatialElement* make up the two notable branches of the IfcProduct type. It could be possible to utilize other branches of the *IfcRoot* class such as the *IfcRelationship* types, however, in practice we found they are often not consistent in their usage among software vendors that export to IFC. Further exploring the potential usage of additional IFC types is left to future work.

5.4 Summary

In this chapter, we have outlined a model-checking method able to evaluate rules (in the form of the rule language from the previous chapter) on IFC building models. The model-checking is separated into four key stages and we outlined the workflow for each. Furthermore, we described an implementation example connecting the rule editor, model editor, and model-checking application and the data flow between them. Future work includes intelligent implicit data saving for expediting subsequent checks and exploring the usage of other IFC types.

The next chapter will further utilize and expand upon the model-check method from this chapter for the purpose of generative design.

Chapter 6

Generative Design

Optimizing interior layout design can be a time-consuming task. This is because there are a number of design constraints that must be met in addition to the preference towards more visually appealing designs for the end consumer. The challenge of striking a balance between an interior space that complies with ergonomic and stylistic rules and guidelines and satisfies the consumer's preferences is exacerbated by the large number of potential placements, orientations, and combinations of objects that the end consumer would like to place in the potential space.

Computationally generating and testing a number of design alternative can potentially expedite this process as it makes use of fast and efficient evaluations of layout alternatives, based on computationally specified and evaluated design guidelines. Therefore, the designer need only to provide the overall space layout as the starting point for the design generation, select from a catalog a set of objects that should be laid out in this space, and specify the design guidelines that the final layout configuration should meet. The system would generate a number of alternative designs, which can subsequently be compared and contrasted to one another based on the basis of other metrics, such as cost for example, or shown as 3D models to the consumer to select the design they prefer, knowing that the final design adheres to the selection of required design constraints.

The key difficulty in developing such a generative system lies in the number and complexity of possible design solutions, which implies that performing an exhaustive search of the solution space is not feasible for this type of problem. Secondly, the design guidelines themselves must be expressed in such a way that they can be computationally evaluated. The specification of design rules in manuals and documents is often ambiguous, making it difficult to evaluate them and conclusively assert whether they "pass" or "fail". The generative-design method presented in this chapter relies on our design model checking based on a domain-specific language for capturing design rules from the previous chapters. The model checking serves as the evaluation of the object placement in the space and to prioritize positive design additions. The candidate object placements are determined based on a grid, configured in the input space and the design search continues until a compliant layout is created.

This chapter is organized as follows. Section 6.1 outlines the generative design methodology. Section 6.2 describes the implementation and method evaluation. The evaluation consists of a comparison of real world kitchen layouts with our generated layouts and the implementation comparison of living room guidelines from previous literature against our rule language. We reflect on the findings of our evaluation in Section 6.3 and conclude with a summary of our contributions and our plans for future work in Section 6.4.

6.1 Methodology

According to [26], the generative-design process requires (a) a *Performance Metric* to impose an order to the solutions in terms of their quality; (b) a *Configuration Variation* method to systematically change the design configurations and their quality; and (c) a *Decision-Making Response* to determine, based on the evaluated configuration variations and corresponding performance metrics, the next design configuration which trends towards improved design solutions.

6.1.1 Performance Metric

Our method proposes a rule-based compliance checking as a quality-evaluation, at each stage in the generative design. We use the model-checking described in the previous chapter, however, with a few alterations to the four stages involved.

The first stage, the rule interpretation, involves converting natural language design constraints into a computer-interpretable rule language. As the rules are the same throughout the generative design, this stage is a one-time execution before the generative design begins and thus the generative design uses the same compiled rules throughout.

The model preparation stage is largely dependent on the generative design method, specifically on the changes to the model at each step in the generative design. Therefore, all static model objects do not require any property recalculation, however, if an object is added or moved, then all relations of the object to any other object must be recalculated. Furthermore, as virtual objects are relative to the whole model, virtual objects must be re-created at each step in the generative design, to ensure it is up-to-date with the model changes.

As this model-check is performed in the context of generative design, which requires a large number of model modifications and evaluations to be performed, we reduce the shape representation complexity for each object by replacing it with a bounding box object. These are created by determining the X, Y, Z dimensions of the shape in the shape's local coordinate system and creating an encompassing box. When generative design completes and the final model is constructed, the box can be replaced back with the true shape representation. Virtual objects, however, retain their shape representation rather than a bounding box.

The third stage, the execution stage, iterates over each rule and determines the result of the rule. While for model-checking, returning a value of true or false is sufficient, for generative design it is advantageous to scale the result of the rule. Providing a numeric value can determine whether a modification in the design configuration is better or worse than another, regardless if the rule fails. This provides a gradient on the configuration search space which a generative design method could use to find the local maximum corresponding to more compliant solutions. While boolean properties and string properties could only provide a true of false value, which would correspond to a value of

1 or 0 respectively, the numeric property check could be formulated to return a result in the range of [0, 1], with 1 being full rule compliance and any value less than 1 being a degree of rule violation. Using the formulation inspired by [29], the numeric property checks return a value using the function:

$$GreaterThan(d, v, \alpha) = \begin{cases} (\frac{d}{v})^{\alpha}, & \text{if } d < v \\ 1, & \text{if } d \geq v \end{cases}$$

$$LessThan(d, v, \alpha) = \begin{cases} (\frac{v}{d})^{\alpha}, & \text{if } d > v \\ 1, & \text{if } d \leq v \end{cases}$$

$$Equal(d, v, \alpha) = \begin{cases} (\frac{d}{v})^{\alpha}, & \text{if } d < v \\ 1, & \text{if } d = v \\ (\frac{v}{d})^{\alpha}, & \text{if } d > v \end{cases}$$

$$NotEqual(d, v, \alpha) = \begin{cases} 0, & \text{if } d = v \\ 1, & \text{if } d \neq v \end{cases}$$

Where d is the property value, v is the check value, and α is the "degree of attraction" [29]. The formulation of the functions provides a means of scaling the rule result values, such that values farther from the check value are always decreasing, yet always remain above 0. Larger α values result in a sharper decline for values not equal to the check values. In practice, the α need only be greater than 1, however, any value too large could result in rule result values to small for floating point precision. Therefore, for the implementation an α value of 2 was sufficient.

In the special case that $v = 0$, we use a value of $v = 0.00001$ since for $v = 0$, v/d would always return a result of 0, giving no information on the severity of the failure.

Using this formulation for the numeric property checks, and 0 and 1 for the boolean and string property checks, the calculations for the logical expression can then be converted to numeric return values. This is done by substituting the AND logic for boolean values to multiplication of the numeric values (A AND $B = A * B$). The numeric equivalent for the OR operation would be to take the maximum value of each of the check results (A OR $B = max(A, B)$). A XOR B can be expressed as (A OR B) AND NOT (A AND B), thus numeric equivalent for XOR would be A XOR $B = max(A, B) * (1 - A * B)$ (associatively using the first two value result against the next etc.). Any negation would result in 1 minus the value (NOT $A = 1 - A$). The same logic is applied to the ALL/ANY/NONE clause operations for the existential clause.

Finally, the reporting stage returns the final quality assessment of the model, based on an aggregation of each of the individual results of the rules. This gives the evaluation for the model state, which is necessary for any generative design algorithm to compare intermediary design decisions so that the generation trends towards the optimal solution.

6.1.2 Configuration Variation

Our generative-design method requires as input a BIM model of an empty space, a set of objects that should be placed in the space, and a set of input design rules. This differs from the majority of previous automated design methods which start with a populated model and only evolve the model into better designs. Our approach is scene population where the final design can have a range of possible objects so long as the design satisfies all rule requirements. Therefore, the process ends with a new BIM model that is based on the input model and also includes (some of) the input objects, placed in a manner that conforms to the input rules.

Our method generates new model configurations by (a) selecting which object to add to the model, (b) deciding the 3D location where the object should be placed, and (c) selecting the orientation of the objects. The process is initialized with the configuration of a grid of possible object-placement locations, based on the input model. One can potentially imagine different ways for configuring such a grid.

In our method, the grid is determined based on the walls in the model, where a variable number of lines are created that run parallel to the wall center line. The grid location points are points on each of these lines, starting from the walls start point to the wall end point. The distances between each point is based on increments of a standard step, established as the greatest common denominator of the widths of the to-be-placed objects. Each of the grid points affords placement of an object in orientations perpendicular and parallel to the walls bounding the space.

For the test case of the kitchen, only one line parallel to each wall is used as we only considered L and U shape kitchens without kitchen islands. For the living rooms we use a sufficient number of lines such that the whole floor was covered. In both cases, grid points that were not placed on a floor or were located under an existing furnishing item were removed. In addition, we removed points that were close to already existing grid points if there distance was bellow a specified threshold. Each time an item was placed in the model, the grid points underneath the new item would be removed.

6.1.3 Decision-Making Response

To determine whether a valid model has been generated, and thus should be added to the valid solution set, is to check the model evaluation, or model-check. If all rules are fully satisfied, then the model is added to the solutions. However, as there is the possibility that rules are conflicting, or simply impossible given the initial model and object set, the "best-so-far" solution, based on the evaluation value, is always temporarily stored.

The algorithm is shown in pseudocode in Algorithm 7. The algorithm is a heuristic search, that takes as input the model, the set of n model elements that can be added, and the ruleset the model must comply with. First, the algorithm must configure the placement grid and the possible rotations of the model objects based on the walls in the model (lines 2 and 3). Then for each combination of object, location, and orientation (configuration variation) (line 7-9), a model configuration is created (line 10), evaluated based on the quality estimate produced by the model-checking evaluation process (performance metric) (line 11), and then ranked (line 12) based on the quality evaluation. The best model configuration, which is an object placement and orientation that have the highest quality based on the model-check, is then selected as the basis for a new round of the process (decision-making response). If there is a tie for the best model configuration, then one is randomly selected (line 16).

Solutions are output when a fully compliant layout is generated or passes the acceptable compliance threshold (lines 17-20). However, if additional objects can still be placed, the process will repeat, using the model with the best configuration (line 21) and testing the remaining objects (line 22) on an updated grid, removing locations underneath the newly placed object (line 23). Therefore, it is possible that each solution in the final compliant solution set can have varying numbers of objects. The algorithm can be called multiple times, giving potentially different solution due to the random best selection at line 16.

6.2 Evaluation

We implemented the generative design as a standalone .NET application, which reads in an IFC model and creates models populated with furnishing objects selected from an object catalog, such that they are compliant with a selected ruleset file exported from the Rule Editor. The generative design application calls the model-check evaluation method from the model-check library. The solutions, i.e., the input model configurations updated to include (some of) the input objects in design-rule compliant placements, are exported incrementally as they are created giving the user the option to terminate the process after they have received "enough" valid Solutions.

To evaluate the generative design methodology, we test our method for two room cases. In the first case we tested kitchen design using initial layouts based on real-world building layouts, the goal being to see the similarity between our results and the existing results. The second test case was a comparison against work form a previous literature on living room layout generation, interpreting there mathematical layout formulas into our language to determine if the interpreted rules can create living rooms that are meet these guidelines.

6.2.1 Kitchen Layout

The first evaluation task is to compare the generated kitchen designs from our system against real world kitchen designs. For this case study, a set of design

Algorithm 7: Generative design algorithm.

```
Input:
    $IFC_{in}$
    $Sol_n \leftarrow \{el_1, el_2, \ldots el_n\}$
    $Ruleset$
Output:
    $IFC_{out}$
BEGIN
$GP_{x,y} \leftarrow$ CreateGridPoints($IFC_{in}$)
$rotations \leftarrow$ DetermineRotations($IFC_{in}$)
$IFC_{out} \leftarrow \{\}$
repeat
    $IFC_{ord} \leftarrow \{\}$
    for $i \leftarrow 1..n$ do
        for $gp \in GP_{x,y}$ do
            for $r \in rotations$ do
                $IFC_{in}^{i,gp,r} \leftarrow$ Place($IFC_{in}, el_i, gp, r$)
                $Q \leftarrow$ Evaluate($IFC_{in}^{i,gp,r}, Ruleset$)
                $IFC_{ord}$.SortInsert($IFC_{in}^{i,gp,r}, Q$)
            end
        end
    end
    $IFC_{best} \leftarrow$ BestSelect($IFC_{ord}$)
    if $Q(IFC_{best}, Ruleset) \geq Q_{Threshold}$ then
        $IFC_{out}.Add(IFC_{best})$
        Export($IFC_{best}$)
    end
    $IFC_{in} \leftarrow IFC_{best}$
    $Sol \leftarrow (Sol_n - \{el_i\})$
    $GP_{x,y}$ = UpdateGridPoints($IFC_{in}, GP_{x,y}$)
until $Sol.Count == 0$

END
```

rules were provided by an industry partner. Each rule uses a combination of properties and VOs from Tables 4.2, 4.3, and 4.4. These rules were translated manually into our rule language, using the rule language editor (see Table A.1). Also provided were a series of standard object sizes from which 3D kitchen objects, such as sinks, cabinets, ranges, were created in addition to objects from Revit's standard families package.

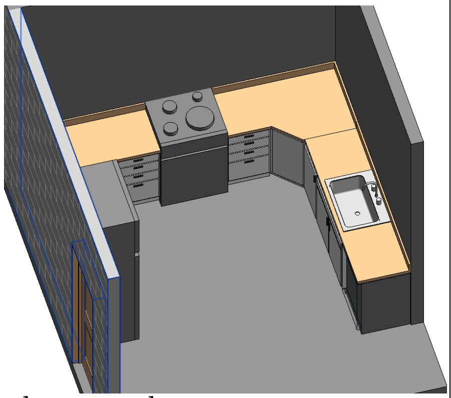
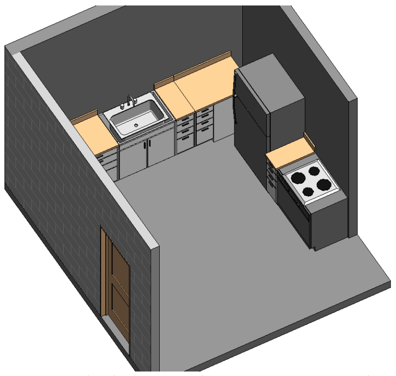
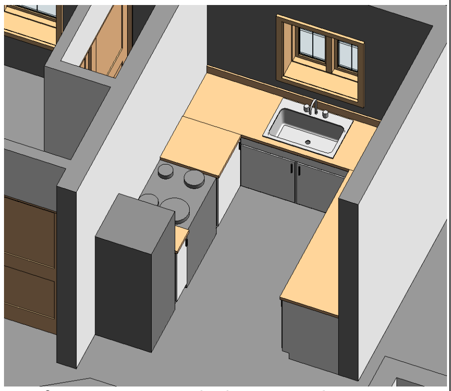
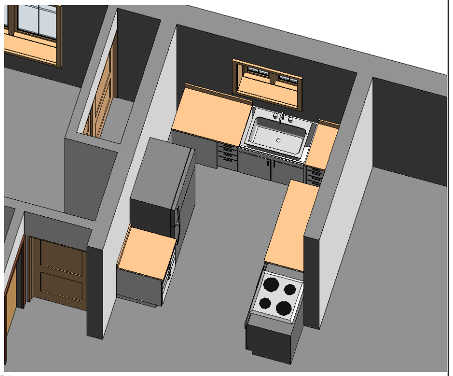
For initial layouts, a number of models were created based on show-home layouts from the web site of a local building manufacturer. These initial layouts contained finished kitchens in 2D, which we recreated in 3D using Revit which can be seen in the first column of Table 6.1.

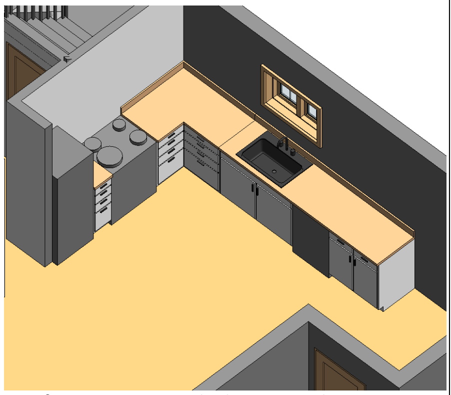
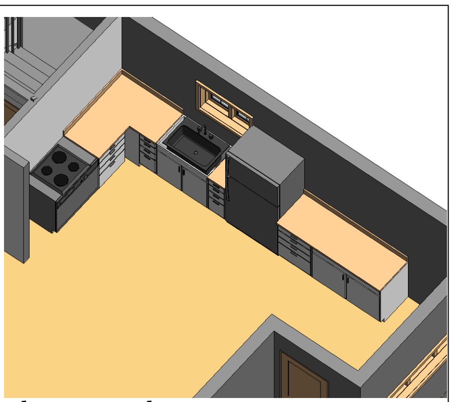
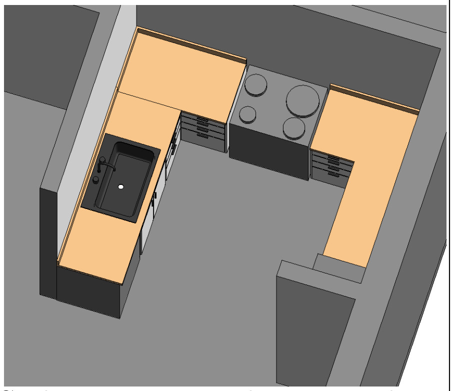
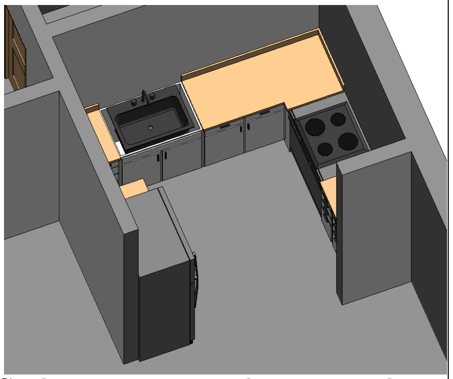
For each of theses real-world layouts, we performed a model check to assess their compliance to the rules from our industry partner. Results of the model-check on the initial layouts are outlined underneath each image in the first column of Table 6.1. Only two of the models were fully compliant (1(a) and 9(a)) with the rules. In seven cases the model-check determined the triangle rule was not compliant (2(a), 3(a), 5(a), 6(a), 7(a), 8(a), and 10(a)) with 6(a) being more severe in that the total triangle edge length was too large. The other major design flaw was the lack of countertop space next to the refrigerators (2(a), 3(a), and 5(a)). In total there were 12 rule violation instances in the initial layouts.

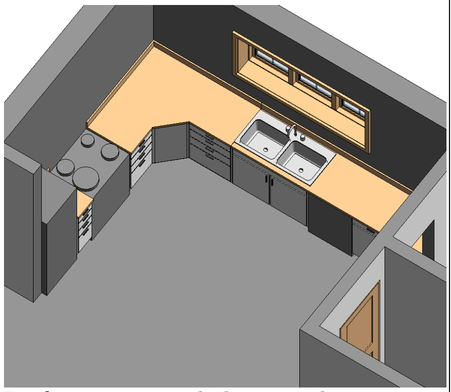
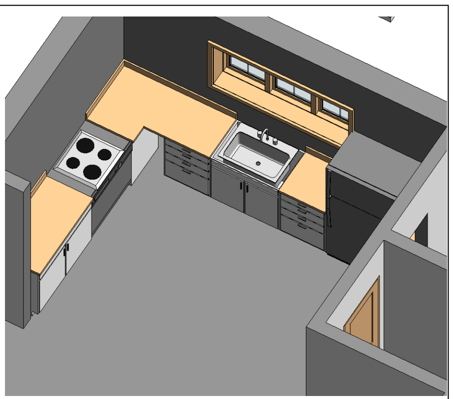
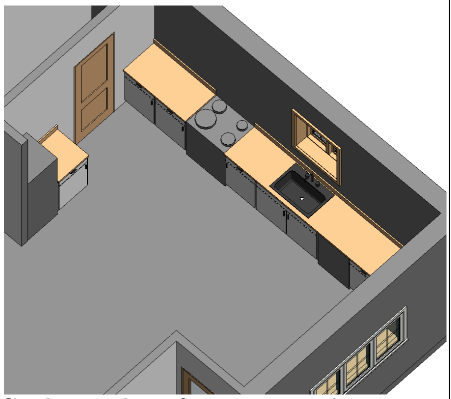
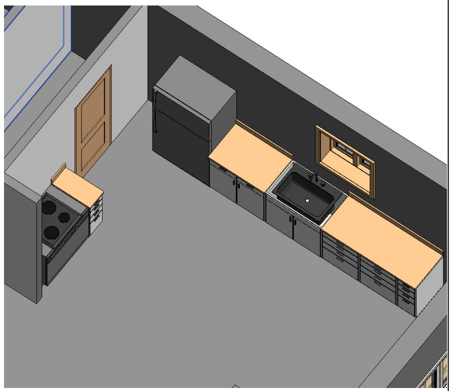
From the selected layouts, the kitchen rooms were isolated to reduce the layout search space. For each layout, a kitchen configuration was generated using the proposed generative design method with objects sharing similar features (such as function and dimensions) as those in the layout. The initial kitchen objects were removed from the building model leaving only the walls, windows, doors, and the floors. The generative-design service was invoked with this empty model as input, the rules defined in out rule language, and the selection of objects to be placed. Based on the empty model, the grid layout determined how many possible configurations there were for each object. As the model objects each had a width that was a multiple of 12in, this was determined as the spacing amount for the grid. Orientations configurations were 0, 90, 180, 270 degrees for each.

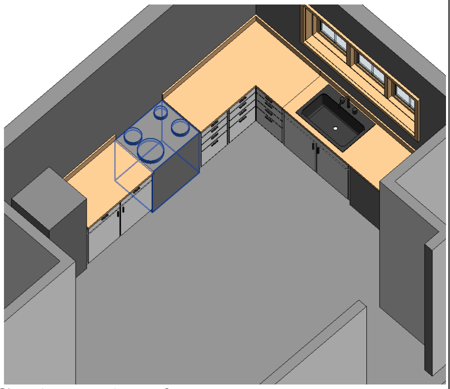
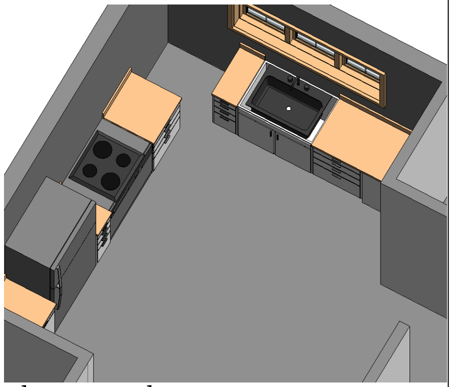
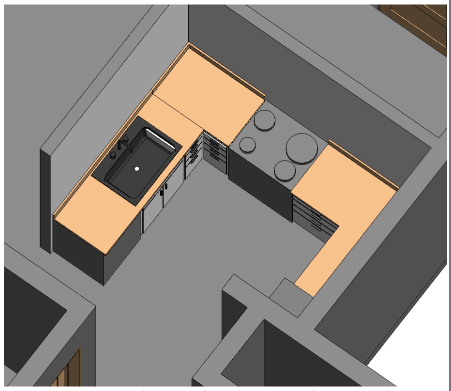
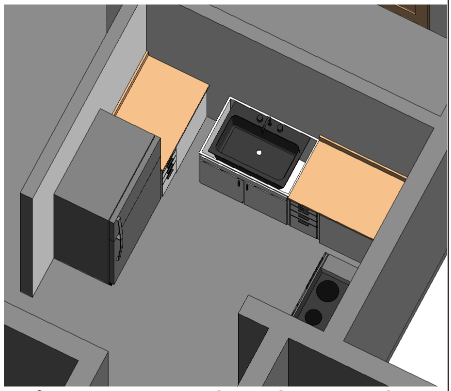
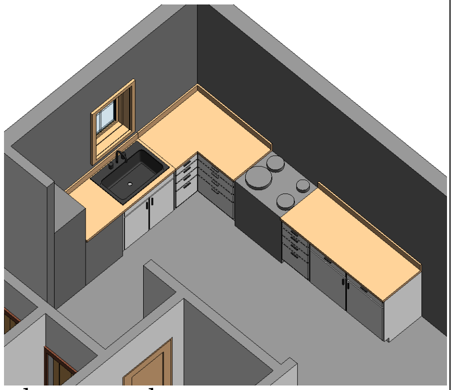
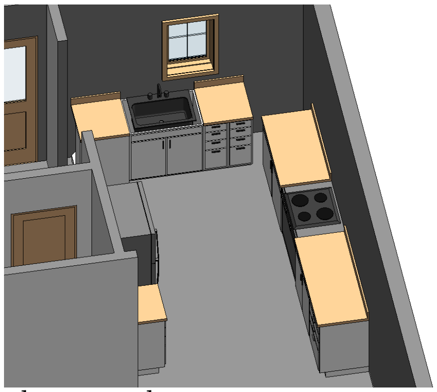
As each test model was run, the best solution thus far, based on the rule check evaluation, was continuously stored. After completion, the best model was given a final evaluation, which can be seen in the second column of Table 6.1 with the model check evaluation of the best model provided underneath.

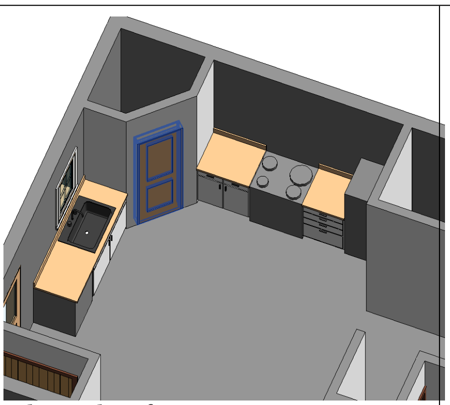
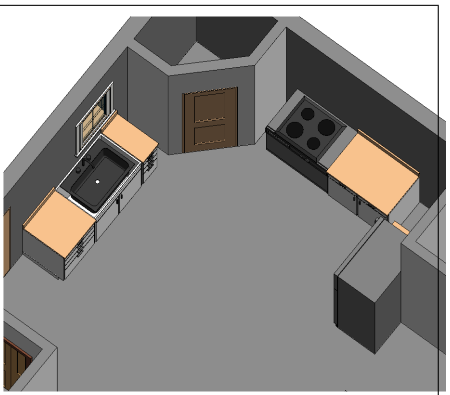
Table 6.1: Comparison of initial kitchen layouts against generated kitchen layouts.

Original Layouts	Generated Layouts
1(a)  All rules passed	1(b)  (1) Range did not have 15 inches countertop space next to it
2(a)  (1) Refrigerator did not have 16 inches of countertop space next to it (2) Range and refrigerator distance was less than 4ft	2(b)  (1) Sink did not have the required 48” free space in front of it

3(a)  (1) Refrigerator did not have 16 inches of countertop space next to it (2) Range and refrigerator distance was less than 4ft	3(b)  All rules passed
4(a)  (1) Sink was not under a window (although no windows were present in the kitchen space)	4(b)  (1) Sink was not under a window (although no windows were present in the kitchen space) (2) Sink did not have the required 48" free space in front of it

5(a)  (1) Refrigerator did not have 16 inches of countertop space next to it (2) Sink and refrigerator distance was greater than 9ft and range and refrigerator distance was less than 4ft (both triangle rule)	5(b)  (1) Range and refrigerator were over 9ft from each other
6(a)  (1) Sink and refrigerator distance was greater than 9ft and range and refrigerator distance was greater than 9ft (both triangle rule) (2) Triangle edge sum was greater than 26 ft	6(b)  (1) Range and sink were over 9ft from each other

7(a)  (1) Sink and refrigerator were over 9ft from each other	7(b)  All rules passed
8(a)  (1) Range and sink were less than 4ft from each other	8(b)  (1) Refrigerator and sink were less than 4ft from each other (2) Triangle edge sum was less than 13 ft
9(a)  All rules passed	9(b)  All rules passed

10(a)	10(b)
	
(1) Sink and refrigerator were more than 9ft from eachother	(1) Sink and refrigerator were more than 9ft from eachother (2) Refrigerator did not have 16 inches of countertop space next to it

Our method produced fully compliant solutions for three of the 10 models. In four cases (1(b), 2(b), 5(b), and 6(b)) the generated solutions violated one rule. In three models the kitchen triangle was violated (5(b),6(b), 8(b), and 10(b)), with 8(b) being more severe in that the total triangle edge length was too small. Model 1(b) had the issue of not having enough countertop space next to the range. In 2(b) and 4(b) the sinks did not have 48 inches of free space in front of them. Model 10(b) did not have enough refrigerator countertop space next to it. The total number of violations in all generated models was 10.

For both the initial model and the generated model, model 4 had a rule violation for sinks not being placed under windows, however, as there were no windows in the kitchen room, it was infeasible for this rule to be satisfied.

While the generated models did have fewer errors and thus were compliant with the kitchen design rules, in some cases they were not aesthetically pleasing in that there were gaps between some of the furnishing objects. For instance, model 1(b), 2(b), 4(b), 8(b) all had some form of small gap between the corner cabinets and the objects on one side of the corner. When the grid is created, it uses points along the wall lines, however, the points from one wall may not align perfectly with the points from its connected walls, thus resulting in the space that is too small for the smallest width cabinet to fit, which in our case is 12 inches. Another undesirable, yet has no effect on the compliance of the design are the corner cabinet placements. There rules only state that corner cabinets must be placed in a corner, however, which corner is technically irrelevant from the rule perspective. This can be seen in the models 7(b), 9(b), and 10(b) where corner cabinets are placed in corners, but the corner that it has been placed in is not the corner the kitchen is built around. To account for this, a guideline would need to be added, for instance, enforcing two objects on either end of the corner cabinet or restricting the distance from the corner cabinet to the centroid of the existing kitchen object.

6.2.2 Living Room Layout

The second evaluation test was to compare our methodology against the works of [29]. We compare with their work, as opposed to more recent efforts, as they do not use a trained model for the relationships and rules of objects but rather explicitly evaluate the layout using quantifiable guidelines.

First, this required interpreting the rules they used in their language into our language. Table 6.2 shows the comparison for each of their rules with explanation on how the rule is formulated, bearing in mind their exact code is not available and thus was interpreted to the best of our ability.

One of the results of the generation using the translated rules, as we have implemented them, and a simplistic empty living room can be seen in Figure 6.1. Major differences in our work versus [29] is evident in the fact they the rules they proposed are tailored to the task of generative design with guidelines, rather than strict compliance rules. This is evident in the fact that a number of their guidelines conflict with one another and thus no fully compliant model is possible. For instance, two couches cannot be both facing each other and a focal point on a wall. Therefore, in their work they shift the designs towards "better" layouts, but there is no notion that a rule must pass. This is further evident in the weighting scheme for the final evaluation value. If all rules were to be intended to pass, then all rules should have equal weighting. In this research we use the model check as the evaluation, which is based on the notion that all rules must be met and thus there was no initial motivation to weigh each rule. To find the balance between the two would be to have rules

Table 6.2: Living Room Guidelines

Rule	Description [29]	Our Rule Language (Table A.2)
Clearance	Furniture items need open space around them to be accessible and functional	(1) All couches must have 30” clearance in front (2) All shelves must have 24” clearance in front (3) All cabinets must have 24” clearance in front (4) All dinning tables 36” clearance all around
Circulation	An effective furniture layout must support circulation through the room and access to all of the furniture	-
Pairwise(Distance)	Recommended distances between pairs of objects	(5) All coffee tables must be 16”-18” in front of all couches (6) All end tables must be 0-12” to the back or side of a couch
Pairwise(Angular)	Recommended angles between pairs of objects	(7) All coffee tables must be in front of couches
Conversation(Distance)	To support conversation at a normal tone of voice, the seats within a conversation area should be roughly four to eight feet apart	(8) Couch to couch distance must be 4-8ft
Conversation(Angular)	The seats should also be angled towards each other to encourage eye contact	(9) Couch to couch angled towards eachother
Balance	The principle is to place the mean of the distribution of visual weight at the center of the composition	(10) Center of mass of furnishing should be close to the room centroid
Alignment(Furniture)	In furniture arrangement, alignment primarily concerns the orientation of the furniture items relative to each other and to the walls of the room	(11) All furniture should be aligned with nearby furniture
Alignment(Wall)	Furniture items should also be aligned with nearby walls.	(12) All furniture should be aligned with nearby walls
Emphasis(Focal Point)	It is generally desirable to have a dominant focal point in the interior, so that the eye can rest without suffering competing demands for visual attention	(13) All furnishing should be angled towards focal point (This case center of room)
Emphasis(Symmetry)	The second emphasis term evaluates the symmetry of groups about their focal points	-

that are "must-pass" while others that are stylistic guidelines, that although important, are given lower priority to the mandatory regulations. This could use the Error/Recommendation/Warning tag associated with each rule but has been left for future work at this time.

The method used in this research for the generation was to place and check placements based on a grid. The major benefit over the method in [29] is our methods ability to vary the number of objects placed in the model. For instance, if you had an initial empty model that you wanted to populate with a kitchen, but did not know the number of objects required for the kitchen, this method could potentially be able to find the minimum number of objects that would need to be placed.

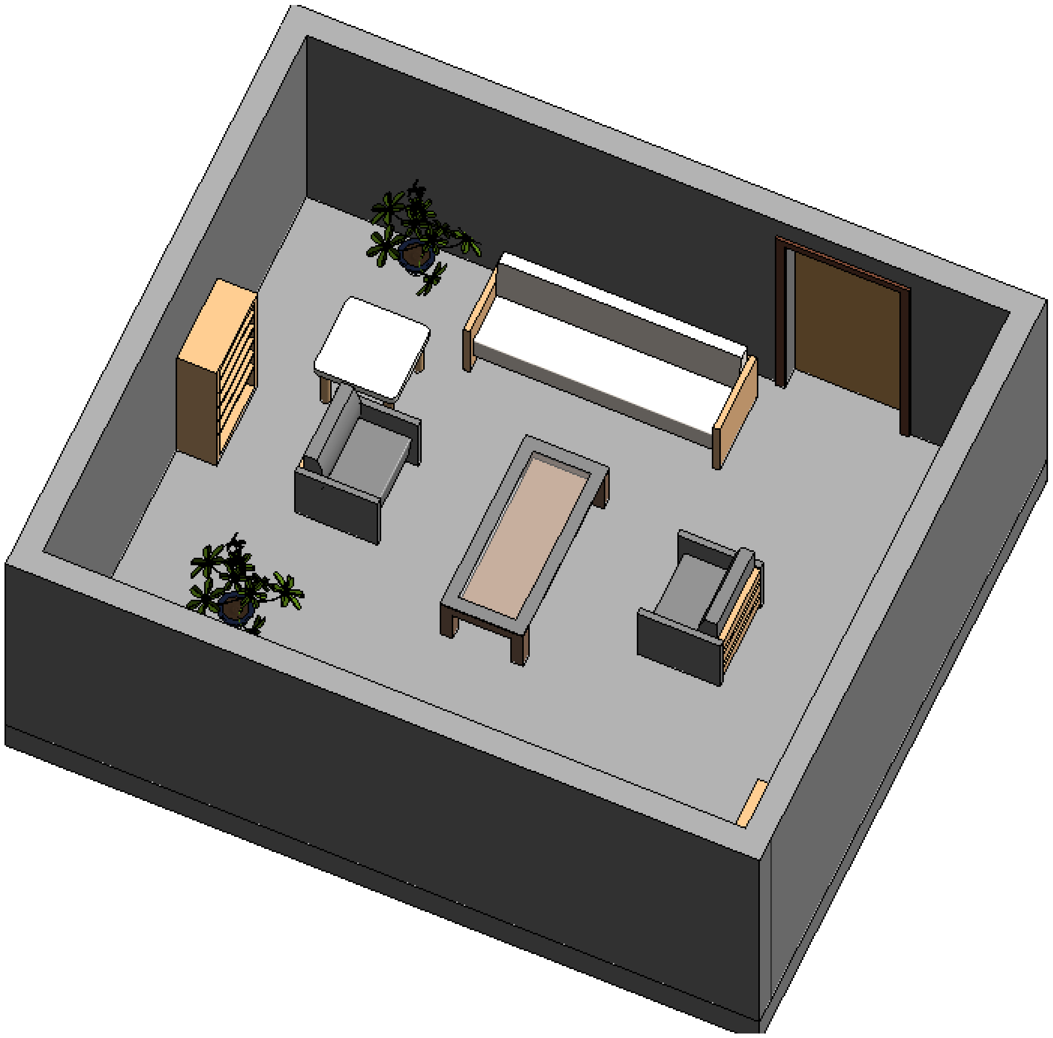


Figure 6.1: Generated living room.

6.3 Discussion

Assumptions A key assumption of the implementation is that the input to-be-placed objects are properly scaled and tagged with BIM metadata necessary for design-rule checking. This metadata includes the object type (*IfcWall*, *IfcFurnishing*, *IfcWindow*, etc.), orientation (front versus back of the object), the function property which can be determined based on the object name (such as 'chair', 'cabinet', 'sink'), and the surface boundary or mesh representation of the object. Therefore, there is an onus on the creator of the BIM objects to properly develop each of the objects.

Surface Geometry Simplification Rather than dealing with the potentially complex geometry of an object, we perform all geometric checks on the bounding-box triangulated meshes, which are created upon model import using the IFC Engine API. In the current implementation the objects are first placed as bounding boxes and then later replaced with the true object. A better approach would be to reduce the surface representation by a triangle reduction method rather than bounding boxes as objects liked curved walls can be an issue, however, for the general cases bounding boxes are sufficient. We will also explore in the future ways to link the virtual objects to the objects they depend on such that they can be updated rather than recreated.

Method Selection There are inherently a number of issues with this search strategy. First, it requires the grid and its spacing values to be set ahead of time including the object orientations. These values can play a key role in the ability to find fully compliant layouts as the granularity might not be sufficient for the rules check values. Second, as it is a one placement at a time approach, certain rules can be problematic for the generation. For instance, if there is a rule that states all objects must be placed surrounding the center of the room, then when the first object is placed it will be placed at the center of the room as no other objects have been placed yet, the next object will then attempt to be placed near the center as well and so on resulting in all objects iteratively being placed towards the center of the room. This was evident in the case of the living room furnishing center of mass being as close as possible to the center of the room.

The alternative would have been to use a Genetic Algorithm (GA) approach to the problem. This would have alleviated the issues with the placement. However, the general principle of this research was to use the model check based on the rule language for any generative design approach as they all require an evaluation stage. Future work will be to refine the project to use either GA or a combination of both for the generation and improve upon the results with a more intelligent generation and human interaction to the system.

Usability Feedback Finally, we did not receive feedback on the generated models from experts in the domain to determine the visual appeal of the generated layouts. Ideally, we would conduct surveys to determine the effectiveness and usability of the current system. This would first require extensive modification to the interface and importantly the 3D BIM model rendering and interaction. As we did not build off an existing software base we essentially build a BIM editor with minor functionality. The key challenge was in the rendering and editing of IFC models, which is an area that is severely limited in the current BIM industry.

6.4 Summary

In this chapter, we describe our method for generative design of interior layouts, based on model checking and design rules from the previous chapters. Our generative design method starts with an empty space to be filled with the desired objects, as opposed to previous methods that start with an initial placement and perform adjustments. The proposed method generates solutions using a heuristic search on a placement grid, and evaluates each object placement configuration, using a rule-based model compliance check.

We demonstrated the promise of our method by comparing the designs it produces for six real-world kitchen spaces, a challenging case study, due to the high number and complexity of relevant design rules. We also performed a comparative analysis against prior works on automated living room layouts, translating their rules into our language. Future work will include the use of Genetic Algorithms to improve search convergence on locally optimal solutions and gain insight and feedback into how a user might interact with such a system.

Chapter 7

Conclusion

The research presented in this thesis defines a rule language for building design compliance checking. The rule language and its subsequent model check methodology combine concepts put forth in prior research and have been implemented to provide validation for these concepts. What many previous languages lack is simplicity to create new rules, while maintaining expressiveness to cover a wide range of possible rules. By allowing for the reuse of implicit added objects (VOs) and geometrical properties and relations, and the ability to create complex rules using simple logical operations, the language proposed in this research can cover a broader range of possible rules.

Furthermore, previous rule languages have been focused on the evaluation of a model, returning a single pass or fail for a rule. We use the evaluation method from the rules-based model check to automatically place interior furnishing in a building layout, resulting in a rule compliant model. Typically in automated design, design guidelines are expressed in complex hard-coded formulas. Therefore, we have expanded the use of our rule language for generative design, scaling the results of rules numerically to mimic the automated design guideline formulas. Additionally, unlike previous automated design methods, we propose our solution in the context of BIM, specifically around the open BIM standard IFC, as the building model information representation. We tested our system using kitchen and living room layouts and populating them with interior furnishing objects. The kitchen results were compared against real-world designs to determine the similarities in the models, while the living rooms used translated rules from previous literature in order to exemplify the ability of their rules to be expressed in our language while maintaining similar results.

Future work will be to improve upon the generative-design algorithm that has more flexibility in the design configurations, as opposed to the grid placement. We also will perform a comprehensive study on the use of rule languages for model-checking and generative design from a user friendliness perspective as well as user interactions with the generative design system. Thus, we can find a way to add user preference and visual aesthetics into the evaluation without sacrificing the model code compliance.

To reiterate, the contributions of this thesis are:

An expressive **Rule Language** for representing design rules for the layout of elements in interior spaces (Chapter 4), outlined by the language syntax (Section 4.1), and the evaluation procedure (Section 4.2).

A **Model-Checking Algorithm** for evaluating the compliance of a BIM model against a set of design rules (Chapter 5). The methodology is outlined by the four stages of model-checking, namely rule interpretation (Section 5.1.1), model preparation (Section 5.1.2), execution (Section 5.1.3), and finally reporting (Section 5.1.4).

A **Generative Design Algorithm** for automating the placement of elements in interior spaces, relying on design rules (Chapter 6), which follows the three generative-design process requirements: A performance metric (Section 6.1.1), the configuration variation (Section 6.1.2), and a decision-making response (Section 6.1.3).

A service-oriented implementation of the above in a **Software Toolkit** that can be invoked by external client applications. This includes the IFC Engine (Chapter 3), the Rule Editor (Section 4.3), the Model-Checking .NET library (Section 5.2), and the Generative-Design Application (Section 6.2).

In summary, the contributions of this thesis are a simple yet expressive rule language for building design compliance checking and the methodology for model-checking using those rules. We demonstrated the ability to further utilize the rule-based model-checking as the evaluation stage in automated design.

References

[1] R. Akase and Y. Okada, "Automatic 3d furniture layout based on interactive evolutionary computation," in *2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*, IEEE, 2013, pp. 726–731. 12, 14, 15

[2] Autodesk. (2019). Dynamo, [Online]. Available: `https://dynamobim.org/`. 6, 15

[3] Autodesk. (2019). Revit, [Online]. Available: `https://www.autodesk.com/products/revit/overview`. 2, 6

[4] BIMServer. (2019). Bimserver, [Online]. Available: `http://bimserver.org/`. 6

[5] BuildingSMART. (2019). Buildingsmart, [Online]. Available: `https://www.buildingsmart.org/`. 17

[6] J. Choi and I. Kim, "A methodology of building code checking system for building permission based on openbim," in *ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction*, Vilnius Gediminas Technical University, Department of Construction Economics . . ., vol. 34, 2017. 9

[7] S. Daum and A. Borrmann, "Checking spatio-semantic consistency of building information models by means of a query language," in *Proc. of theIntl Conference on Construction Applications of Virtual Reality*, 2013. 39

[8] S. Davidson. (2019). Grasshopper, [Online]. Available: `https://www.grasshopper3d.com/`. 6

[9] J. Dimyadi and R. Amor, "Automated building code compliance checking–where is it at," *Proceedings of CIB WBC*, vol. 6, 2013. 1, 35

[10] L. Ding, R. Drogemuller, M. Rosenman, D. Marchant, and J. Gero, "Automating code checking for building designs-designcheck," 2006. 6

[11] C. Eastman, J.-m. Lee, Y.-s. Jeong, and J.-k. Lee, "Automatic rule-based checking of building designs," *Automation in construction*, vol. 18, no. 8, pp. 1011–1033, 2009. 1, 6, 34, 35

[12] T. Germer and M. Schwarz, "Procedural arrangement of furniture for real-time walkthroughs," in *Computer Graphics Forum*, Wiley Online Library, vol. 28, 2009, pp. 2068–2078. 11, 12

[13] V. Getuli, S. M. Ventura, P. Capone, and A. L. Ciribini, "Bim-based code checking for construction health and safety," *Procedia engineering*, vol. 196, pp. 454–461, 2017. 6

[14] Graphisoft. (2019). Archicad., [Online]. Available: `https://www.graphisoft.com/archicad/?`. 6

[15] N. Group. (2017). Corenet eplancheck, [Online]. Available: `https://www.nova-hub.com/e-government/`. 6

[16] E. Hjelseth, "Converting performance based regulations into computable rules in bim based model checking software," *eWork and eBusiness in Architecture, Engineering and Construction: ECPPM*, pp. 461–469, 2012. 10

[17] E. Hjelseth and N. Nisbet, "Capturing normative constraints by use of the semantic mark-up rase methodology," in *Proceedings of CIB W78-W102 Conference*, 2011, pp. 1–10. 10

[18] A. S. Ismail, K. N. Ali, and N. A. Iahad, "A review on bim-based automated code compliance checking system," in *2017 International Conference on Research and Innovation in Information Systems (ICRIIS)*, IEEE, 2017, pp. 1–6. 1, 6

[19] L. Jiang and R. M. Leicht, "Automated rule-based constructability checking: Case study of formwork," *Journal of Management in Engineering*, vol. 31, no. 1, A4014004, 2014. 6

[20] H. Kim, J.-K. Lee, J. Shin, and J. Choi, "Visual language approach to representing kbimcode-based korea building code sentences for automated rule checking," *Journal of Computational Design and Engineering*, vol. 6, no. 2, pp. 143–148, 2019. 10

[21] H. Lee, J.-K. Lee, S. Park, and I. Kim, "Translating building legislation into a computer-executable format for evaluating building permit requirements," *Automation in Construction*, vol. 71, pp. 49–61, 2016. 2, 9

[22] J. K. Lee, "Building environment rule and analysis (bera) language and its application for evaluating building circulation and spatial program," PhD thesis, Georgia Institute of Technology, 2011. 2, 7, 8

[23] J.-K. Lee, C. M. Eastman, and Y. C. Lee, "Implementation of a bim domain-specific language for the building environment rule and analysis," *Journal of Intelligent & Robotic Systems*, vol. 79, no. 3-4, pp. 507–522, 2015. 2, 8

[24] Y. Lee, C. Eastman, and J. Lee, "Automated rule-based checking for the validation of accessibility and visibility of a building information model," in *Computing in Civil Engineering 2015*, 2015, pp. 572–579. 6

[25] G. G. P. Ltd. (2019). Geometrygymifc, [Online]. Available: `https://github.com/GeometryGym/GeometryGymIFC`. 17

[26] A. Marsh, "Generative and performative design: A challenging new role for modern architects," in *The Oxford conference 2008*, WIT Press Oxford, 2008. 49

[27] W. Mazairac and J. Beetz, "Bimql–an open query language for building information models," *Advanced Engineering Informatics*, vol. 27, no. 4, pp. 444–456, 2013. 7, 8

[28] P. Merrell, E. Schkufza, and V. Koltun, "Computer-generated residential building layouts," in *ACM Transactions on Graphics (TOG)*, ACM, vol. 29, 2010, p. 181. 11, 12

[29] P. Merrell, E. Schkufza, Z. Li, M. Agrawala, and V. Koltun, "Interactive furniture layout using interior design guidelines," in *ACM transactions on graphics (TOG)*, ACM, vol. 30, 2011, p. 87. 2, 12, 13, 51, 62–64

[30] R. Niemeijer, B. De Vriès, and J. Beetz, "Check-mate: Automatic constraint checking of ifc models," *Managing IT in construction/managing construction for tomorrow, CRC Press, London*, pp. 479–486, 2009. 7, 10, 18

[31] S. Park, Y.-C. Lee, and J.-K. Lee, "Definition of a domain-specific language for korean building act sentences as an explicit computable form," *Electronic Journal of Information Technology in Construction*, vol. 21, no. 2016, 2016. 2, 7, 9

[32] P. Pauwels, D. Van Deursen, R. Verstraeten, J. De Roo, R. De Meyer, R. Van de Walle, and J. Van Campenhout, "A semantic rule checking environment for building performance checking," *Automation in Construction*, vol. 20, no. 5, pp. 506–518, 2011. 7, 8

[33] P. Pauwels and S. Zhang, "Semantic rule-checking for regulation compliance checking: An overview of strategies and approaches," in *32rd international CIB W78 conference*, 2015. 8

[34] C. Preidel and A. Borrmann, "Automated code compliance checking based on a visual language and building information modeling," in *ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction*, IAARC Publications, vol. 32, 2015, p. 1. 2, 10

[35] C. Preidel and A. Borrmann, "Towards code compliance checking on the basis of a visual programming language," *Journal of Information Technology in Construction (ITcon)*, vol. 21, no. 25, pp. 402–421, 2016. 2, 10

[36] C. Preidel and A. Borrmann, "Refinement of the visual code checking language for an automated checking of building information models regarding applicable regulations," in *Computing in Civil Engineering 2017*, 2017, pp. 157–165. 2, 7, 10

[37] M. Rahmani Asl, M. Bergin, A. Menter, and W. Yan, "Bim-based parametric building energy performance multi-objective optimization," 2014. 15

[38] M. Rahmani Asl, S. Zarrinmehr, M. Bergin, and W. Yan, "Bpopt: A framework for bim-based performance optimization," *Energy and Buildings*, vol. 108, pp. 401–412, 2015. 15

[39] M. Rahmani Asl, S. Zarrinmehr, and W. Yan, "Towards bim-based parametric building energy performance optimization," 2013. 15

[40] Solibri. (2019). Solibri, [Online]. Available: `https://www.solibri.com/`. 2, 5

[41] W. Solihin, J. Dimyadi, Y.-C. Lee, C. Eastman, and R. Amor, "The critical role of the accessible data for bim based automated rule checking system," in *Proceedings of the Joint Conference on Computing in Construction (JC3)*, vol. 1, 2017, pp. 53–60. 9, 36

[42] V. 1. Statsbygg BIM Manual. (2019). Statsbygg bim manual, version 1.2.1(sbm1.2.1), [Online]. Available: `https://www.statsbygg.no/files/publikasjoner/manualer/StatsbyggBIM-manual-ver1-2-1eng-2013-12-17.pdf`. 5

[43] E. Touloupaki and T. Theodosiou, "Performance simulation integrated in parametric 3d modeling as a method for early stage design optimization—a review," *Energies*, vol. 10, no. 5, p. 637, 2017. 5

[44] S. Wawan, "A simplified bim data representation using a relational database schema for an efficient rule checking system and its associated rule checking language," *School of Architecture, Georgia Institute of Technology*, 2016. 2, 7, 9

[45] T. Wortmann and G. Nannicini, "Introduction to architectural design optimization," in *City Networks*, Springer, 2017, pp. 259–278. 5

[46] L.-F. Yu, S. K. Yeung, C.-K. Tang, D. Terzopoulos, T. F. Chan, and S. Osher, "Make it home: Automatic optimization of furniture arrangement.," *ACM Trans. Graph.*, vol. 30, no. 4, pp. 86–1, 2011. 3, 12–14

[47] J. Zhang and N. M. El-Gohary, "Extending building information models semiautomatically using semantic natural language processing techniques," *Journal of Computing in Civil Engineering*, vol. 30, no. 5, p. C4016004, 2016. 10

Appendix A

Background Material

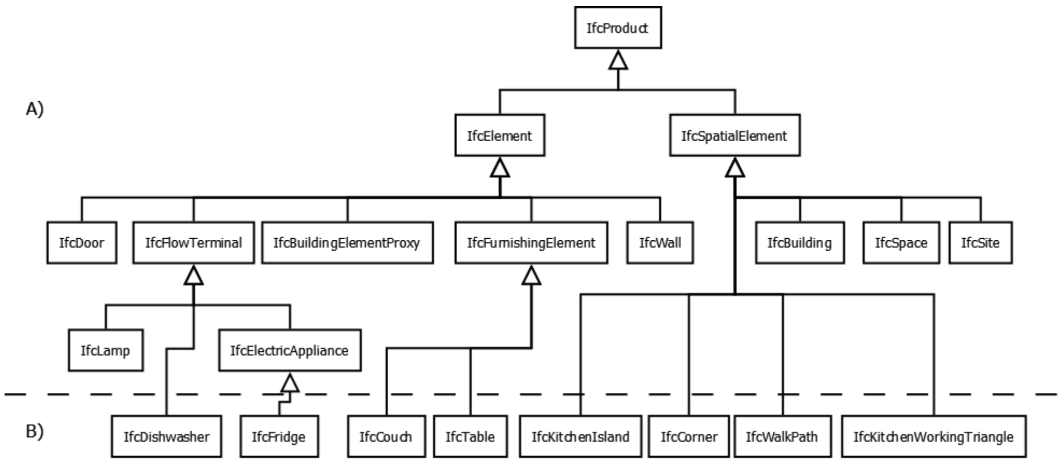


Figure A.1: Subset of the IFC class type hierarchy. A) IFC object types above the dashed line represent a subset of the current IFC4 schema. B) IFC object types under the dashed line indicate sample extensions to the IFC hierarchy; those under *IfcSpatialElement* representing VOs that would be created implicitly from the existing model *IfcElements*

Table A.1: Kitchen design rules in Natural Language and the proposed Rule Language.

Natural Language Rule	Rule Language
All furnishing objects must be placed with their back against a wall	ALL Object0 = IfcFurnishing ANY Object1 = IfcWall (Object1 and Object0 MUST-HAVE Is-Behind EQUAL True AND Object0 and Object1 MUST-HAVE BoundingBoxDistance LESS-THAN-OR-EQUAL 2INCH)

All corner cabinets must be placed in a corner	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Cornercabinet} ANY Object1 = IfcWall ANY Object2 = IfcWall (Object1 and Object0 MUST-HAVE IsBehind EQUAL True AND Object2 and Object0 MUST-HAVE IsNextTo EQUAL True AND Object1 and Object0 MUST-HAVE BoundingBoxDistance LESS-THAN-OR-EQUAL 2INCH AND Object2 and Object0 MUST-HAVE BoundingBoxDistance LESS-THAN-OR-EQUAL 12INCH)
Each edge of the kitchen working triangle, implied by the front center of the sink, range, and refrigerator, must be between 4ft and 9ft long	ANY Object0 = IfcTriangle (Object0 MUST-HAVE MinEdgeLength GREATER-THAN-OR-EQUAL 4FT AND Object0 MUST-HAVE MaxEdgeLength LESS-THAN-OR-EQUAL 9FT)
The sum of edges of the kitchen working triangle must be between 13ft and 26ft	ANY Object0 = IfcTriangle (Object0 MUST-HAVE TotalEdgeLength GREATER-THAN-OR-EQUAL 13FT AND Object0 MUST-HAVE TotalEdgeLength LESS-THAN-OR-EQUAL 26FT)
No furnishing items should block a window	ALL Object0 = IfcFurnishing {FunctionOfObj NOT-EQUAL Sink} ALL Object1 = IfcWindow (Object1 and Object0 MUST-HAVE IsBehind EQUAL False OR Object0 and Object1 MUST-HAVE BoundingBoxDistance GREATER-THAN-OR-EQUAL 4FT)

No furnishing should block a door	ALL Object0 = IfcDoor ALL Object1 = IfcFurnishing (Object0 and Object1 MUST-HAVE BoundingBoxDistance GREATER-THAN 60INCH OR (Object1 and Object0 MUST-HAVE InFrontOf EQUAL False AND Object1 and Object0 MUST-HAVE IsBehind EQUAL False))
A sink must be placed under a window	ANY Object0 = IfcFurnishing {FunctionOfObj EQUAL Sink} ANY Object1 = IfcWindow (Object0 and Object1 MUST-HAVE CenterDistanceXY LESS-THAN-OR-EQUAL 2FT)
The range cannot be placed under a window and should be 12in away from a window	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Range} ALL Object1 = IfcWindow (Object0 and Object1 MUST-HAVE CenterDistanceXY GREATER-THAN 6FT)
No object may be placed less than 48in in front of a furnishing object with a door	ALL Object0 = IfcFurnishing {FunctionOfObj NOT-EQUAL Cornercabinet} ALL Object1 = IfcAllNonVirtualTypes (Object1 and Object0 MUST-HAVE BoundingBoxDistance GREATER-THAN 48INCH OR Object1 and Object0 MUST-HAVE InFrontOf EQUAL False)

The sink must have a minimum of 3in of counter space on one side and 18in on the other	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Sink} ANY Object1 = IfcCounterTop ANY Object2 = IfcCounterTop (Object1 MUST-HAVE Width GREATER-THAN-OR-EQUAL 3INCH AND Object2 MUST-HAVE Width GREATER-THAN-OR-EQUAL 18INCH AND Object0 and Object1 MUST-HAVE IsNextTo EQUAL True AND Object0 and Object2 MUST-HAVE IsNextTo EQUAL True AND Object0 and Object1 MUST-HAVE BoundingBoxDistance LESS-THAN-OR-EQUAL 1FT AND Object0 and Object2 MUST-HAVE BoundingBoxDistance LESS-THAN-OR-EQUAL 1FT)
The range must have a minimum of 15in of counter space on one side	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Range} ANY Object1 = IfcCounterTop (Object1 MUST-HAVE Width GREATER-THAN-OR-EQUAL 15INCH AND Object1 and Object0 MUST-HAVE IsNextTo EQUAL True AND Object0 and Object1 MUST-HAVE BoundingBoxDistance LESS-THAN-OR-EQUAL 1FT)
The refrigerator must have a minimum of 16in of counter space on one side	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Refrigerator} ANY Object1 = IfcCounterTop (Object1 MUST-HAVE Width GREATER-THAN-OR-EQUAL 16INCH AND Object0 and Object1 MUST-HAVE BoundingBoxDistance LESS-THAN-OR-EQUAL 1FT AND Object1 and Object0 MUST-HAVE IsNextTo EQUAL True)

(Additional) Cabinets must be directly against another kitchen furnishing object	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Cabinet} ANY Object1 = IfcFurnishing (Object0 and Object1 MUST-HAVE BoundingBoxDistance LESS-THAN-OR-EQUAL 2INCH)

Table A.2: Living room design rules in Natural Language and the proposed Rule Language.

Natural Language Rule	Rule Language

(1) All couches must have 30” clearance in front	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Couch }
	ALL Object1 = IfcAllNonVirtualTypes {FunctionOfObj NOT-EQUAL CoffeeTable }
	
	(Object1 and Object0 MUST-HAVE InFrontOf EQUAL False OR
	Object0 and Object1 MUST-HAVE BoundingBoxDistance GREATER-THAN 30INCH)

(2) All shelves must have 24” clearance in front	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Shelf } ALL Object1 = IfcAllNonVirtualTypes (Object1 and Object0 MUST-HAVE InFrontOf EQUAL False OR Object0 and Object1 MUST-HAVE BoundingBoxDistance GREATER-THAN 24INCH)
(3) All cabinets must have 24” clearance in front	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Cabinet} ALL Object1 = IfcAllNonVirtualTypes (Object0 and Object1 MUST-HAVE BoundingBoxDistance GREATER-THAN 24INCH OR Object1 and Object0 MUST-HAVE InFrontOf EQUAL False)

(4) All dinning tables 36" clearance all around	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL DiningTable} ALL Object1 = IfcAllNonVirtualTypes (Object0 and Object1 MUST-HAVE BoundingBoxDistance GREATER-THAN 36INCH)
(5) All coffee tables must be 16"-18" in front of all couches	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL CoffeeTable } ALL Object1 = IfcFurnishing {FunctionOfObj EQUAL Couch } (Object0 and Object1 MUST-HAVE InFrontOf EQUAL True AND Object0 and Object1 MUST-HAVE BoundingBoxDistance LESS-THAN-OR-EQUAL 18INCH AND Object0 and Object1 MUST-HAVE BoundingBoxDistance GREATER-THAN-OR-EQUAL 16INCH)
(6) All end tables must be 0-12" to the back or side of a couch	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL EndTable } ANY Object1 = IfcFurnishing {FunctionOfObj EQUAL Couch } (Object0 and Object1 MUST-HAVE BoundingBoxDistance GREATER-THAN-OR-EQUAL 0INCH AND Object0 and Object1 MUST-HAVE BoundingBoxDistance LESS-THAN-OR-EQUAL 12INCH AND (Object0 and Object1 MUST-HAVE IsNextTo EQUAL True OR Object0 and Object1 MUST-HAVE IsBehind EQUAL True))

(7) All coffee tables must be in front of couches	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL CoffeeTable } ALL Object1 = IfcFurnishing {FunctionOfObj EQUAL Couch } (Object1 and Object0 MUST-HAVE AngleBetweenForwardAndAngleTo EQUAL 0DEG)
(8) Couch to couch distance must be 4-8ft	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Couch } ALL Object1 = IfcFurnishing {FunctionOfObj EQUAL Couch } (Object0 and Object1 MUST-HAVE CenterDistanceXY GREATER-THAN-OR-EQUAL 4FT AND Object0 and Object1 MUST-HAVE CenterDistanceXY LESS-THAN-OR-EQUAL 8FT)
(9) Couch to couch angled towards eachother	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Couch } ANY Object1 = IfcFurnishing {FunctionOfObj EQUAL Couch } (Object0 and Object1 MUST-HAVE AngleBetweenForwardAndAngleTo EQUAL 0DEG)
(10) Center of mass of furnishing should be close to the room centroid	ALL Object0 = IfcFurnishingCoM ALL Object1 = IfcRoomCentroid (Object0 and Object1 MUST-HAVE CenterDistanceXY EQUAL 0INCH)

(11) All furniture should be aligned with nearby furniture	ALL Object0 = IfcFurnishing ALL Object1 = IfcFurnishing (Object0 and Object1 MUST-HAVE AlignmentAngle EQUAL 0DEG OR Object0 and Object1 MUST-HAVE AlignmentAngle EQUAL 90DEG OR Object0 and Object1 MUST-HAVE AlignmentAngle EQUAL 180DEG OR Object0 and Object1 MUST-HAVE BoundingBoxDistance GREATER-THAN 2FT)
(12) All furniture should be aligned with nearby walls	ALL Object0 = IfcFurnishing ALL Object1 = IfcWall (Object0 and Object1 MUST-HAVE AlignmentAngle EQUAL 90DEG OR Object0 and Object1 MUST-HAVE BoundingBoxDistance GREATER-THAN 2FT)
(13) All furnishing should be angled towards focal point (This case center of room)	ALL Object0 = IfcFurnishing ANY Object1 = IfcRoomCentroid (Object0 and Object1 MUST-HAVE AngleBetweenForwardAndAngleTo EQUAL 0DEG)
(Extra) All shelves must be places against a wall	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Shelf } ANY Object1 = IfcWall (Object1 and Object0 MUST-HAVE IsBehind EQUAL True AND Object0 and Object1 MUST-HAVE BoundingBoxDistance LESS-THAN-OR-EQUAL 1INCH)
(Extra) All plants should be placed near a wall	ALL Object0 = IfcFurnishing {FunctionOfObj EQUAL Plant } ANY Object1 = IfcWall (Object0 and Object1 MUST-HAVE BoundingBoxDistance EQUAL 0INCH)