Ingenieurfakultät Bau Geo Umwelt
Lehrstuhl für Computergestützte Modellierung und Simulation
Prof. Dr.-Ing. André Borrmann

A Mechanism For Capturing Implicit Design Knowledge In Building Information Models Using Graph Transformations

Gunther Bidlingmaier

Interdisziplinäres Projekt
für den Master of Science Studiengang Informatik

Autor:	Gunther Bidlingmaier
Matrikelnummer:	
Betreuer:	Prof. Dr.-Ing. André Borrmann
	Jimmy Abualdenien, M.Sc.
Abgabedatum:	27. März 2020

Abstract

With the increasing adoption of Building Information Modeling (BIM), Computer Assisted Design has become a vital tool in every lifecycle phase of a building. Creating a Building Information Model however is also time consuming and costly.
Computational Design Synthesis (CDS) is a research field which tries to automate the Computer Aided Design process by formalizing implicit design knowledge, making it explicit. A possible barrier for its adoption however is the additional skill required for formalizing the design knowledge, which is not a typical part of e.g. architecture education.
This work is concerned with seamlessly integrating the formalization of implicit design knowledge using graph transformations into the design process. We first show how buildings can be represented as graphs and how the formal concept of graph transformations can be used define modifications on these building graphs. The software tool GrGen (Geiß *et al.*, 2006) is used to demonstrate how to implement and apply graph transformation rules.
We proceed to present a conceptual mechanism allowing the automatic generation of graph transformation rules from a user-selected part of an existing Building Information Model. These graph transformation rules capture the implicit design knowledge which went into the creation the of existing model. The rules can then be applied in a different context (in the same or a different building model) to automate the design process. The main advantage of our generation mechanism is that it allows the usage of the graph transformation formalism without requiring knowledge or skill in it.
Our presentation of the generation mechanism has two parts:

1. A formal definition of the generation mechanism based on graph theory.

2. A discussion on how this generation mechanism can be integrated into a BIM design workflow.

We complement our conceptual framework with a software prototype implementing the rule generation from a given building graph using the GrGen API.

Contents

Chapter 1

Introduction

With the increasing adoption of Building Information Modeling (BIM), Computer Assisted Design has become a vital tool in every lifecycle phase of a building. From the initial design, the actual construction to the maintenance and renovation of buildings, a shared Building Information Model enables better coordination and communication between stakeholders (Abualdenien & Borrmann, 2019; Borrmann *et al.*, 2018).
Besides the obvious benefit of having a shared 3D geometric model of the building, BIM has a second, more intricate benefit over 2D drawings: It can not only model the geometry of building elements but also their semantics. This is what has enabled a whole class of new downstream applications, aiding in the design and construction phase. These range from planning tools for construction scheduling and quantity take-off to domain specific applications for the structural analysis of models, energy simulations and daylighting analysis. Creating a Building Information Model however also needs a lot of work of valuable architects and engineers (Borrmann *et al.*, 2018).
Computational Design Synthesis (CDS) is a research field which tries to automate the computer aided design process by formalizing implicit design knowledge, making it explicit. Graph grammars or graph rewriting systems are one such formalization method which has been maturing over the last decades to the point where it can capture real and complex engineering design rules (Chakrabarti *et al.*, 2011). A possible barrier for its adoption however might be the additional skill and knowledge required for using graph rewriting systems.
This work is concerned with seamlessly integrating the formalization of implicit design knowledge into the design process using graph transformations. We first give a rough intuition about how buildings can be represented as graphs and show how the concept of graph transformations can be used define modifications on these building graphs. The software tool GrGen (Geiß *et al.*, 2006) is used to demonstrate how to create and apply graph transformation rules. We proceed to present a concept where elements of a building graph are selected as a pattern by the designer and then automatically extracted into a graph transformation rule. This graph transformation rule captures implicit design knowledge and can be created without

knowledge about the actual formalization language. The rule can be applied to the same or different building graphs in order to automatically apply the now formalized knowledge in a different context. We also discuss how this formal mechanism could be integrated into a BIM design workflow. The conceptual presentation of the mechanism is complemented with a software prototype implementing the rule generation from a given building graph using the GrGen API.

Chapter 2

Graph Transformations

This chapter explains all concepts needed for understanding the generation mechanism described in chapter 3. We start with the basic notion of a graph, show how a graph can model a building and how a graph transformation can be formalized and then implemented with the GrGen tool.

2.1 Basic Graph Theory

A graph is a discrete mathematical structure used to model relationships (edges) between objects (vertices). Formally, a graph is an ordered pair $G = (V, E)$ where V describes the set of vertices and E the set of edges. An edge $e \in E$ consists of two vertices $u, v \in V$. In mathematics one usually discerns *directed* and *undirected* graphs. For the former, edges have a direction and are hence defined as ordered tuples $(u, v) \in E$. For the latter the direction of edges is unspecified and edges are hence defined as unordered sets $\{u, v\} \in E$. Note that directed graphs are more general, since an undirected graph can be represented by replacing every edge $\{u, v\}$ with two directed edges (u, v) and (v, e).
Graphs have an intuitive visual representation. For every vertex $v \in V$ we draw a circle and label it with the name of v. For every $(u, v) \in E$ we draw an arrow from u to v. In the following we will assume all graphs to be directed with undirected edges between vertices u and v denoting the presence of both a directed edge from u to v and from v to u.
The top of figure 2.1 shows how a graph can model relationships in buildings. Our vertices V represent rooms of the building floor plan shown at the top left of the figure. [1] An edge (u, v) between vertices u and v denotes that room u is next to room v. This example illustrates how a graph can be used to model relationships in a domain, but also shows the need for a more nuanced graph model, if we want to model all information present in the floor plan.

[1] Note that for the sake of simplicity we will restrict ourselves to individual floor plans of buildings in this paper. The concepts however seamlessly carry over to the examination of multiple levels at the same time.

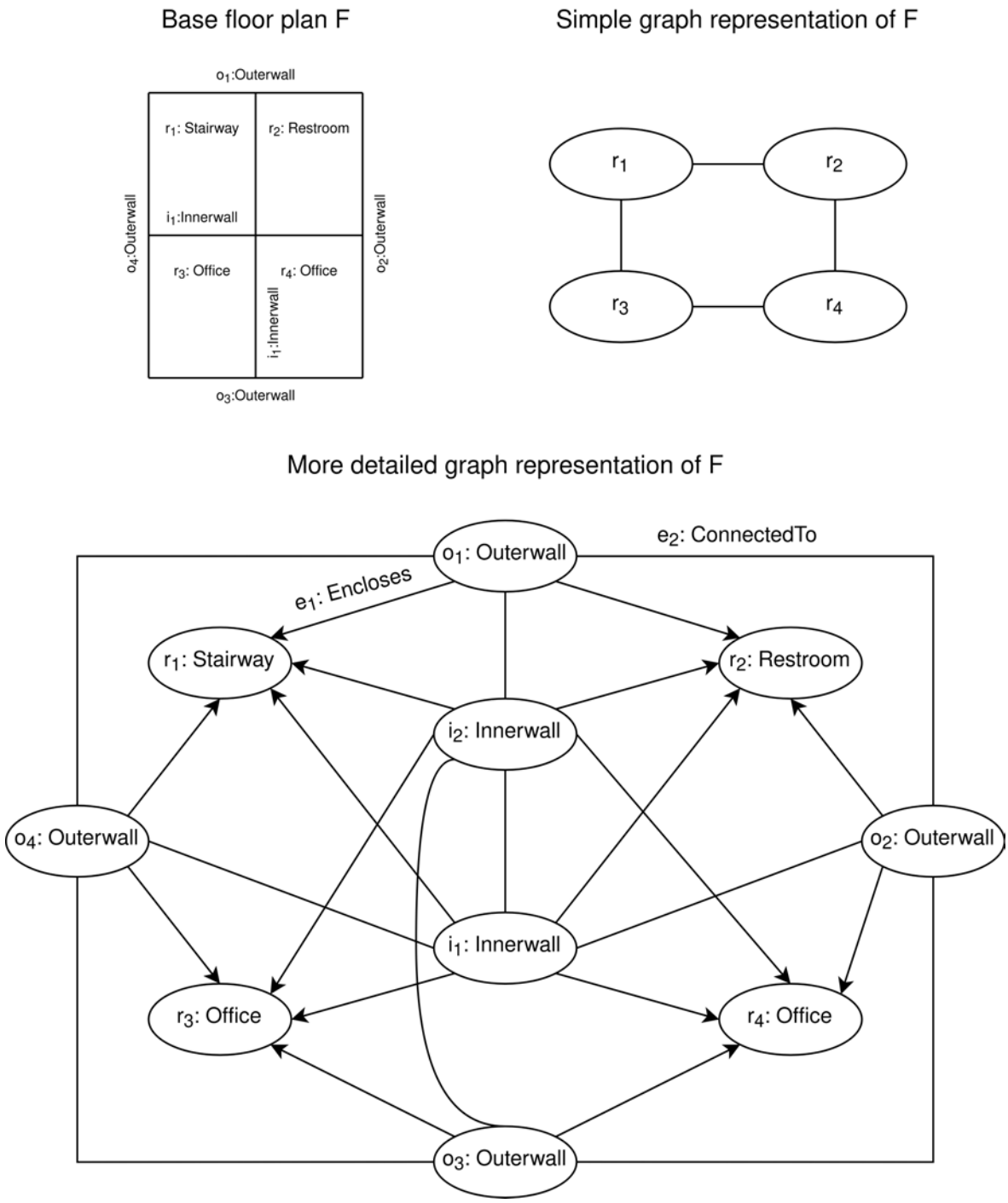

Figure 2.1: Two graph representations of the floor plan F with different levels of detail. Note that we have left out most of the edge annotations for the sake of simplicity.

2.2 Representing Buildings As Graphs

This section introduces the intuitive notion of a typed attributed graph. We restrict ourselves to the intuition motivated by the previous example in figure 2.1, since this is sufficient for the purpose of this paper. For a more formal definition of typed attributed graphs in the framework of category theory, refer to (Ehrig *et al.*, 2004).
If we really want to capture a whole building and its semantic we need to add two additional kinds of information:

1. We need to capture all elements of the building and their relationships. This means we need different types of nodes for representing not only rooms, but also walls, doors, windows and other building elements. Since these elements have different kinds of relationships with each other (e.g. spatial containment of a door in a wall), we also need different types of edges to represent this.

2. Objects in buildings posses important semantic properties such as their material. To model these, we have to incorporate attributes into our definition of nodes.

If we incorporate this information for our example building, we get the much more sophisticated graph at the bottom of figure 2.1. While we have chosen this particular graph representation, there are many possible ways to encode the geometric and semantic relationships between building elements into edges. One could e.g. add edges to connect windows and doors to the rooms they belong to, and not only the walls they are contained in.
Note that which particular graph representation models a building correctly may depend on the particular use case. The reason for this is that by representing the building as a graph, we have abstracted the precise geometry from the model. This abstraction is useful since it enables us to formulate abstract graph transformation rules on buildings, but it's also a loss of information. We will later see that this abstraction in some cases makes it hard to specify the exact geometry of building elements after their creation or transformation in the building graph.

2.3 Graph Transformation Formalism

While there are multiple algebraic methods of formalising graph transformations (also called graph rewrites), for the purpose of this paper we restrict ourselves to the *Single Pushout* approach. For a more thorough examination, refer to (Ehrig *et al.*, 1997), (Rozenberg, 1997).
Intuitively, the *Single Pushout* method defines a generic pattern which is searched for in the application graph and replaced as specified.
More formally we define: A *graph transformation rule* t is a triple (L, R, r), where L is a graph

called the *Pattern Graph* and R is called the *Rewrite Graph*. $r : L \to R$ is a (possibly) partial injective morphism called the *preservation morphism*. In an application of t to an *Application Graph* A, the *Pattern Graph* L is searched for as subgraph of A. Formally, an isomorphism m from L to a subgraph of A is searched. $m(L)$ is then the matched subgraph of A to be replaced by R. This is where r is needed. Since m is an isomorphism it has a unique inverse m^{-1}.
We can now replace every node or edge o in $m(L)$:

1. If $r(m^{-1}(o))$ is defined, we replace o in A with $r(m^{-1}(o))$.
2. If $r(m^{-1}(o))$ is not defined, we discard o from A.
3. We add to A every node and edge of R not mapped to by r.

The result is the modified Application Graph A'.

2.3.1 An Example Graph Transformation

In the following we explain this formalism exemplary for the building graph at the bottom of figure 2.1.
Suppose we want to add automatic doors between office spaces and stairways, in our case between r_1 and r_3. More specifically, we want to insert an automatic door into every inner wall which encloses both an office and a stairway. This could be sensible to make sure doors are not open for long, ensuring the HVAC system in the office spaces can work properly.
The *Pattern Graph* L has to consist of three nodes. A central node of type *Innerwall* and two room nodes connected to the wall via *Enclose* edges going from the wall node to the respective room. One of the rooms has to be of type *Office* the other of type *Stairway*.
The *Rewrite Graph* R should also have these nodes and the connecting edges, since we only want to add to the *Application Graph*. For the addition of the door, we need to add a node of type *AutomaticDoor* and a connecting edge of type *Contains* to the wall node in R.
Figure 2.2 shows L and R. Additionally, the preservation morphism r is shown with the dashed lines. In the bottom of the figure, the only possible match of L in our building graph is shown in blue. The resulting addition of the door node is marked in green. The isomorphism m corresponding to this match is the following: $m(i_L) = i_1, m(sw_L) = r_1, m(o_L) = r_3, m(e_L) = e_1, m(f_L) = e_2$. Note that there could be multiple possible matches of L in a different building graph, e.g. if r_2 also was an office room.

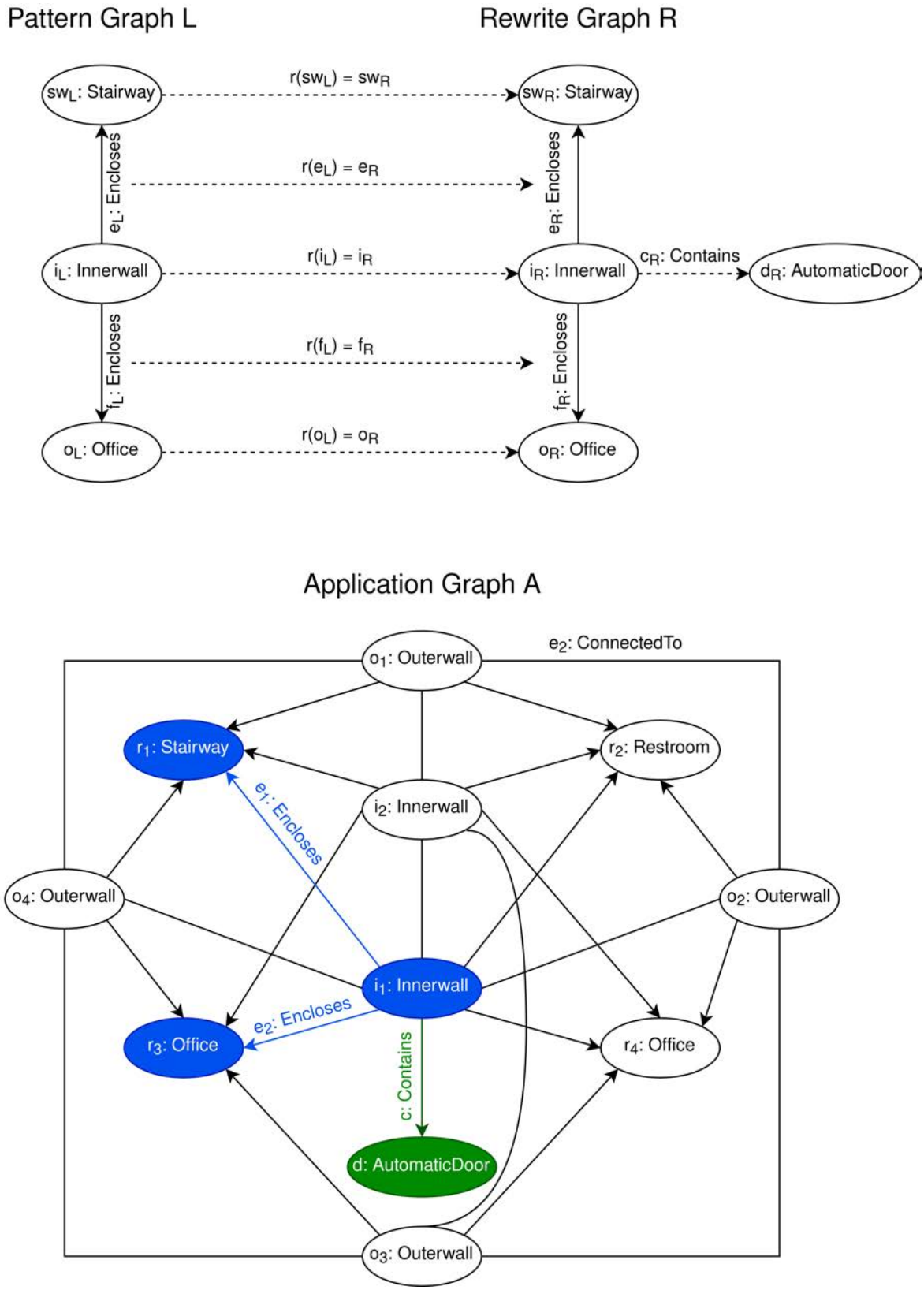

Figure 2.2: The graph transformation rule described in section 2.3.1 is shown. The match of L in A is marked in blue. The addition to A by the rule application is marked in green.

2.4 GrGen

GrGen (Graph Rewrite Generator) is a software development tool that offers programming languages optimised for graph structured data, with declarative pattern matching and rewriting at their core, with support for imperative and object-oriented programming (Jakumeit *et al.*, 2010). In other words, it offers languages specific for implementing graph transformation rules. At the core of GrGen are two languages:

1. The *Graph Model Language*, which is used to define the elements of a valid typed and attributed graph. It defines all node and edge types and their respective attributes.
2. The *Rule Language*, which is used to define graph transformation rules on a given graph model. These transformation rules can then be applied to a graph using the GrGen API.

While GrGen has a lot of advanced features, we restrict ourselves to the functionality needed for our prototype. A complete reference can be found in the GrGen manual (Jakumeit *et al.*, 2010).

2.4.1 The GrGen Graph Model Language

The *Graph Model Language* is used to define the elements of a valid graph. In the following `[name]` marks that `[name]` can be replaced with a chosen name for the rule, attribute or class. The brackets are therefore not part of the GrGen syntax. New node types and their attributes can be defined with the following syntax.

```
1 node class [class name] {
2 [attribute1 name]:[attribute1 type];
3 ...
4 }
```

Note that attributes are also typed. They are however subject to a type system separate from the one for nodes and edges. Edge types can be defined similarly:

```
1 edge class [class name] {
2 [attribute1 name]:[attribute1 type];
3 ...
4 }
```

The *Graph Model Language* implements an object oriented type/class system in essence similar to the ones used by many contemporary general purpose programming languages. This means one can model class hierarchies often found in the real world. A class inheritance is indicated after the class name of a node or edge class using the *extends* keyword.

```
1  node class Wall;
2  node class Outerwall extends Wall;
3  node class Innerwall extends Wall;
4  node class Room;
5  node class Stairway extends Room;
6  node class Restroom extends Room;
7  node class Office extends Room;
8  node class Door;
9  node class Window;
10 edge class Contains extends Edge;
11 edge class Encloses extends Edge;
12 edge class ConnectedTo extends UEdge;
```

Figure 2.3: A sample GrGen Graph Model. All node and edge types of the building graph in 2.1 are defined.

```
1  node class [class name 1] extends [class name 2] { ... }
```

The typical polymorphic behaviour of object oriented hierarchies applies: An object of class A which extends class B can be used in a context requiring an object of class B but not the other way round.

The *Graph Model Language* is best understood when applying it to an actual example. For our previous building graph in figure 2.1, the corresponding *Graph Model* definitions are as shown in figure 2.3. We left out all attributes to keep the example clear. The model definition is nonetheless sufficient for our example building graph in figure 2.1. Note the predefined edge classes *Edge* and *UEdge*, provided by GrGen to distinguish between directed and undirected edges.

2.4.2 The GrGen Rule Language

The *Rule Language* is used to define transformation rules on graphs. Rules can be defined with the syntax:

```
1  rule [rule name] {
2    [pattern1];
3    ...
4    replace {
5      [graphlet1];
6      ...
7    }
8  }
```

The patterns define how the *Application Graph* has to look like for the rule to be applicable. The *replace* part defines how the *Application Graph* is modified upon application of the rule. Rewrite rules are best understood by example, which is why we will explain their parts while

```
1  rule insertAutomaticDoor {
2    i:Innerwall;
3    i -e1:Encloses-> office:Office;
4    i -e2:Encloses-> sw:Stairway;
5    negative {
6      i -e4:Contains-> ac1:AutomaticDoor;
7    }
8    if {
9      office.ventilated;
10   }
11   replace {
12     office <-e2- i -e1-> sw;
13     i -e4:Contains-> ac2:AutomaticDoor;
14   }
15 }
```

Figure 2.4: A GrGen rule implementing the graph transformation described in section 2.3.1

implementing the exemplary graph transformation described in 2.3.1. The complete GrGen rule implementing this transformation is shown in figure 2.4.

Rule Patterns

The patterns describe what we defined in section 2.3 as the *Pattern Graph*. The simplest form of a pattern is a *graphlet*, which consists of typed nodes and typed edges between those nodes.

Lines 2-4 in our rule are graphlet patterns. Line 2 describes a single node. This has two effects: First, it requires an *Application Graph* to have a node of type *Innerwall* for the rule to be applicable. Second, it binds this *Innerwall* node to the name *i*. This makes it possible to refer to the specific *Innerwall* node *i* and not some other *Innerwall* in the following lines. Line 3 and 4 further define connections which are required of *i*. Line 3 requires the *Application Graph* to have an *Office* node which is connected to *i* with an *Enclose* edge going from *i* to the *Office* node. Line 4 requires the *Application Graph* to have a *Stairway* node which is connected to *i* with an *Enclose* edge going from *i* to the *Stairway* node. Taking lines 2-4 together results exactly in our example *Pattern Graph* of figure 2.2.

GrGen allows slightly more than our previous graph formalism. *negative* patterns specify graphlets which are not allowed be present in the *Application Graph*. Lines 5-7 specify a *negative* pattern, not allowing node i to already contain an *AutomaticDoor* node.

if patterns specify boolean conditions on the properties of nodes and edges. These conditions have to hold for the rule to be applicable. Lines 8-10 specify an *if* pattern, which requires the office node to have the property *ventilated* set to *True*.

As mentioned, lines 5-10 go beyond the graph rewrite specification of our previous example. Since they are additional requirements for the rule to match, they make the rule more specific.

```
1  rule insertAutomaticDoor {
2    i:Innerwall;
3    i -e1:Encloses-> office:Office;
4    i -e2:Encloses-> sw:Stairway;
5    negative {
6      i -e4:Contains-> ac1:AutomaticDoor
7    }
8    if {
9      office.ventilated
10   }
11   modify {
12     i -e4:Contains-> ac2:AutomaticDoor
13   }
14 }
```

Figure 2.5: A GrGen rule equivalent to the one shown in figure 2.4 using the *modify* keyword

Replace Part

The replacement part of a GrGen rule describes what we defined in section 2.3 as the *Rewrite Graph*.
Hence it is only natural that the *replace* part starting in line 11 of figure 2.4 also uses graphlets. Line 12 specifies the part which we want to keep: The inner wall i and its edges to the office room and the stairway. This also shows that multiple connections can be specified in one line or graphlet. Line 13 specifies the part which we want to add: A node ac_2 of type *AutomaticDoor* which is connected with node i via a directed edge of type *Contains* going from i to ac_2. Again, this describes the same graph as shown in figure 2.2, but this time the *Rewrite Graph*.

There is an alternative in GrGen for the *replace* part, using the *modify* keyword. An equivalent definition of the `insertAutomaticDoor` rule using the *modify* keyword is shown in figure 2.5. The difference between the two keywords is that *modify* keeps all graph elements of the *Pattern Graph* by default. Hence we do not have to explicitly specify that we want to keep the office and the stairway. Going forward we will only use the *modify* keyword.

A Note On The Preservation Morphism

One might wonder whether the definition of the preservation morphism r in section 2.3 was unnecessary since we didn't mention it in our rule. But in fact, we specified this morphism implicitly through the name bindings. For the rule 2.4 using the *rewrite* keyword, we reused the identifiers $office, e_2, i, e_1, sw$ in the *rewrite* part. This corresponds to the mapping $r(office) = office$ and $r(e_2) = e_2$, and so on. Since we mentioned all identifiers in the *replace* part, r is defined for all elements in the *Pattern Graph*. Therefore no nodes are removed upon application of the rule. The edge e_4 and the node ac_2 are not mapped to by r

(formally there does not exist an object o in the *Pattern Graph*, s.t. $r(o) = e_4$ or $r(o) = ac_2$). Therefore e_4 and ac_2 are added to the *Application Graph* upon application of the rule.

Chapter 3

Generation Of Graph Transformation Rules

In chapter 2 we have presented a powerful formalism and its implementation which can be used to transform building models represented as graphs. The formalism is however hardly intuitive and its implementation with GrGen as presented above has little chance of being useful to an actual user of BIM software.
We therefore present a conceptual mechanism for generating graph transformation rules from patterns in existing building graphs. Integrating this mechanism within BIM software could allow the usage of graph transformations for buildings without knowledge in the graph transformation formalism or the GrGen language.

In section 3.1 we present the conceptual mechanism and possible use cases. We also sketch how this mechanism can be integrated into a BIM design workflow, regarding both usability and software integration. In section 3.2 we furthermore explain a prototypical implementation of the core mechanism using the GrGen API.

3.1 Generation Mechanism

3.1.1 Formal Definition

The core mechanism for the generation of graph transformation rules works as follows: Given a building graph E, we select a subgraph $E_{pattern}$ of E for extraction. We call E the *Extraction Graph*. The subgraph $E_{pattern}$ should be an instance of a design pattern in the domain. [1]

[1] As an example refer to the graph transformation described in section 2.3.1: Doors between office spaces and stairways should be automatically opening and closing. This can be sensible to ensure that the doors to the office spaces are not open for long, enabling the HVAC system to properly regulate air flow and temperature in the offices.

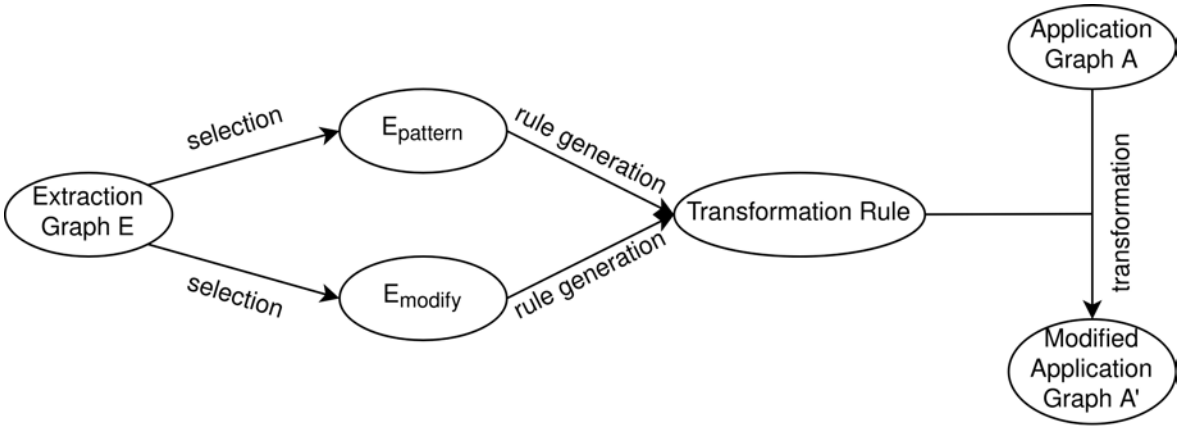

Figure 3.1: A high level view of the core generation mechanism.

$E_{pattern}$ is the basis for the generation of our graph transformation rule. Formally, it functions as the *Pattern Graph* of the transformation rule. We also select a subgraph from E - E_{modify} - specifying which elements should be added to a building graph upon application of the transformation rule. $E_{pattern}$ and E_{modify} should not overlap, formally $E_{pattern} \cap E_{modify} = \emptyset$. Taken together - formally $E_{pattern} \cup E_{modify}$ - the two subgraphs specify the *Rewrite Graph* of our transformation rule. The preservation morphism r therefore preserves all elements of $E_{pattern}$ for the *Rewrite Graph*. This graph transformation rule now captures design knowledge which went into the creation of the building represented by E.
The rule can be applied to a different building graph, which we will call the *Application Graph* A. This automates the design steps necessary to implement the design knowledge captured with $E_{pattern}$ and E_{modify} in the building represented by A. Note that E and A can also be the same building graph. The mechanism is graphically depicted in figure 3.1.

3.1.2 Use Cases

The generation mechanism is best understood by example. Note that while we use the wording *use case* we are by no means experts in the architecture or construction domains. Therefore the use cases presented here may seem artificial. In fact they mainly serve the purpose of making the functionality of the presented mechanism clearer. Proper real world use cases will need to be found by domain experts during the use of the mechanism and its implementation. In the following each use case will be explained by 4 figures:

1. A floor plan F_1 representing a level of a multi level building, from which the graph transformation rule is supposed to be extracted. Note that the mechanism can seamlessly be extended for examination of multiple levels at the same time.

2. A graph representation of F_1. For the purpose of simplicity we restrict ourselves to only the relevant part of this graph. In terms of the presented formalism, this is the

Extraction Graph E. In E we have marked the pattern selection $E_{pattern}$ and the modification selection E_{modify}.

3. A floor plan F_2 representing a single level of a multi level building to which the generated graph transformation rule is supposed to be applied. The modification by the graph transformation rule are marked.

4. A graph representation of F_2. In terms of the presented formalism, this is the *Application Graph A*. Again we restrict ourselves to the relevant parts for the application of our rule. In A we have marked the parts corresponding to a match and the modification of the graph by the transformation rule.

Use Case I

The first use case shown in figure 3.2 is very simple and mainly illustrates the distinction between *Extraction* and *Application Graph*. Note that we defined south to be the "bottom" of the floor plan.
The very simple design knowledge we want to extract is the following: Given an outer wall facing south, this wall should have a window. This knowledge is implemented in our extraction building F_1 given the outer wall o_3, its southwards orientation and the window w_1 that is contained in o_3. Looking at the *Extraction Graph E*, it is apparent that the *Pattern Graph* $E_{pattern}$ consists of the context c_3, modelling the southwards direction, o_3 and the edge connecting them. The modification part E_{modify} consists of only the *Window* node w_1 and its connection to o_3.
Applying this graph transformation to the *Application Graph A* of the floor plan F_2, we see that there is only one match: The only outer wall facing southwards is o_4. More specifically, o_4 in A takes the place of o_3 in $E_{pattern}$, while the context c_4 in A takes the place of c_3 in $E_{pattern}$. Formally, we have found an isomorphism m between $E_{pattern}$ and a subgraph of A: $m(o_3) = o_4$ and $m(c_3) = c_4$. As a result of the match, the node w_1 is added to o_4 in the *Application Graph A*.
Note that we have not yet defined the way going back from the modified *Application Graph* A' to the modified floor plan F_2', since we have not specified the exact geometry of the added window w_1. There is no "correct" way to do this, since the whole purpose of the graph representation *is* to abstract away the specific building geometry. In use case III we propose a possible approach for this problem based on relative position.

Use Case II

Use case II shows how a generated rule can be applied multiple times to the same *Application Graph*. A rewrite rule similar to the one described in section 2.3.1 is generated. Use case II

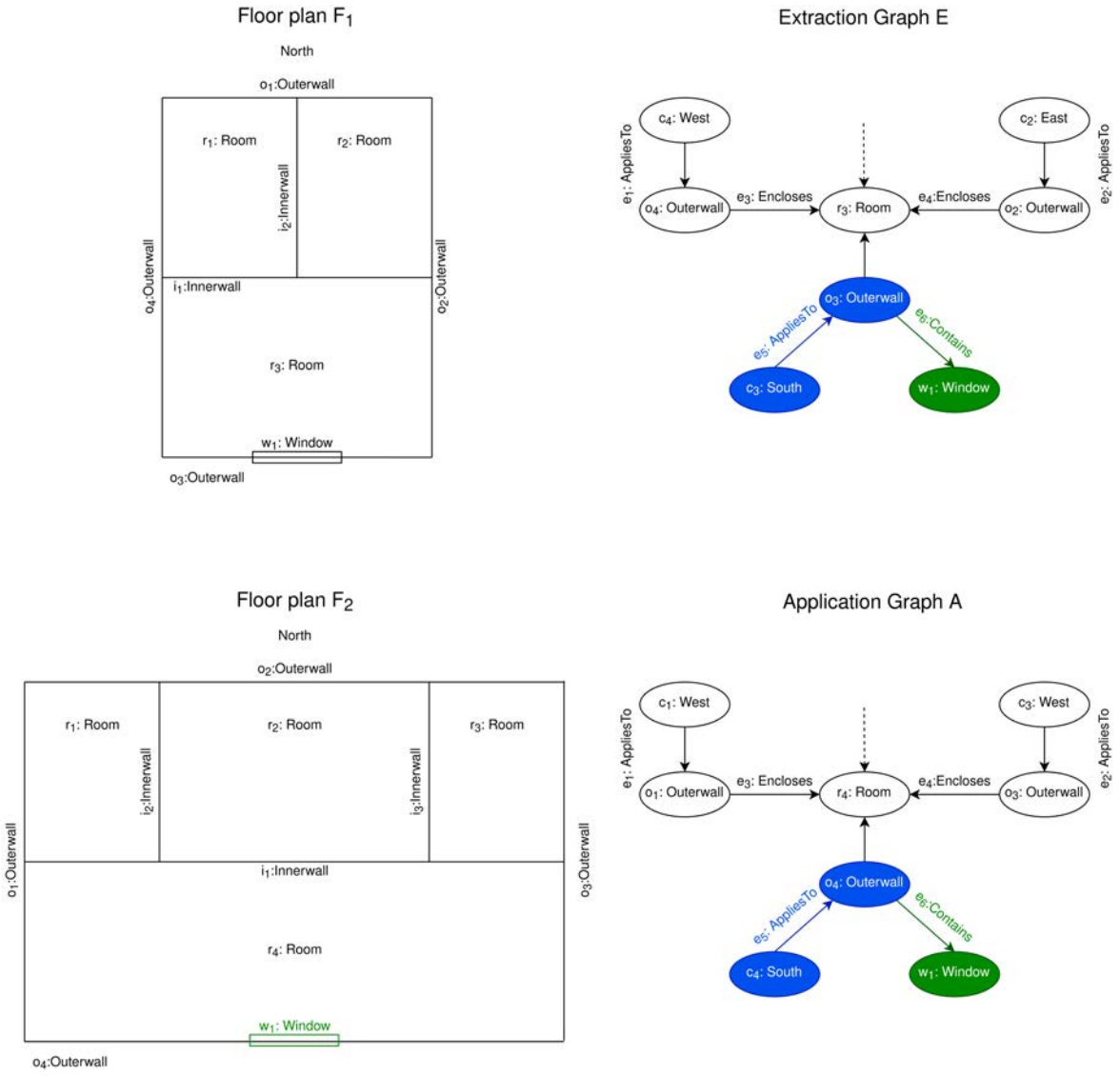

Figure 3.2: Use case I. $E_{pattern}$ and its match in A are marked in blue. E_{modify} and the corresponding modification to A are marked in green.

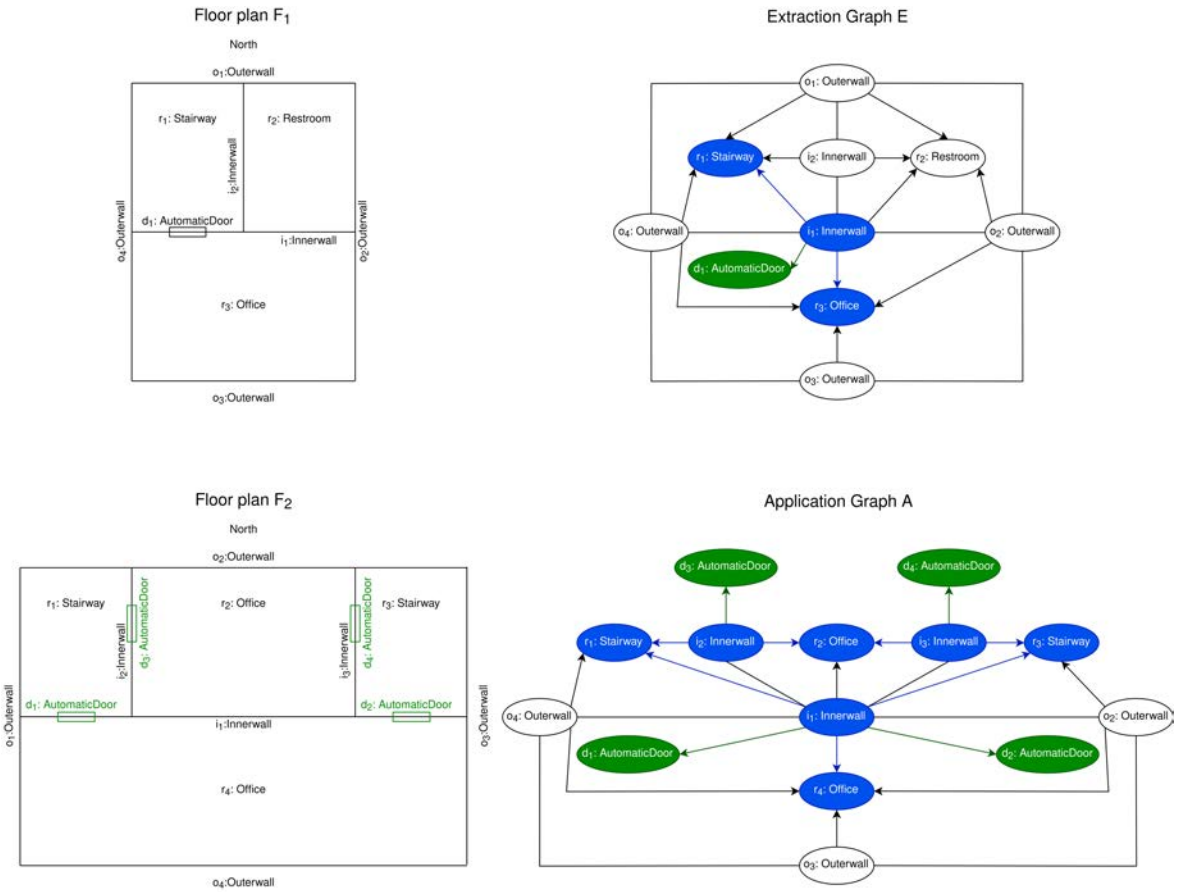

Figure 3.3: Use case II

is shown in figure 3.3.
Floor plan F_1 in use case I and II is very similar. The only difference is the slightly more nuanced semantic of the rooms: The types of the rooms - *Stairway*, *Office*, *Restroom* - are more specific than the previous generic *Room* type. This added semantic allows us to formulate the design knowledge captured with the graph transformation rule 2.4. Repeating its idea in terms of the domain: Doors between office spaces and stairways should be automatically opening and closing.
In the *Extraction Graph* E this pattern is implemented by r_1, r_3, i_1 and the automatic door d_1 contained in i_1. $E_{pattern}$ therefore consists of r_1, r_3, i_1 and the edges connecting those nodes. E_{modify} is the automatic door d_1 and its connecting edge coming from i_1.
The application of this transformation rule to A is interesting, since we can find 4 matches m_1, m_2, m_3, m_4 of $E_{pattern}$ in A. The subgraph r_1, i_1, r_4 and the connecting edges is the first match, r_3, i_1, r_4 the second, r_1, i_2, r_2 the third and r_3, i_2, r_2 the fourth match. As a result of the applications, 2 automatic doors d_1, d_2 are added to i_1, d_3 is added to i_2 and d_4 is added to i_3.
Again note that we have not yet specified a way for going back from the modified *Application Graph* A' to the modifications in the floor plan F_2, making the position of the added doors in F_2 somewhat arbitrary.

Use Case III

Use case III shows a rule generation involving semantic properties of building elements going beyond mere type information of nodes. While we didn't formally define the mechanism for including properties, the extension follows naturally:
For every node (or edge) in the *Pattern Graph* $E_{pattern}$, we also specify which properties should be part of the pattern. We call these properties in the following *pattern properties*. When applying the generated rule to an *Application Graph* A, we require not only the structure of a subgraph of A to match $E_{pattern}$, but also the specified *pattern properties* to be equal.
Similarly, we specify for every node in E_{modify}, which properties should be part of the modification. We call these properties *rewrite properties*. Upon modification of A, added nodes will only have the *rewrite properties*. The values of the *rewrite properties* will be the same as in E_{modify}. One possibly could also want to modify or add properties to nodes in the subgraph of A matching $E_{pattern}$. To make this possible, we also allow the specification of *rewrite properties* in $E_{pattern}$. For this to make sense, the *rewrite properties* and the *pattern properties* may of course not overlap.
The following use case shown in figure 3.4 will make this informal description clearer. F_1 is very similar to the floor plan in use case II, except for the door. Door d_1 is now contained in the inner wall i_2 and is a fire door. The domain knowledge which is supposed to be captured is the following: If there is an inner wall i enclosing a stairway s and the stairway is part

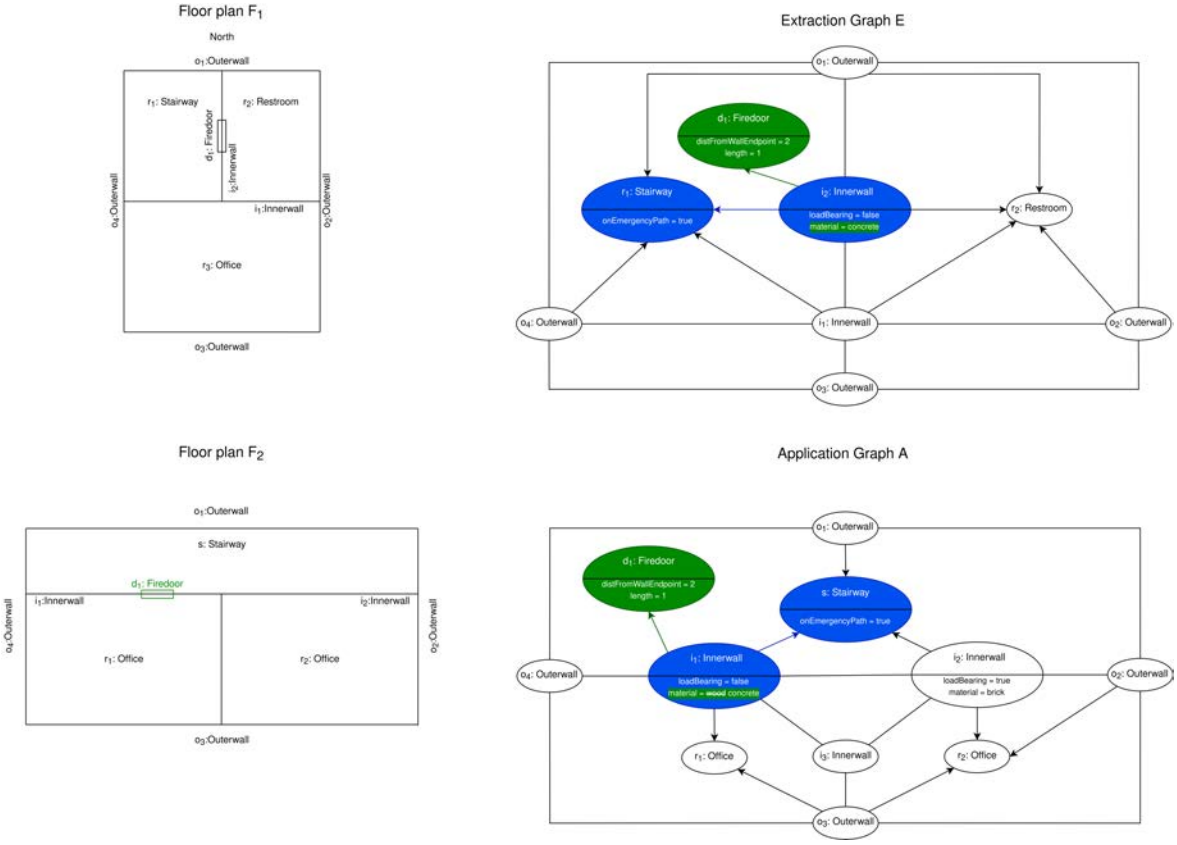

Figure 3.4: Use case III. Note that we have left out the properties of all nodes not relevant for the use case. We have also left out the edge annotations.

of the emergency exit path, insert a fire door into wall i. Furthermore, change the material of i to concrete. Since changing the material of a load-bearing wall can be problematic, we restrict i to be not load-bearing.
This design knowledge is implemented in the subgraph $E_{pattern}$, comprising nodes r_1 and i_2, and E_{modify} comprising d_1. The new part is now that we also specified *pattern properties* and *rewrite properties* for our nodes. This encodes all domain knowledge just described: The stairway has to be part of the emergency path and the inner wall may not be load-bearing. In case of an application, the material of the inner wall should be changed to concrete. In the *rewrite properties* of the fire door we encode geometric information: The width of the fire door (*length*) and its distance from the containing wall endpoint (*distFromWallEndpoint*) for the first time allow us to properly define how an *Application Graph* is to be translated back into specific geometry after the rule application.
Applying the generated rule to A yields the following result: Despite the symmetry of A, only one match is possible, namely the subgraph i_1, s and the connecting edge. The subgraph consisting of i_2 and s does not match the pattern, since i_2 is load-bearing. Therefore a fire door d_1 is only added to i_1 and the material of i_1 is changed from wood to concrete.
The *rewrite properties* for d_1 are interesting, because they allow us to specify the specific geometry of d_1 when going back from the modified *Application Graph* A' to resulting modifications in floor plan F_2. The *distFromWallEndpoint* property specifies the distance from d_1 to a wall endpoint of the wall it is contained in, in our case i_1. This makes it possible to accurately locate d_1 in the floor plan at distance 2 from the right endpoint of i_1. While this is a possible solution to the problem of specifying specific geometry after a graph transformation, it is by no means perfect:

1. There are multiple endpoints of i_1. This can be at least partly solved e.g. by ordering wall endpoints in some total order based on their coordinates.
2. There is no guarantee that the inserted door fits into the wall it is contained in. E.g. if i_1 had a length of less than 1, the rule would introduce a geometric inconsistency.

An alternative approach would be to make the values for length and the distance to an endpoint also relative to the containing wall. In our *Pattern Graph* for example, we could have extracted the absolute distance 2 of the fire door to the end point of the inner wall as a relative distance $\frac{1}{2}$ referring to the length of the containing inner wall. In the application floor plan F_2 this would result in the fire door being inserted in the middle of i_1.
While this approach based on relative size and position is no silver bullet for obtaining useful specific geometry, we still think it is a promising approach to this hard problem.

3.1.3 Design Workflow

Integrating the generation mechanism into a generic BIM workflow poses two main challenges:

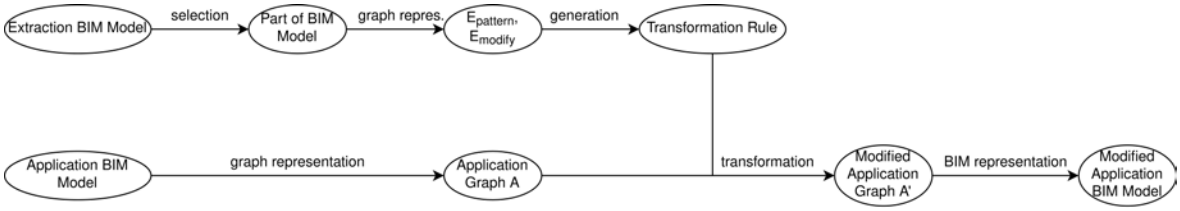

Figure 3.5: A high level view on how the generation mechanism could be integrated into a BIM workflow. Note that the selection now takes place in the BIM model and not the *Extraction Graph*.

1. Hiding the necessary graph representation of the BIM model from the user.
2. Transforming a modified building graph back into specific geometry of a BIM model.

Both challenges are not the main focus of this paper and need more conceptual thought as well as investigation in the real world. For the second challenge we have already sketched a possible approach in use case III (3.1.2). This section will further describe the first challenge to make clear what it needs in order to integrate our mechanism into BIM software.

In our very first example of representing a building as a graph in 2.2, it is quite apparent that the graph representation does not make the structure of the building more apparent for a human. The graph representation in our case is an abstraction mainly done for our formal definition of our rule generation mechanism, or in the end for the software implementing this mechanism. The goal should therefore be to hide the internal graph representation from the BIM user.
This implies that the selection process - what we referred to in 3.1.1 as $E_{pattern}$ and E_{modify} - should not be done in the *Extraction Graph* E, but in the BIM model which E is representing. Figure 3.5 shows the data flow for this case. As a result we have a constraint for the graph representation: Every element selectable in the BIM model must be present in the building graph. For further information about the conversion from a BIM model to a building graph, see (Ismail *et al.*, 2018), (Ismail *et al.*, 2017).
There is also the problem of the implicit nature of relationships between building elements which are represented as edges in the graph representation. A pragmatic approach to solve this problem might be to only let a user select objects (nodes) in the BIM model and implicitly add all relationships (edges) between those objects to the selection. In fact the prototype explained in section 3.2 assumes this behaviour.

3.2 Technical Implementation

In this section we explain a software prototype implementing the mechanism described in 3.1.1. This prototype generates a GrGen rule from an existing *Extraction Graph* E with

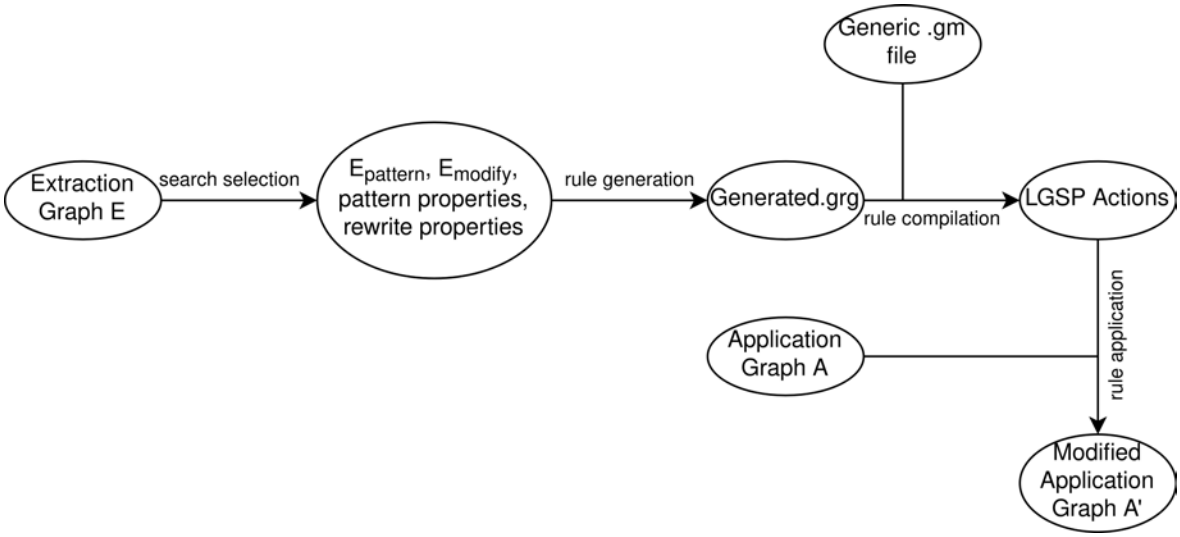

Figure 3.6: The high level data flow of the software prototype.

predefined selection of $E_{pattern}$, E_{modify} which can then be applied to a different (or the same) *Application Graph* to automatically perform graph transformations.
The inclusion of semantic properties as described in use case III (3.1.2) is also implemented. The prototype does however neither include the conversion from a BIM model to its graph representation nor the other way round. Only the core generation mechanism is implemented. The implementation and the generated GrGen rules for all three use cases described in 3.1.2 are explained.

3.2.1 Overview

From a high level view, the prototype works in 4 steps, shown in figure 3.6.

1. Given the *Extraction Graph E*, search for the selection of $E_{pattern}$, E_{modify}, *pattern properties* and *rewrite properties*.
2. Given the selection, generate a corresponding textual representation of a GrGen rule in the *Generated.grg* rule file.
3. Compile this GrGen rule on the fly with the GrGen API using a generic model file *Building.gm*.
4. The compilation yields an *LGSP Actions* object which can be used with the GrGen API to apply the rule to an *Application Graph A*.

3.2.2 Search For Selection

In the context of this paper, the search for selection part is not very interesting. In fact, we have made the following simplification for the selection of $E_{pattern}, E_{modify}$: There is a central node *selected*. $E_{pattern}$ and E_{modify} are limited to direct neighbours of *selected*. More formally, $E_{pattern} \cup E_{modify}$ is restricted to a connected graph of diameter 3. This simplification is sufficient to e.g. implement all use cases presented in section 3.1.2. A general search algorithm can easily be implemented, e.g. using a breadth first search.

3.2.3 Rule Generation

This part of the code is the most interesting, since this is where the actual work of generating the rule is implemented. The code is shown in figure 3.7. Line 3 searches for the central *selected* node. Line 4-8 are merely initializations of data structures used later. The rest of the function has 2 parts:
In the first part beginning in line 12, the *foreach* loop iterates through all neighbours of the central *selected* node. The *line* variable in line 22 defines the graphlet comprising the *selected* node, the neighbour node we are currently iterating over and the edge between them. Based on whether the neighbour node is *selectedForPattern* or *selectedForRewrite*, the graphlet line is added either to $E_{pattern}$ (*patternLines*), or E_{modify} (*rewriteLines*).
In order to handle the semantics of the *pattern properties*, we add two *if* constraints to the pattern part: The *pattern property* must be present and have the same value as in the *Extraction Graph*. This is done in lines 28-34. Symmetrically, in the *modify* part, we need a clause to make sure the properties of the added nodes are set according to the *rewrite properties* specified in the *Extraction Graph*. This is done in lines 46-48. We also add a negative pattern clause, assuring E_{modify} is only added if it not yet present, see line 51. In order to assure unambiguous names, we use the *nodeCount* and *edgeCount* variables to create increasing identifiers for the graphlets, see lines 37-38, 52-53.
The second part, starting in line 56, implements how the individual graphlet lines and constraint lines are combined to form a valid GrGen rule. In lines 56-63 the *joinLines* function is used to ensure proper indentation, making the rule file human readable. The *rule* variable in line 73 combines all the parts into one String which can then be written to a *.grg* rule file for compilation by the GrGen API.

3.2.4 Rule Compilation And Application

This part of the code uses the textual representation of the generated rule with the GrGen API on an *Application Graph*. The code is shown in figure 3.8.
First, the function generates the rule string and writes it to a *.grg* file, see lines 3-5. Afterwards,

```
1  public static String generateRule(LGSPGraphProcessingEnvironment procEnv,
       BuildingNamedGraph graph, BaseRulesActions actions) {
2      // find "central" selected node
3      ISelectable selected = findSelected(graph);
4      // counters to generate unique identifiers for nodes, edges
5      int edgeCount = 0;
6      int nodeCount = 0;
7      // initialize lists storing lines of specific parts of the generated rule
8      ...
9      // check all nodes around selected one
10     // add corresponding line to pattern match / rewrite part
11     IEnumerable<IEdge> edges = selected.Incident;
12      foreach (IEdge e in edges) {
13         ISelectable neighbor = (ISelectable)(e.Source != selected ?
14             e.Source : e.Target);
15         // find direction of edge
16         EdgeDirection dir = e.Type.IsA(graph.GetEdgeType("UEdge")) ?
17             EdgeDirection.Undirected :
18             (e.Source != selected ?
19             EdgeDirection.Incoming : EdgeDirection.Outgoing);
20         String nodeIdent = $"n{nodeCount}";
21         // line defining graphlet comprising selected, neighbor node
22         String line = $"selected {edgePattern(edgeCount, e.Type, dir)}
23             {nodeIdent}:{neighbor.Type};";
24         // if neighbor part of E_pattern
25         if (neighbor.selectedForPattern) {
26             patternLines.Add(line);
27             // make patternProperties part of pattern using if-clause
28             foreach (string prop in neighbor.patternProperties) {
29                 string val;
30                 if (neighbor.properties.TryGetValue(prop, out val)) {
31                     string isIn = $"\"{prop}\" in {nodeIdent}.properties && ";
32                     string sameVal = $"{nodeIdent}.
33                         properties[\"{prop}\"]==\"{val}\";";
34                     ifLines.Add(isIn + sameVal);
35                 }
36             }
37             nodeCount++;
38             edgeCount++;
39         }
40         // neighbor part of E_modify
41         else if (neighbor.selectedForRewrite) {
42             rewriteLines.Add(line);
43             // handle addition of properties to nodes
44             foreach (string prop in neighbor.rewriteProperties) {
45                 string val;
46                 if (neighbor.properties.TryGetValue(prop, out val)) {
47                     string setProp = $"{nodeIdent}.properties.add(\"{prop
                           }\",\"{val}\");";
48                     evalLines.Add(setProp);
49                 }
50             }
51             patternLinesNegative.Add($"negative {{ {line} }}");
52             nodeCount++;
53             edgeCount++;
54         }
55     }
```

Figure 3.7: The generateRule function

```
56      // add parts together to form a valid rule
57      String header = "#using \"Building.gm\"";
58      String pattern = joinLines(patternLines, "\t");
59      String patternNegatives = joinLines(patternLinesNegative, "\t");
60      String ifStmt = joinLines(ifLines, "\t\t");
61      String rewrite = joinLines(rewriteLines, "\t\t");
62      String eval = joinLines(evalLines, "\t\t\t");
63      String negativeRewriteProps = joinLines(negativeRewritePropsLines, "\t\t\t
            ");
64      String negativeRewritePropsClause =
65          negativeRewritePropsLines.Count > 0 ?
66              "\tnegative {\n" +
67                  "\t\tif {\n" +
68                      negativeRewriteProps +
69                  "\t\t}\n" +
70              "\t}\n" :
71              "";
72
73      String rule =
74          header + "\n" +
75          "rule generated {\n" +
76              pattern +
77              patternNegatives +
78              "\tif {\n" +
79                  ifStmt +
80              "\t}\n" +
81              negativeRewritePropsClause +
82              "\tmodify {\n" +
83                  rewrite +
84                  "\t\teval {\n" +
85                      eval +
86                  "\t\t}\n" +
87              "\t}\n" +
88            "}";
89      return rule;
90  }
```

Figure 3.7: The generateRule function (cont.)

```
1  public static void compileAndApply() {
2    ...
3    // generate rule
4    String rule = generateRule(extractionProcEnv, extractionGraph,
         extractionActions)
5    System.IO.File.WriteAllText("Generated.grg", rule);
6
7    // create dummy application graph
8    LGSPNamedGraph applicationGraph;
9    LGSPActions applicationActions;
10
11   // compile generated rule file
12   new LGSPBackend().CreateNamedFromSpec($"Generated.grg", null, 0,
13       out applicationGraph, out applicationActions);
14   // load actual application graph
15   ...
16   // apply generated rule to application graph
17   LGSPGraphProcessingEnvironment applicationProcEnv =
18       new LGSPGraphProcessingEnvironment(applicationGraph, applicationActions);
19   applicationProcEnv.ApplyGraphRewriteSequence("generated*");
20 }
```

Figure 3.8: The function compiling and applying the generated rule

the *CreateNamedFromSpec* function of the GrGen API is used in lines 8-13 to compile the rule file. The inclusion of the graph model file *Building.gm* is done using the first line in the rule file and hence does not have to be explicitly mentioned in the code. This compilation types our *applicationGraph* variable in which we can then load the actual *Application Graph* in lines 14-15.
The resulting *applicationActions* variable can then be used in the processing environment *applicationProcEnv* to apply the generated rule to the *Application Graph* in lines 16-19. Note that GrGen uses a simple rule application language. The $*$ operator tries to apply the rule as often as it matches. In order to assure a match is only rewritten once, it is therefore important to have the *negative* clauses mentioned in section 3.2.3. This will become clearer looking at the generated rules in the following use cases.

3.2.5 Generic Building Graph Model

Since GrGen needs a graph model file for the compilation, there are two options:

1. Generate a graph model file on the fly for each *Extraction Graph* and each *Application Graph*.

2. Define a generic graph model.

```
# Building.gm
node class Selectable {
  selected: boolean;
  selectedForPattern: boolean;
  selectedForRewrite: boolean;
  properties: map<string, string>;
  patternProperties: array<string>;
  rewriteProperties: array<string>;
}

node class Spatial extends Selectable {
  ps: array<int>;
}
node class Line extends Selectable {
  a_x: int;
  a_y: int;
  b_x: int;
  b_y: int;
}

undirected edge class ConnectedTo extends UEdge {
  x:int;
  y:int;
}
edge class Contains extends Edge;
edge class Encloses extends Edge;
```

Figure 3.9: The generic GrGen graph model file used for the rule compilation

Generating the model on the fly has the problem of compatibility of *Extraction* and *Application Graph*. We therefore decided to define a generic building graph model for our prototype, which is shown in figure 3.9. The only GrGen node types in this generic model are the following:

1. *Selectable* for selectable elements of the building model.
2. *Line* for elements such as windows and doors, which are (for the sake of simplicity) in this paper modelled as a line between 2 points.
3. *Spatial* for spatial elements which are (for the sake of simplicity) in this paper modelled as polygons.

Note that the *Line* and *Spatial* types both inherit from the *Selectable* type and therefore share its properties. Node properties are saved in the *properties* map attribute. This attribute maps property names to their values (in String representation) ensuring generic handling of properties at the cost of the missing type information of the corresponding property. The specific type of a node is tracked as part of this property map. All other attributes of the *Selectable* class are used to specify the selected parts of a graph for the rule generation ($E_{pattern}$, E_{modify}, *pattern properties, rewrite properties).*

```
1  rule generated {
2      selected:Line -e1:Contains-> n1:Selectable;
3      negative { selected -e0:Contains-> n0:Line; }
4      if {
5         "type" in selected.properties
6              && selected.properties["type"]=="OuterWall";
7         "type" in n1.properties && n1.properties["type"]=="Context";
8         "orientation" in n1.properties && n1.properties["orientation"]=="south";
9      }
10     modify {
11         selected -e0:Contains-> n0:Line;
12         eval {
13             n0.properties.add("type","Window");
14             n0.properties.add("distFromWallEndpoint","3");
15             n0.properties.add("length","2");
16         }
17     }
18 }
```

Figure 3.10: The generated rule for use case I

3.2.6 Use Cases

This section explains the GrGen rules generated by our prototype for the use cases in section 3.1.2. Since we already thoroughly explained the uses cases in this previous section, we only explain the generated GrGen rules in the following. The code for compilation and application are simple enough to not needing further mentioning here.

Use Case I

The first use case handled the addition of a window to outer walls with southwards orientation. Line 2 of the generated rule shown in figure 3.10 defines the basic graphlet pattern. Note that since we have provided GrGen with the generic Building Model file, there are no specific GrGen types present. We can find the specific type information for the graphlet in the *if* clause: The node *selected* must be of type *Outerwall* while the node n_1 needs to be of type *Context*. n_1 also needs to have the property *orientation* with value *south*.
Note that the *negative* clause in line 2 ensures that the rule can only be applied once to the same outer wall. The *modify* clause adds a new node n_0 to the *selectable* node and sets its type to *Window*. Note that we have also included the properties *length* and *distFromWallEndpoint* to specify the specific geometry of n_0.

Use Case II

Use case II extracted the pattern of adding automatic doors between office rooms and stairways into a rule. The rule generated by our prototype can be seen in figure 3.11.

```
1  rule generated {
2      selected:Line -e0:Encloses-> n0:Spatial;
3      selected -e1:Encloses-> n1:Spatial;
4      negative { selected -e2:Contains-> n2:Line; }
5      if {
6          "type" in n0.properties && n0.properties["type"]=="Stairway";
7          "type" in n1.properties && n1.properties["type"]=="Office";
8          "type" in selected.properties
9              && selected.properties["type"]=="Innerwall";
10     }
11     modify {
12         selected -e2:Contains-> n2:Line;
13         eval {
14             n2.properties.add("type","AutomaticDoor");
15             n2.properties.add("distFromWallEndpoint","1");
16             n2.properties.add("length","1");
17         }
18     }
19 }
```

Figure 3.11: The generated rule for use case II

Note that the *if* clause in lines 5-9 merely specifies type information. The pattern graphlet in the first two lines therefore has the following semantics: Match a node *selected* of type *Innerwall* which is connected to both, a node n_0 of type *Stairway* and a node n_1 of type *Office*.
The *negative* pattern in line 4 again assures that there is no door already present in the matched inner wall, in turn making sure the rule is only applicable once to the same wall. The modification clause adds a node n_0 with type *AutomaticDoor* to the matched wall. Again we have also added *rewrite properties* for the specification of the added doors geometry.

Use Case III

The design pattern used in use case III had two parts:

1. Adding fire doors to walls enclosing a stairway being part of the emergency path.
2. Changing the matched walls material to concrete. Since this step is problematic for load bearing walls, we restricted the rules applicability to walls not being load-bearing.

The GrGen rule generated by our prototype can be seen in 3.12.
The graphlet in line 2 - taken together with the type information of lines 5-7 - encodes the following pattern: Match a node *selected* of type *Innerwall* which encloses a *Stairway* node n_0. The additional constraints in the *if* clause (lines 8-11) require the inner wall not to be load bearing and the stairway to be part of the emergency path.
The two *negative* clauses again make sure the modifications which would be done by the

```
rule generated {
    selected:Line -e0:Encloses-> n0:Spatial;
    negative { selected -e1:Contains-> n1:Line; }
    if {
        "type" in selected.properties
            && selected.properties["type"]=="Innerwall";
        "type" in n0.properties && n0.properties["type"]=="Stairway";
        "loadBearing" in selected.properties
            && selected.properties["loadBearing"]=="false";
        "emergencyPath" in n0.properties
            && n0.properties["emergencyPath"]=="true";
    }
    negative {
        if {
            "material" in selected.properties
                && selected.properties["material"] == "concrete";
        }
    }
    modify {
        selected -e1:Contains-> n1:Line;
        eval {
            n1.properties.add("type","FireDoor");
            selected.properties.add("material","concrete");
            n1.properties.add("distFromWallEndpoint","2");
            n1.properties.add("length","1");
        }
    }
}
```

Figure 3.12: The generated rule for use case III

rule application are not already present in the application graph. Note that the second *negative* clause in line 13-18 requires the material of the inner wall not to already be *concrete*, since changing its material to concrete would be part of the modification done by the rule application. This behaviour is pessimistic in regard to the rule application. In fact the first negative clause is sufficient to ensure the rule is only applicable once to the same wall. To evaluate which behaviour is more suited to a real world use of the mechanism, a user test would need to be conducted.
The *modify* clause in line 19-25 implements the addition of a fire door n_1 with geometric properties to the inner wall *selected* and changes the walls material to *concrete* as intended (line 23).

Chapter 4

Conclusion

We have shown how graph transformation rules can formalise implicit design knowledge and how to implement and apply those rules using the GrGen software. Representing buildings as graphs is however still an ongoing research topic which has to be settled before graph transformations can be fully utilised in the context of buildings.
This holds especially true for our generation mechanism and its integration into actual BIM software. Nonetheless we think that our rule generation mechanism and its prototypical software implementation provide a valuable insight into how the graph transformation formalism might be integrated into a BIM workflow. Especially, since our mechanism does not require the user to be proficient in any formalism. The real world usefulness of our mechanism however will of course have to be evaluated by domain experts once it is integrated into BIM software.

Bibliography

Abualdenien, J. & Borrmann, A. (2019). A meta-model approach for formal specification and consistent management of multi-LOD building models. *Advanced Engineering Informatics* 40(1474-0346), S. 135–153.

Borrmann, A., König, M., Koch, C. & Beetz, J. (2018). *Building Information Modeling Technology Foundations and Industry Practice: Technology Foundations and Industry Practice*. Springer.

Chakrabarti, A., Shea, K., Stone, R., Cagan, J., Campbell, M., Hernandez, N. V. & Wood, K. L. (2011, 06). Computer-Based Design Synthesis Research: An Overview. *Journal of Computing and Information Science in Engineering* 11(2). 021003.

Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A. & Corradini, A. (1997, 02). Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. Volume 247-312, S. 247–312.

Ehrig, H., Prange, U. & Taentzer, G. (2004). Fundamental Theory for Typed Attributed Graph Transformation. In: H. Ehrig, G. Engels, F. Parisi-Presicce, & G. Rozenberg (Hrsg.), *Graph Transformations*, Berlin, Heidelberg, S. 161–177. Springer Berlin Heidelberg.

Geiß, R., Batz, G. V., Grund, D., Hack, S. & Szalkowski, A. (2006). GrGen: A Fast SPO-Based Graph Rewriting Tool. In: A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, & G. Rozenberg (Hrsg.), *Graph Transformations*, Berlin, Heidelberg, S. 383–397. Springer Berlin Heidelberg.

Ismail, A., Nahar, A. & Scherer, R. (2017, 07). Application of graph databases and graph theory concepts for advanced analysing of BIM models based on IFC standard.

Ismail, A., Strug, B. & Ślusarczyk, G. (2018). Building Knowledge Extraction from BIM/IFC Data for Analysis in Graph Databases. In: L. Rutkowski, R. Scherer, M. Korytkowski, W. Pedrycz, R. Tadeusiewicz, & J. M. Zurada (Hrsg.), *Artificial Intelligence and Soft Computing*, Cham, S. 652–664. Springer International Publishing.

Jakumeit, E., Buchwald, S. & Kroll, M. (2010, 01). GrGen.NET - The expressive, convenient and fast graph rewrite system. *STTT* 12, S. 263–271.

Rozenberg, G. (Hrsg.) (1997). *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. USA: World Scientific Publishing Co., Inc.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich den vorliegenden IDP-Report selbstständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Ich versichere außerdem, dass die vorliegende Arbeit noch nicht einem anderen Prüfungsverfahren zugrunde gelegen hat.

München, 27. März 2020

Gunther Bidlingmaier

Gunther Bidlingmaier